El Lenguaje de Programacion Rust

Jose Narvaez

Published with GitBook

Tabla de contenido

% El Lenguaje de Programación Rust

¡Bienvenido! Este libro te enseñará acerca del Lenguaje de Programación Rust. Rust es un lenguaje de programación de sistemas enfocado en tres objetivos: seguridad, velocidad y concurrencia. Rust logra estos objetivos sin tener un recolector de basura, haciendolo util para un numero de casos de uso para los cuales otros lenguajes no son tan buenos: embeber en otros lenguajes, programas con requerimientos específicos de tiempo y espacio, escritura de código de bajo nivel, como controladores de dispositivo y sistemas operativos. Rust mejora por sobre los lenguajes actuales en este nicho a través de un numero de chequeos en tiempo de compilación que no incurren ninguna penalidad en tiempo de ejecución, eliminando al mismo tiempo las condiciones de carrera. Rust también implementa 'abstracciones con cero costo', abstracciones que se sienten como las de un lenguaje de alto nivel. Aun así, Rust permite control preciso tal y como un lenguaje de bajo nivel lo haria.

“El Lenguaje de Programación Rust” esta dividido en siete secciones. Esta introducción es la primera. Después de esta:

- Primeros Pasos - Configura tu maquina para el desarrollo en Rust.
- Aprende Rust - Aprende programación en Rust a través de pequeños proyectos.
- Rust Efectivo - Conceptos de alto nivel para escribir excelente código Rust.
- Sintaxis y Semántica - Cada pieza de Rust, dividida en pequenos pedazos.
- Rust Nocturno - Características todavía no disponibles en builds estables.
- Glosario - Referencia de los términos usados en el libro.
- Investigación Académica - Literatura que influencio Rust.

Después de leer esta introducción, dependiendo de tu preferencia, querrás leer ‘Aprende Rust’ o ‘Sintaxis y Semántica’: ‘Aprende Rust’ si quieres comenzar con un proyecto o ‘Sintaxis y Semántica’, si prefieres comenzar por lo más pequeño aprendiendo un único concepto detalladamente antes de moverte al siguiente. Abundantes enlaces cruzados conectan dichas partes.

Contribuyendo

Los archivos fuente de los cuales este libro es generado pueden ser encontrados en Github: github.com/goyox86/elpr-sources

Una breve introducción a Rust

¿Es Rust un lenguaje en el cual estarías interesado? Examinemos unos pequeños ejemplos de código que demuestran algunas de sus fortalezas.

El concepto principal que hace único a Rust es llamado 'pertenencia' ('ownership'). Considera este pequeño ejemplo:

```
fn main() {
    let mut x = vec!["Hola", "mundo"];
}
```

Este programa crea un una variable llamada `x` .El valor de esta variable es `Vec<T>` , un 'vector', que creamos a través de una macro definida en la biblioteca estandar. Esta macro se llama `vec` , las macros son invocadas con un `!` . Todo esto siguiendo un principio general en Rust: hacer las cosas explícitas. Las macros pueden hacer cosas significativamente más complejas que llamadas a funciones, es por ello que son visualmente distintas. El `!` ayuda también al análisis sintáctico, haciendo la escritura de herramientas más fácil, lo cual es también importante.

Hemos usado `mut` para hacer `x` mutable: En Rust las variables son inmutables por defecto. Más tarde en este ejemplo estaremos mutando este vector.

Es importate mencionar que no necesitamos una anotación de tipos aquí: si bien Rust es estaticamente tipado, no necesitamos anotar el tipo de forma explicita. Rust posee inferencia de tipos para balancear el poder de el tipado estático con la verbosidad de las anotaciones de tipos.

Rust prefiere asignación de memoria desde la pila que desde el montículo: `x` es puesto directamente en la pila. Sin embargo, el tipo `Vec<T>` asigna espacio para los elementos del vector en el montículo. Si no estas familiarizado con esta distinción puedes ignorarla por ahora o echar un vistazo 'La Pila y el Monticulo'. Rust como un lenguaje de programación de sistemas, te da la habilidad de controlar como la memoria es asignada, pero como estamos comenzando no es tan relevante.

Anteriormente mencionamos que la 'pertenencia' es nuevo concepto clave en Rust. En terminología Rust, `x` es el 'dueño' del vector. Esto significa que cuando `x` salga de ámbito, la memoria asignada a el vector sera liberada. Esto es hecho por el compilador de Rust de manera deterministica, sin la necesidad de un mecanismo como un recolector de basura. En otras palabras, en Rust, no haces llamadas a funciones como `malloc` y `free` explícitamente: el compilador determina de manera estática cuando se necesita asignar o liberar memoria, e inserta esas llamadas por ti. Errar es de humanos, pero los compiladores nunca olvidan.

Agreguemos otra línea a nuestro ejemplo:

```
fn main() {
    let mut x = vec!["Hola", "mundo"];

    let y = &x[0];
}
```

Hemos introducido otra variable, `y`. En este caso, `y` es una 'referencia' a el primer elemento de el vector. Las referencias en Rust son similares a los apuntadores en otros lenguajes, pero con chequeos de seguridad adicionales en tiempo de compilación. Las referencias interactuan con el sistema de pertenencia a través de el 'prestamo' ('borrowing'), ellas toman prestado a lo que apuntan, en vez de adueñarse de ello. La diferencia es que cuando la referencia salga de ámbito, la memoria subyacente no sera liberada. De ser ese el caso estaríamos liberando la misma memoria dos veces, lo cual es malo.

Agreguemos una tercera línea. Dicha línea luce inocente pero causa un error de compilación:

```
fn main() {
    let mut x = vec!["Hola", "mundo"];

    let y = &x[0];

    x.push("foo");
}
```

`push` es un metodo en los vectores que agrega un elemento al final del vector. Cuando tratamos de compilar el programa obtenemos un error:

```
error: cannot borrow `x` as mutable because it is also borrowed as immutable
    x.push("foo");
    ^
note: previous borrow of `x` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `x` until the borrow ends
    let y = &x[0];
             ^
note: previous borrow ends here
fn main() {

}
^
```

¡Uff! El compilador de Rust algunas veces puede proporcionar errores bien detallados y esta vez una de ellas. Como el error lo explica, mientras hacemos la variable mutable no podemos llamar a `push`. Esto es porque ya tenemos una referencia a un elemento del vector, `y`. Mutar algo mientras existe una referencia a ello es peligroso, porque podemos

invalidar la referencia. En este caso en especifico, cuando creamos el vector, solo hemos asignado espacio para dos elementos. Agregar un tercero significaría asignar un nuevo segmento de memoria para todos los elementos, copiar todos los valores anteriores y actualizar el apuntador interno a esa memoria. Todo eso esta bien. El problema es que `y` no seria actualizado, generando un 'puntero colgante'. Lo cual esta mal. Cualquier uso de `y` seria un error en este caso, y el compilador nos ha prevenido de ello.

Entonces, ¿cómo resolvemos este problema? Hay dos enfoques que podríamos tomar. El primero es hacer una copia en lugar de una referencia:

```
fn main() {
    let mut x = vec!["Hola", "mundo"];

    let y = x[0].clone();

    x.push("foo");
}
```

Rust tiene por defecto semántica de movimiento, entonces si queremos hacer una copia de alguna data, llamamos el método `clone()`. En este ejemplo `y` ya no es una referencia a el vector almacenado en `x`, sino una copia de su primer elemento, `"Hola"`. Debido a que no tenemos una referencia nuestro `push()` funciona perfectamente.

Si realmente queremos una referencia, necesitamos otra opción: asegurarnos de que nuestra referencia salga de ámbito antes que tratamos de hacer la mutación. De esta manera:

```
fn main() {
    let mut x = vec!["Hola", "mundo"];

    {
        let y = &x[0];
    }

    x.push("foo");
}
```

Con el par adicional de llaves hemos creado un ámbito interno. `y` saldrá de ámbito antes que llamemos a `push()`, entonces no hay problema.

Este concepto de pertenencia no es solo bueno para prevenir punteros colgantes, sino un conjunto entero de problemas, como invalidación de iteradores, concurrencia y más.

% Primeros Pasos

Esta primera sección del libro te pondrá andando en Rust y sus herramientas. Primero, instalaremos Rust. Después, el clásico programa ‘Hola Mundo’. Finalmente, hablaremos de Cargo, el sistema de construcción y manejador de paquetes de Rust.

% Instalando Rust

¡El primer paso para usar Rust es instalarlo! Existen un número de maneras para instalar Rust, pero la más fácil es usar el script `rustup`. Si estás en Linux o una Mac, todo lo que necesitas hacer es (sin escribir los `$`s, solo indican el inicio de cada comando):

```
$ curl -sf -L https://static.rust-lang.org/rustup.sh | sh
```

Si estás preocupado acerca de la inseguridad potencial de usar `curl | sh`, por favor continúa leyendo y ve nuestra declaración. Siéntete libre de usar la version de dos pasos de la instalación y examinar nuestro script:

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh
```

Si estás en Windows, por favor descarga el instalador de 32 bits o el instalador de 64 bits y ejecutalo.

Desintalando

Si decides que ya no quieres más Rust, estaremos un poco tristes pero está bien, ningún lenguaje de programación es para todo el mundo. Simplemente ejecuta el script de desinstalación:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

Si usaste el instalador de Windows simplemente vuelve a ejecutar el `.msi` y este te dará la opción de desinstalar.

Algunas personas, con mucho derecho, se ven perturbadas cuando les dices que deben `curl | sh`. Básicamente, al hacerlo, estás confiando en que la buena gente que mantiene Rust no va a hackear tu computador y hacer cosas malas. ¡Eso es buen instinto! Si eres una de esas personas por favor echa un vistazo a la documentación acerca de compilar Rust desde el codigo fuente, o la página oficial de descargas de binarios.

¡Ah!, también debemos mencionar las plataformas soportadas oficialmente:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 o mayor, varias distribuciones), x86 and x86-64
- OSX 10.7 (Lion) o mayor, x86 y x86-64

Nosotros probamos Rust extensivamente en dichas plataformas, y algunas otras pocas, como Android. Estas son las que garantizan funcionar debido a que tienen la mayor cantidad de pruebas.

Finalmente, un comentario acerca de Windows. Rust considera Windows como una plataforma de primera clase en cuanto a release, pero si somos honestos, la experiencia en Windows no es tan integrada como la de Linux/OS X. ¡Estamos trabajando en ello! Si algo no funciona, es un bug. Por favor haznoslo saber. Todos y cada uno de los commits son probados en Windows justo como en cualquier otra plataforma.

Si ya tienes Rust instalado, puedes abrir una consola y escribir:

```
$ rustc --version
```

Deberías ver el número de versión, hash del commit, fecha del commit y la fecha de compilación:

```
rustc 1.0.0 (a59de37e9 2015-05-13) (built 2015-05-14)
```

Si lo has visto, Rust ha sido instalado satisfactoriamente. ¡Felicitaciones!

Este instalador también instala una copia de la documentación localmente para que puedas leerla aún estando desconectado. En sistemas UNIX, `/usr/local/share/doc/rust` es la locación. En Windows, es en el directorio `share/doc`, dentro de donde sea que hayas instalado Rust.

Si no, hay un número de lugares en los que puedes obtener ayuda. El más fácil es el canal de IRC #rust en irc.mozilla.org, al cual puedes acceder con Mibbit. Sigue ese enlace y estarás hablando con Rusteros (a tonto apodo que usamos entre nosotros), y te ayudaremos. Otros excelentes recursos son el foro de usuarios, y Stack Overflow.

% ¡Hola, mundo!

Ahora que has instalado Rust, escribamos tu primer programa. Es tradición que tu primer programa en cualquier lenguaje sea uno que imprima el texto “¡Hola, mundo!” a la pantalla. Lo bueno de comenzar con un programa tan simple es que verificas no solo que el compilador esta instalado, sino que esta funcionando. Imprimir información a la pantalla es una cosa muy común.

La primera cosa que debemos hacer es crear un archivo en donde poner nuestro código. A mi me gusta crear un directorio `proyectos` en mi directorio de usuario y mantener todos mis proyectos allí. A Rust no le importa dónde reside el código.

Esto lleva en cuestión a otro asunto que debemos aclarar: esta guía asumirá que posees una familiaridad básica con la linea de comandos. Rust no demanda nada con respecto a tus herramientas de edición o dónde vive tu código. Si prefieres un IDE a la interfaz de linea de comandos, probablemente deberías echar un vistazo a SolidOak, o donde sea que los plugins estén para tu IDE favorito. Existen un número de extensiones de calidad variada en desarrollo por la comunidad. El equipo detrás de Rust también publica plugins para varios editores. Configurar tu editor o IDE escapa de los objetivos de éste tutorial, chequea la documentación para tu configuración específica.

Dicho esto, creemos un directorio en nuestro directorio de proyectos.

```
$ mkdir ~/proyectos
$ cd ~/proyectos
$ mkdir hola_mundo
$ cd hola_mundo
```

Si estás en Windows y no estás usando PowerShell, el `~` podría no funcionar. Consulta la documentacion específica para tu terminal para mayor detalle.

Creemos un nuevo archivo de código fuente. Llamaremos `main.rs` a nuestro archivo. Los archivos Rust terminan con la extensión `.rs`. Si estás usando más de una palabra en tu nombre de archivo, usa un subguion: `hola_mundo.rs` en vez de `holamundo.rs`.

Ahora que tienes tu archivo abierto escribe esto en el:

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

Guarda los cambios en el archivo, y escribe lo siguiente en la ventana de tu terminal:

```
$ rustc main.rs
$ ./main # o main.exe en Windows
¡Hola, mundo!
```

¡Éxito! Ahora veamos que ha pasado en detalle.

```
fn main() {

}
```

Estas lineas definen una *función* en Rust. La función `main` es especial: es el principio de todo programa Rust. La primera línea dice: "Estoy declarando una función llamada `main` la cual no recibe argumentos y no retorna nada." Si existieran argumentos, estos irían dentro de paréntesis (`(` and `)`), y debido a que no estamos retornando nada de esta función, podemos omitir el tipo de retorno completamente. Llegaremos a esto más adelante.

También notaras que la función está envuelta en llaves (`{` and `}`). Rust requiere dichas llaves delimitando todos los cuerpos de función. Es también considerado buen estilo colocar la llave de apertura en la misma linea de la declaración de la función, con un espacio intermedio.

Lo siguiente es esta linea:

```
    println!("¡Hola, mundo!");
```

Esta línea hace todo el trabajo en nuestro pequeño programa. Hay un número de detalles que son importantes aquí. El primero es que está sangrado con cuatro espacios, no tabulaciones. Por favor, configura tu editor a insertar cuatro espacios con la tecla tab. Proveemos algunas configuraciones de ejemplo para varios editores.

El segundo punto es la parte `println!()`. Esto es llamar a una macro Rust macro, que es como se hace metaprogramación en Rust. Si esto fuese en cambio una función, luciria asi: `println()`. Para nuestros efectos, no necesitamos preocuparnos acerca de esta diferencia. Solo saber que algunas veces verás `!`, y que esto significa que estas llamando a una macro en vez de una función normal. Rust implementa `println!` como una macro en vez de como una función por buenas razones, pero eso es un tópico avanzado. Una última cosa por mencionar: Las macros en Rust son diferentes de las macros en C, si las has usado. No estés asustado de usar macros. Llegaremos a los detalles eventualmente, por ahora simplemente debes confiar en nosotros.

A continuación, `"¡Hola, mundo!"` es una cadena de caracteres. Las cadenas de caracteres son un tópico sorprendentemente complejo en lenguajes de programación de sistemas, y ésta concretamente es una cadena de carateres 'asiganda estáticamente'. Si te gustaría leer acerca de asignación de memoria, echa un vistazo a la pila y el montículo, pero por ahora no necesitas hacerlo si no lo deseas. Pasamos esta cadena de caracteres como un argumento a `println!` quien a su vez imprime la cadena de caracteres a la pantalla. ¡Fácil!

Finalmente, la linea termina con un punto y coma (`;`). Rust es un lenguaje orientado a expresiones, lo que significa que la mayoría de las cosas son expresiones, en vez de sentencias. El `;` se usa para indicar que la expresión ha terminado, y que la siguiente está lista para comenzar. La mayoría de las líneas de código en Rust terminan con un `;` .

Finalmente, compilar y ejecutar nuestro programa. Podemos compilar con nuestro compilador `rustc` pasándole el nombre de nuestro archivo de código fuente:

```
$ rustc main.rs
```

Esto es similar a `gcc` or `clang` , si provienes de C o C++. Rust emitirá un binario ejecutable. Puedes verlo con `ls` :

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

O en Windows:

```
$ dir
main.exe  main.rs
```

Hay dos archivos: nuestro código fuente, con la extensión `.rs` , y el ejecutable (`main.exe` en Windows, `main` en los demás)

```
$ ./main  # o main.exe en Windows
```

Lo anterior imprime nuestro texto `¡Hola, mundo!` a nuestra terminal.

Si provienes de un lenguaje dinámico como Ruby, Python o Javascript, probablemente no estés acostumbrado a ver estos dos pasos como separados. Rust es un lenguaje compilado, lo cual significa que puedes compilar tu programa, dárselo a alguien más, y éste no necesita tener Rust instalado. Si les das a alguien un archivo `.rb` o `.py` o `.js` , este necesita tener una implementación de Ruby/Python/JavaScript, pero sólo necesitas un comando para ambos, compilar y ejecutar tu programa. Todo es acerca de balance entre ventajas/desventajas en el diseño de lenguajes, y Rust ha elegido.

Felicitaciones, has escrito oficialmente un programa Rust. ¡Eso te convierte en un programador Rust! Bienvenido.

A continuación me gustaría presentarte otra herramienta, Cargo, el cual es usado para escribir programas Rust para el mundo real. Solo usar `rustc` esta bien para cosas simples, pero a medida que tu proyecto crece, necesitarás algo que te ayude a administrar todas las opciones que este tiene, así como hacer fácil compartir el codigo con otras personas y proyectos.

% Hola, Cargo!

Cargo es una herramienta que los Rusteros usan como ayuda para administrar sus proyectos Rust. Cargo esta actualmente en estado pre-1.0, y debido a ello es todavía un trabajo en proceso. Sin embargo ya es lo suficientemente bueno para muchos proyectos Rust, y se asume que los proyectos Rust usaran Cargo desde el principio.

Cargo administra tres cosas: la compilación de tu código, descarga de las dependencias que tu código necesita, y la compilación de dichas dependencias. En primera instancia tu código no tendrá ninguna dependencia, es por ello que estaremos usando solo la primera parte de la funcionalidad de Cargo. Eventualmente, agregaremos mas. Debido a que comenzamos usando Cargo desde el principio sera fácil agregar después.

Si has instalado Rust a través de los instaladores oficiales entonces deberías tener Cargo. Si has instalado Rust de alguna otra manera, podrías echar un vistazo a el README de Cargo para instrucciones especificas acerca de como instalarlo.

Migrando a Cargo

Convirtamos Hola Mundo a Cargo.

Para Carguificar nuestro proyecto, necesitamos dos cosas: Crear un archivo de configuración `Cargo.toml`, y colocar nuestro archivo de código fuente en el lugar correcto. Hagamos esa parte primero:

```
$ mkdir src
$ mv main.rs src/main.rs
```

Nota que debido a que estamos creando un ejecutable, usamos `main.rs`. Si quisiéramos crear una biblioteca, deberíamos usar `lib.rs`. Locaciones personalizadas para el punto de entrada pueden ser especificadas con una clave [`[[lib]]` or `[[bin]]`]crates-custom en el archivo TOML descrito a continuación.

Cargo espera que tus archivos de código fuente residan en el directorio `src`. Esto deja el nivel raíz para otras cosas, como READMEs, información de licencias, y todo aquello no relacionado con tu código. Cargo nos ayuda a mantener nuestros proyectos agradables y ordenados. Un lugar para todo, y todo en su lugar.

A continuación, nuestro archivo de configuración:

```
$ editor Cargo.toml
```

Asegurate te tener este nombre correcto: necesitas la `C` mayuscula!

Coloca esto dentro:

```
[package]

name = "hola_mundo"
version = "0.0.1"
authors = [ "Tu nombre <tu@ejemplo.com>" ]
```

Este archivo esta en formato TOML. Dejemos que sea el mismo quien se explique:

> El objetivo de TOML es ser un formato de configuración minimo facil de leer debido a una semántica obvia. TOML esta diseñado para mapear de forma in-ambigua a una tabla hash. TOML debería ser fácil de convertir en estructuras de datos en una amplia variedad de lenguajes.

TOML es muy similar a INI, pero con algunas bondades extra.

Una vez que tengas este archivo es su lugar, deberíamos estar listos para compilar! Prueba esto:

```
$ cargo build
   Compiling hola_mundo v0.0.1 (file:///home/tunombre/proyectos/hola_mundo)
$ ./target/debug/hola_mundo
Hola, mundo!
```

Bam! Construimos nuestro proyecto con `cargo build`, y lo ejecutamos con `./target/debug/hola_mundo`. Podemos hacer los dos pasos en uno con `cargo run`:

```
$ cargo run
     Running `target/debug/hola_mundo`
Hola, mundo!
```

Nótese que no reconstruimos el proyecto esta vez. Cargo determino que no habíamos cambiado el archivo de código fuente, así que simplemente ejecuto el binario. Si hubiéremos realizado una modificación, deberíamos haberlo visto haciendo los dos pasos:

```
$ cargo run
   Compiling hola_mundo v0.0.1 (file:///home/tunombre/proyectos/hola_mundo)
     Running `target/debug/hola_mundo`
Hola, mundo!
```

Esto no nos ha aportado mucho por encima de nuestro simple uso de `rustc`, pero piensa en el futuro: cuando nuestros proyectos se hagan mas complicados, necesitaremos hacer mas cosas para lograr que todas las partes compilen correctamente. Con Cargo, a medida que nuestro proyecto crece, simplemente ejecutamos `cargo build`, y todo funcionara de forma correcta.

Cuando nuestro proyecto esta finalmente listo para la liberacion, puedes usar `cargo build --release` para compilarlo con optimizaciones.

También habras notado que Cargo ha creado un archivo nuevo: `Cargo.lock`.

```
[root]
name = "hola_mundo"
version = "0.0.1"
```

Este archivo es usado por Cargo para llevar el control de las dependencias usadas en tu aplicación. Por ahora, no tenemos ninguna, y esta un poco disperso. Nunca deberías necesitar tocar este archivo por tu cuenta, solo deja a Cargo manejarlo.

Eso es todo! Hemos construido satisfactoriamente `hola_mundo` con Cargo. Aunque nuestro programa sea simple, esta usando gran parte de las herramientas reales que usaras por el resto de tu carrera con Rust. Puedes asumir que para comenzar virtualmente con todo proyecto Rust harás lo siguiente:

```
$ git clone algunaurl.com/foo
$ cd foo
$ cargo build
```

Un Proyecto Nuevo

No tienes que pasar por todo ese proceso completo cada vez que quieras comenzar un proyecto nuevo! Cargo posee la habilidad de crear un directorio plantilla en el cual puedes comenzar a desarrollar inmediatamente.

Para comenzar un proyecto nuevo con Cargo, usamos `cargo new`:

```
$ cargo new hola_mundo --bin
```

Estamos pasando `--bin` porque estamos creando un programa binario: si estuviéramos creando una biblioteca, lo omitiríamos.

Echemos un vistazo a lo que Cargo ha generado para nosotros:

```
$ cd hola_mundo
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Si no tienes el comando `tree` instalado, probablemente podrías obtenerlo mediante el manejador de paquetes de tu distribución. No es necesario, pero es ciertamente útil.

Esto es todo lo que necesitas para comenzar. Primero veamos nuestro `Cargo.toml`:

```
[package]

name = "hola_mundo"
version = "0.0.1"
authors = ["Tu Nombre <tu@ejemplo.com>"]
```

Cargo a rellenado este archivo con valores por defecto basado en los argumentos que le proporcionaste y tu configuración global `git`. Podrias notar tambien que Cargo ha inicializado el directorio `hola_mundo` como un repositorio `git`.

Aqui esta el contenido de `src/main.rs`:

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

Cargo ha generado un "Hola, mundo!" para nosotros, ya estas listo para empezar a codear. Cargo posee su propia guia la cual cubre todas sus características con mucha mas profundidad.

Ahora que hemos aprendido las herramientas, comencemos en realidad a aprender mas de Rust como lenguaje. Esto es la báse que te servirá bien por el resto de tu tiempo con Rust.

Tienes dos opciones: Sumergirte en un proyecto con 'Aprende Rust', o comenzar desde el fondo y trabajar hacia arriba con 'Sintaxis y Semántica'. Programadores de sistemas mas experimentados probablemente preferiran 'Aprende Rust', mientras que aquellos provenientes de lenguajes dinamicos podrian tambien disfrutarlo. Gente diferente aprende diferente! Escoje lo que funcione mejor para ti.

% Aprende Rust

¡Bienvenido! Esta sección tiene unos pocos tutoriales que te enseñarán Rust a través de la construcción de proyectos. Obtendrás una visión general, pero también le echaremos un vistazo a los detalles.

Si prefieres una experiencia más al estilo ‘de abajo hacia arriba’, echa un vistazo a Sintaxis y Semántica.

% Juego de Adivinanzas

Para nuestro primer proyecto, implementaremos un problema clásico de programación para principiantes: un juego de adivinanzas. Como funciona el juego: Nuestro programa generara un numero entero aleatorio entre uno y cien. Nos pedira que introduzcamos una corazonada. Despues de haber proporcionado nuestro numero, este nos dirá si estuvimos muy por debajo y muy por encima. Una vez que adivinemos el numero correcto, nos felicitara. Suena bien?

Configuración Inicial

Creemos un nuevo proyecto. Anda a tu directorio de proyectos. Recuerdas como creamos nuestra estructura de directorios y un `Cargo.toml` para `hola_mundo` ? Cargo posse un comando que hace eso por nosotros. Probemoslo:

```
$ cd ~/proyectos
$ cargo new adivinanzas --bin
$ cd adivinanzas
```

Pasamos el nombre de nuestro proyecto a `cargo new` , junto con el flag `--bin` , debido a que estamos creando un binario, en vez de una biblioteca.

Echa un vistazo al `Cargo.toml` generado:

```
[package]

name = "adivinanzas"
version = "0.1.0"
authors = ["Tu Nombre <tu@ejemplo.com>"]
```

Cargo obtiene esta información de tu entorno. Si no esta correcta, corrigela.

Finalmente, Cargo ha generado un ‘Hola, mundo!’ para nosotros. Echa un vistazo a `src/main.rs` :

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

Tratemos de compilar lo que Cargo nos ha proporcionado:

```
$ cargo build
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
```

Excelente! Abre tu `src/main.rs` otra vez. Estaremos escribiendo todo nuestro codigo en este archivo.

Antes de continuar, dejame mostrarte un comando mas de Cargo: `run`. `cargo run` es una especie de `cargo build`, pero con la diferencia de que tambien ejecuta el binario producido. Probemoslo:

```
$ cargo run
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
     Running `target/debug/adivinanzas`
Hola, mundo!
```

Grandioso! El comando `run` es muy útil cuando necesitas iterar rapido en un proyecto. Nuestro juego es uno de esos proyectos, necesitaremos probar rapidamente cada iteración antes de movernos a la siguiente.

Procesando un Intento de Adivinanza

Probemoslo! La primera cosa que necesitamos hacer para nuestro juego de adivinanzas es permitir a nuestro jugador ingresar un intento de adivinanza. Coloca esto en tu `src/main.rs`:

```
use std::io;

fn main() {
    println!("Adivina el numero!");

    println!("Por favor introduce tu corazonada.");

    let mut corazonada = String::new();

    io::stdin().read_line(&mut corazonada)
        .ok()
        .expect("Fallo al leer linea");

    println!("Tu corazonada fue: {}", corazonada);
}
```

Hay un monton aqui! Tratemos de ir a traves de ello, pieza por pieza.

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

Necesitaremos recibir entrada del usuario, para luego imprimir el resultado como salida. Debido a esto necesitamos la biblioteca `io` de la biblioteca estandar. Rust solo importa unas pocas cosas para todos los programas, este conjunto de cosas se denomina 'preludio'. Si no esta en el preludio tendrás que llamarlo directamente a traves de `use` .

```
fn main() {
```

Como has visto con anterioridad, la función `main()` es el punto de entrada a tu programa. La sintaxis `fn` declara una nueva función, los `()` s indican que no hay argumentos y `{` comienza el cuerpo de la función. Debido a que no incluimos un tipo de retorno, se asume ser `()` una tupla vacía.

```
    println!("Adivina el numero!");

    println!("Por favor introduce tu corazonada.");
```

Anteriormente aprendimos que `println!()` es una macro que imprime una cadena de caracteres a la pantalla.

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

Ahora nos estamos poniendo interesantes! Hay un montón de cosas pasando en esta pequeña linea. La primera cosas a notar es que es una sentencia let, usada para crear variables. Tiene la forma:

```
let foo = bar;
```

Esto creara una nueva variable llamada `foo` , y la enlazara al valor `bar` . En muchos lenguajes, esto es llamado una 'variable' pero las variables de Rust tienen un par de trucos bajo la manga.

Por ejemplo, son immutables por defecto. Es por ello que nuestro ejemplo usa `mut` : esto hace un binding mutable, en vez de inmutable. `let` no solo toma un nombre del lado izquierdo, `let` acepta un 'patrón'. Usaremos los patrones un poco mas tarde. Es suficiente por ahora usar:

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

Ah, `//` inicia un comentario, hasta el final de la linea. Rust ignora todo en comentarios

Entonces sabemos que `let mut corazonada` introducirá un binding mutable llamado `corazonada`, pero tenemos que ver al otro lado del `=` para saber a que esta siendo asociado: `String::new()`.

`String` es un tipo de cadena de caracter, proporcionado por la biblioteca estandar. Un `String` es un segmento de texto codificado en UTF-8 capaz de crecer.

La sintaxis `::new()` usa `::` porque es una 'función asociada' de un tipo particular. Es decir esta asociada con `String` en si mismo, en vez de con una instacia en particular de `String`. Algunos lenguajes llaman a esto un 'metodo estatico'.

Esta funcion es llamada `new()`, porque crea un nuevo `String` vacio. Encontraras una función `new()` en muchos tipos, debido a que es un nombre común para la creación de un nuevo valor de algun tipo.

Continuemos:

```
    io::stdin().read_line(&mut corazonada)
        .ok()
        .expect("Fallo lectura de linea");
```

Otro monton! Vallamos pieza por pieza. La primera linea tiene dos partes. He aqui la primera:

```
io::stdin()
```

Recuerdas como usamos `use` en `std::io` en la primera linea de nuestro programa? Ahora estamos llamando una función asociada en `std::io`. De no haber usado `use std::io`, pudimos haber escrito esta linea como `std::io::stdin()`.

Esta función en particular retorna un handle a la entrada estándar de tu terminal. Mas especificamente, un std::io::Stdin.

La siguiente parte usara dicho handle para obtener entrada del usuario:

```
.read_line(&mut corazonada)
```

Aqui, llamamos el metodo `read_line()` en nuestro handle. Los metodos son similares a las funciones asociadas, pero solo estan disponibles en una instancia en particular de un tipo, en vez de en el tipo en si. También estamos pasando un argumento a `read_line()`: `&mut corazonada`.

Recuerdas cuando creamos `corazonada` ? Dijimos que era mutable. Sin embargo `read_line` no acepta un `String` como argumento: acepta un `&mut String` . Rust posee una caracteristica llamada 'referencias' ('references'), la cual permite tener multiples referencias a una pieza de data, de esta manera se reduce la necesidad de copiado. Las referencias son una caracteristica compleja, debido a que uno de los puntos de venta mas fuertes de Rust es acerca de cuan fácil y seguro es usar referencias. Por ahora no necesitamos saber mucho de esos detalles para finalizar nuestro programa. Todo lo que necesitamos saber por el momento es que al igual que los bindings `let` las referencias son inmutables por defecto. Como consecuencia necesitamos escribir `&mut corazonada` en vez de `&corazonada` .

Porque `read_line()` acepta una referencia mutable a una cadena de caracteres. Su trabajo es tomar lo que el usuario ingresa en la entrada estandar, y colocarlo en una cadena de caracteres. Debido a ello toma dicha cadena como argumento, y debido a que debe de agregar la entrada del usuario, este necesita ser mutable.

Todavia no hemos terminado con esta linea. Si bien es una sola linea de texto, es solo la primera parte de una linea logica de código completa:

```
        .ok()
        .expect("Fallo lectura de linea");
```

Cuando llamas a un metodo con la sintaxis `.foo()` puedes introducir un salto de linea y otro espacio. Esto te ayuda a dividir lineas largas. Pudimos haber escrito:

```
    io::stdin().read_line(&mut corazonada).ok().expect("Fallo lectura de linea");
```

Pero eso es mas difícil de leer. Así que lo hemos dividido en tres lineas para tres llamadas a metodo. Ya hemos hablado de `read_line()` , pero que acerca de `ok()` y `expect()` ? Bueno, ya mencionamos que `read_line()` coloca la entrada del usuario en el `&mut String` que le proprocionamos. Pero tambien retorna un valor: en este caso un `io::Result` . Rust posee un numero de tipos llamados `Result` en su biblioteca estandar: un `Result` generico, y versiones especificas para sub-bibliotecas, como `io::Result` .

El proposito de esos `Result` es codificar información de manejo de errores. Valores del tipo `Result` tienen metodos definidos en ellos. En este caso `io::Result` posee un metodo `ok()` , que se traduce en 'queremos asumir que este valor es un valor exitoso. Sino, descarta la información acerca del error'. Porque descartar la información acerca del error?, para un programa básico, queremos simplemente imprimir un error generico, cualquier problema que signifique que no podamos continuar. El metodo `ok()` retorna un valor que tiene otro metodo definito en el: `expect()` . El `metodo expect()` toma el valor en el cual es

llamado y si no es un valor exitoso, hace panico `panic!` con un mensaje que le hemos proporcionado. Un `panic!` como este causara que el programa tenga una salida abrupta (crash), mostrando dicho mensaje.

Si quitamos las llamadas a esos dos metodos, nuestro programa compilara, pero obtendremos una advertencia:

```
$ cargo build
  Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
src/main.rs:10:5: 10:44 warning: unused result which must be used, #[warn(unused_must_use
src/main.rs:10     io::stdin().read_line(&mut corazonada);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust nos advierte que no hemos usado el valor `Result` . Esta advertencia viene de una anotación especial que tiene `io::Result` . Rust esta tratando de decirte que no has manejado un posible error. La manera correcta de suprimir el error es, en efecto escribir el código para el manejo de erroes. Por suerte, si solo queremos terminar la ejecución del programa de haber un problema, podemos usar estos dos pequeños metodos. Si pudieramos recuperarnos del error de alguna manera, hariamos algo diferente, pero dejemos eso para un proyecto futuro.

Solo nos queda una linea de este primer ejemplo:

```
    println!("Tu corazonada fue: {}", corazonada);
}
```

Esta linea imprime la cadena de caracteres en la que guardamos nuestra entrada. Los `{}` s son marcadores de posición, es por ello que pasamos `adivinanza` como argumento. De haber habido multiples `{}` s, debiamos haber pasado multiples argumentos:

```
let x = 5;
let y = 10;

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

Fácil.

De cualquier modo, ese es el tour. Podemos ejecutarlo con `cargo run` :

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/tu/proyectos/adivinanzas)
     Running `target/debug/adivinanzas`
Adivina el numero!
Por favor introduce tu corazonada.
6
Tu corazonada fue: 6
```

En hora buena! Nuestra primera parte ha terminado: podemos obtener entrada del teclado e imprimirla de vuelta.

Generando un numero secreto

A continuación necesitamos generar un numero secreto. Rust todavía no incluye una funcionalidad de numeros aleatorios en la biblioteca estándar. Sin embargo, el equipo de Rust provee un crate `rand`. Un 'crate' es un paquete de código Rust. Nosotros hemos estado construyendo un 'crate binaro', el cual es un ejecutable. `rand` es un 'crate biblioteca', que contiene codigo a ser usado por otros programas.

Usar crates externos es donde Cargo realmente brilla. Antes que podamos escribir código que haga uso de `rand`, debemos modificar nuestro archivo `Cargo.toml`. Abrelo, y agrega estas lineas al final:

```
[dependencies]

rand="0.3.0"
```

La sección `[dependencies]` de `Cargo.toml` es como la sección `[package]`: todo lo que le sigue es parte de ella, hasta que la siguiente sección comience. Cargo usa la sección de dependencias para saber en cuales crates externos dependemos, asi como las versiones requeridas. En este caso hemos usado la versión `0.3.0`. Cargo entiende Versionado Semantico, que es un estandar para las escritura de numeros de versión. Si quisieramos usar la ultima version podriamos haber usado `*` o un rango de versiones. La documentación de Cargo contiene mas detalles.

Ahora, sin cambiar nada en nuestro código, construyamos nuestro proyecto:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.8
 Downloading libc v0.1.6
   Compiling libc v0.1.6
   Compiling rand v0.3.8
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
```

(Podrias ver versiones diferentes, por supuesto.)

Un montón de salida mas! Ahora que tenemos una dependencia externa, Cargo descarga del registro las ultimas versiones de todo, lo cual puede copiar datos desde Crates.io. Crates.io es donde las personas del ecosistema Rust publican sus proyectos open source para que otros los usen.

Despues de actualizar el registro, Cargo chequea nuestras dependencias (en `[dependencies]`) y las descarga de no tenerlas todavía. En este caso solo dijimos que queriamos depender en `rand`, y tambien obtuvimos una copia de `libc`. Esto es debido a que `rand` depende a su vez de `libc` para funcionar. Despues de descargar las dependencias, Cargo las compila, para despues compilar nuestro código.

Si ejecutamos `cargo build`, obtendremos una salida diferente:

```
$ cargo build
```

Asi es, no hay salida! Cargo sabe que nuestro proyecto ha sido construido, asi como todas sus dependencias, asi que no nay razón para hacer todo el proceso otra vez. Sin nada que hacer, simplemente termina la ejecucion. Si abrimos `src/main.rs` otra vez, hacemos un cambio trivial, salvamos los cambios, solamente veriamos una linea:

```
$ cargo build
    Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
```

Entonces, le hemos dicho a Cargo que queriamos cualquier versión `0.3.x` de `rand`, y este descargo la ultima versión para el momento de la escritura de este tutorial, `v0.3.8`. Pero que pasa cuando la siguiente versión `v0.3.9` sea publicada con un importante bugfix? Si bien recibir bugfixes es importante, que pasa si `0.3.9` contiene una regresion que rompe nuestro codigo?

La respuesta a este problema es el archivo `Cargo.lock`, archivo que encontraras en tu directorio de proyecto. Cuando construyes tu proyecto por primera vez, cargo determina todas las versiones que coinciden con tus criterios y las escribe en el archivo `Cargo.lock`. Cuando construyes tu proyecto en el futuro, Cargo notara que un archivo `Cargo.lock`

existe, y usara las versiones especificadas en el mismo, en vez de hacer todo el trabajo de determinar las versiones otra vez. Esto te permite tener una construcción reproducible de manera automatica. En otras palabras, nos quedaremos en `0.3.8` hasta que subamos de version de manera explicita, de igual manera lo hará la gente con la que hemos compartido nuestro código, gracias al archivo `Cargo.lock`.

Pero que pasa cuando *queremos* usar `v0.3.9`? Cargo posee otro comando, `update`, que se traduce en 'ignora el bloqueo y determina todas las ultimas versiones que coincidan con lo que hemos especficado. De funcionar esto, escribe esas versiones al archivo de bloqueo `Cargo.lock`'. Pero, por defecto, Cargo solo buscara versiones mayores a `0.3.0` y menores a `0.4.0`. Si queremos movernos a `0.4.x`, necesitariamos actualizar el archivo `Cargo.toml` directamente. Cuando lo hacemos, la siguente vez que ejecutemos `cargo build`, Cargo actualizara el indice y re-evaluara nuestros requerimentos acerca de `rand`.

Hay mucho mas que decir acerca de Cargo y su ecosistema, pero por ahora, eso es todo lo que necesitamos saber. Cargo hace realmente fácil reusar bibliotecas, y los Rusteros tienden a escribir proyectos pequenos los cuales estan construidos por un conjunto de paquetes mas pequeños.

Hagamos *uso* ahora de `rand`. He aqui nuestro siguiente paso:

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    println!("El numero secreto es: {}", numero_secreto);

    println!("Por favor introduce tu corazonada.");

    let mut corazonada = String::new();

    io::stdin().read_line(&mut corazonada)
        .ok()
        .expect("Fallo al leer linea");

    println!("Tu corazonada fue: {}", corazonada);
}
```

La primera cosa que hemos hecho es cambiar la primera linea. Ahora dice `extern crate rand`. Debido a que declaramos `rand` en nuestra sección `[dependencies]`, podemos usar `extern crate` para hacerle saber a Rust que estaremos haciendo uso de `rand`. Esto es equivalente a un `use rand;`, de manera que podamos hacer uso de lo que sea dentro del crate `rand` a traves del prefijo `rand::`.

Después, hemos agregado otra linea `use`: `use rand::Rng`. En unos momentos estaremos haciendo uso de un metodo, y esto requiere que `Rng` este disponible para que funcione. La idea basica es la siguiente: los metodos estan dentro de algo llamado 'traits' (Rasgos), y para que el metodo funcione necesita que el trait este disponible. Para mayores detalles dirigete a la sección Rasgos (Traits).

Hay dos lineas mas en el medio:

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

    println!("El numero secreto es: {}", numero_secreto);
```

Hacemos uso de la función `rand::thread_rng()` para obtener una copia del generador de numeros aleatorios, el cual es local al hilo de ejecucion en el cual estamos. Debido a que hemos hecho disponible a `rand::Rng` a traves de `use rand::Rng`, este tiene un metodo `gen_range()` disponible. Este metodo acepta dos argumentos, y genera un numero aleatorio entre estos. Es inclusivo en el limite inferior, pero es exclusivo en el limite superior, por eso necesitamos `1` y `101` para obtener un numero entre uno y cien.

La segunda linea solo imprime el numero secreto. Esto es útil mietras desarrollamos nuestro programa, de manera tal que podamos probarlo. Estaremos eliminando esta linea para la version final. No es un juego si imprime la respuesta justo cuando lo inicias!

Trata de ejecutar el programa unas pocas veces:

```
$ cargo run
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
     Running `target/debug/adivinanzas`
Adivina el numero!
El numero secreto es: 7
Por favor introduce tu corazonada.
4
Tu corazonada fue: 4
$ cargo run
     Running `target/debug/adivinanzas`
Adivina el numero!
El numero secreto es: 83
Por favor introduce tu corazonada.
5
Tu corazonada fue: 5
```

Gradioso! A continuacion: comparemos nuestra adivinanza con el numero secreto.

Comparando adivinanzas

Ahora que tenemos entrada del usuario, comparemos la adivinanza con nuestro numero secreto. He aqui nuestro siguiente paso, aunque todavia no funciona completamente:

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    println!("El numero secreto es: {}", numero_secreto);

    println!("Por favor introduce tu corazonada.");

    let mut corazonada = String::new();

    io::stdin().read_line(&mut corazonada)
        .ok()
        .expect("Fallo al leer linea");

    println!("Tu corazonada fue: {}", corazonada);

    match corazonada.cmp(&numero_secreto) {
        Ordering::Less    => println!("Muy pequeño!"),
        Ordering::Greater => println!("Muy grande!"),
        Ordering::Equal   => println!("Haz ganado!"),
    }
}
```

Algunas piezas acá. La primera es otro `use`. Hemos hecho disponible un tipo llamado `std::cmp::Ordering`. Despues, cinco nuevas lineas al fondo que lo usan:

```
match corazonada.cmp(&numero_secreto) {
    Ordering::Less    => println!("Muy pequeño!"),
    Ordering::Greater => println!("Muy grande!"),
    Ordering::Equal   => println!("Haz ganado!"),
}
```

El metodo `cmp()` puede ser llamado an cualquier cosa que pueda ser comparada, este toma una referencia a la cosa con la cual quieras comparar. Retorna el tipo `Ordering` que hicimos disponible anteriormente. Hemos usado una sentencia `match` para determinar exactamente que tipo de `Ordering` es. `Ordering` es un `enum`, abreviacion para ‘enumeration’, las cuales lucen de la siguiente manera:

```
enum Foo {
    Bar,
    Baz,
}
```

Con esta definición, cualquier cosa de tipo `Foo` puede ser bien sea un `Foo::Bar` o un `Foo::Baz`. Usamos el `::` para indicar el espacio de nombres para una variante `enum` en particular.

La enum `Ordering` tiene tres posibles variantes: `Less`, `Equal`, and `Greater` (menor, igual y mayor respectivamente). La sentencia `match` toma un valor de un tipo, y te permite crear un 'brazo' para cada valor posible. Debido a que tenemos tres posibles tipos de `Ordering`, tenemos tres brazos:

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Muy pequeño!"),
    Ordering::Greater => println!("Muy grande!"),
    Ordering::Equal   => println!("Haz ganado!"),
}
```

Si es `Less`, imprimimos `Too small!`, si es `Greater`, `Too big!`, y si es `Equal`, `Haz ganado!`. `match` es realmente util, y es usado con fercuencia en Rust.

Anteriormente mencione que todavia no funciona. Pongamoslo a prueba:

```
$ cargo build
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected `&collections::string::String`,
    found `&_`
(expected struct `collections::string::String`,
    found integral variable) [E0308]
src/main.rs:28     match corazonada.cmp(&numero_secreto) {
                                    ^~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `adivinanzas`.
```

Oops! Un gran error. Lo principal en el es que tenemos 'tipos incompatibles' ('mismatched types'). Rust posee un fuerte, sistema de tipos estatico. Sin embargo, también tiene inferencia de tipos. Cuando escribimos `let corazonada = String::new()`, Rust fue capaz de inferir que `corazonada` debia ser un `String`, y por ello no nos hizo escribir el tipo. Con nuestro `numero_secreto`, hay un numero de tipos que pueden tener un valor entre uno y cien: `i32`, un numero de treinta y dos bits, `u32`, un numero sin signo de treinta y dos bits, o `i64` un numero de sesenta y cuatro bits u otros. Hasta ahora, eso no ha importado,

debido a que Rust por defecto usa `i32`. Sin embargo, en este caso, Rust no sabe como comparar `corazonada` con `numero_secreto`. Ambos necesitan ser del mismo tipo. A la final, queremos convertir el `String` que leimos como entrada en un tipo real de numero, para efectos de la comparación. Podemos hacer eso con tres lineas mas. He aqui nuestro nuevo programa:

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    println!("El numero secreto es: {}", numero_secreto);

    println!("Por favor introduce tu corazonada.");

    let mut corazonada = String::new();

    io::stdin().read_line(&mut corazonada)
        .ok()
        .expect("Fallo al leer linea");

    let corazonada: u32 = corazonada.trim().parse()
        .ok()
        .expect("Por favor introduce un numero!");

    println!("Tu corazonada fue: {}", corazonada);

    match corazonada.cmp(&numero_secreto) {
        Ordering::Less    => println!("Muy pequeño!"),
        Ordering::Greater => println!("Muy grande!"),
        Ordering::Equal   => println!("Haz ganado!"),
    }
}
```

Las tres nuevas lineas:

```
    let corazonada: u32 = corazonada.trim().parse()
        .ok()
        .expect("Por favor introduce un numero!");
```

Espera un momento, pensé que ya teniamos una `corazonada` ? La tenemos, pero Rust nos permite sobreescribir ('shadow') la `corazonada` previa con una nueva. Esto es usado con frecuencia en esta misma situación, en donde `corazonada` es un `String` , pero queremos convertirlo a un `u32` . Este shadowing nos permite reusar el nombre `corazonada` en vez de forzarnos a idear dos nombres únicos como `corazonada_str` y `corazonada` , u otros.

Estamos asociando `corazonada` a una expresión que luce como algo que escribimos anteriormente:

```
guess.trim().parse()
```

Seguido por una invocación a `ok().expect()` . Aquí `corazonada` hace referencia a la vieja versión, la que era un `String` que contenía nuestra entrada de usuario en ella. El metodo `trim()` en los `String` s elimina cualquier espacio en blanco al principio y al final de nuestras cadenas de caracteres. Esto es importante, debido a que tuvimos que presionar la tecla 'retorno' para satisfacer a `read_line()` . Esto significa que si escribimos `5` y presionamos 'retorno' `corazonada` luce como así: `5\n` . El `\n` representa 'nueva linea' ('newline'), la tecla enter. `trim()` se deshace de esto, dejando nuestra cadena de caracteres solo con el `5` . El metodo `parse()` en las cadenas caracteres parsea una cadena de caracteres en algún tipo de numero. Debido a que puede parsear una variedad de numeros, debemos darle a Rust una pista del tipo exacto de numero que deseamos. De ahí la parte `let corazonada: u32` . Los dos puntos (`:`) despues de `corazonada` le dicen a Rust que vamos a anotar el tipo. `u32` es un entero sin signo de treinta y dos bits. Rust posee una variedad de tipos numero integrados, pero nosotros hemos escojido `u32` . Es una buena opción por defecto para un numero positivo pequeño.

Al igual que `read_line()` , nuestra llamada a `parse()` podria causar un error. Que tal si nuestra cadena de caracteres contiene `A␣👍%` ? No habría forma de convertir eso en un numero. Es por ello que haremos lo mismo que hicimos con `read_line()` : usar los metodos `ok()` y `expect()` para terminar abruptamente si hay algun error.

Probemos nuestro programa!

```
$ cargo run
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
     Running `target/adivinanzas`
Adivina el numero!
El numero secreto es: 58
Por favor introduce tu corazonada.
  76
Tu corazonada fue: 76
Muy grande!
```

Excelente! Puedes ver que incluso he agregado espacios antes de mi intento, y aun así el programa determino que intente 76. Ejecuta el programa unas pocas veces, y verifica que adivinar el numero funciona, asi como intentar un numero muy pequeno.

Ahora tenemos la mayoria del juego funcionando, pero solo podemos intentar adivinar una vez. Tratemos de cambiar eso agregando ciclos!

Iteración

La palabra clave `loop` nos proporciona un ciclo infinito. Agreguemosla:

Adivina el numero! El numero secreto es: 58 Por favor introduce tu adivinanza. 76 Tu corazonada fue: 76 Muy grande!

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    println!("El numero secreto es: {}", numero_secreto);

    loop {
        println!("Por favor introduce tu corazonada.");

        let mut corazonada = String::new();

        io::stdin().read_line(&mut corazonada)
            .ok()
            .expect("Fallo al leer linea");

        let corazonada: u32 = corazonada.trim().parse()
            .ok()
            .expect("Por favor introduce un numero!");

        println!("Haz corazonada: {}", corazonada);

        match corazonada.cmp(&numero_secreto) {
            Ordering::Less    => println!("Muy pequeño!"),
            Ordering::Greater => println!("Muy grande!"),
            Ordering::Equal   => println!("Haz ganado!"),
        }
    }
}
```

Pruebalo. Pero espera, no acabamos de agregar un ciclo infinito? Sip. Recuerdas nuestra discusión acerca de `parse()` ? Si damos una respuesta no numérica, retornaremos (`return`) y finalizaremos la ejecución. Observa:

```
$ cargo run
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
     Running `target/adivinanzas`
Adivina el numero!
El numero secreto es: 59
Por favor introduce tu corazonada.
45
Tu corazonada fue: 45
Muy pequeño!
Por favor introduce tu corazonada.
60
Tu corazonada fue: 60
Muy grande!
Por favor introduce tu corazonada.
59
Tu corazonada fue: 59
Haz ganado!
Por favor introduce tu corazonada.
quit
thread '<main>' panicked at 'Please type a number!'
```

Ja! `quit` en efecto termina la ejecución. Asi como cualquier otra entrada que no sea un numero. Bueno, esto es suboptimo por decir lo menos. Primero salgamos cuando ganemos:

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    println!("El numero secreto es: {}", numero_secreto);

    loop {
        println!("Por favor introduce tu corazonada.");

        let mut corazonada = String::new();

        io::stdin().read_line(&mut corazonada)
            .ok()
            .expect("Fallo al leer linea");

        let corazonada: u32 = corazonada.trim().parse()
            .ok()
            .expect("Por favor introduce un numero!");

        println!("Tu corazonada fue: {}", corazonada);

        match corazonada.cmp(&numero_secreto) {
            Ordering::Less    => println!("Muy pequeño!"),
            Ordering::Greater => println!("Muy grande!"),
            Ordering::Equal   => {
                println!("Haz ganado!");
                break;
            }
        }
    }
}
```

Al agregar la linea `break` despues del "Haz ganado!", romperemos el ciclo cuando ganemos. Salir del ciclo también significa salir del programa, debido a que es la ultima cosa en `main()`. Solo nos queda una sola mejora por hacer: cuando alguien introduzca un valor no numérico, no queremos terminar la ejecución, queremos simplemente ignorarlo. Podemos hacerlo de la siguiente manera:

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    println!("El numero secreto es: {}", numero_secreto);

    loop {
        println!("Por favor introduce tu corazonada.");

        let mut corazonada = String::new();

        io::stdin().read_line(&mut corazonada)
            .ok()
            .expect("Fallo al leer linea");

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

        println!("Tu corazonada fue: {}", corazonada);

        match corazonada.cmp(&numero_secreto) {
            Ordering::Less    => println!("Muy pequeño!"),
            Ordering::Greater => println!("Muy grande!"),
            Ordering::Equal   => {
                println!("Haz ganado!");
                break;
            }
        }
    }
}
```

Estas son las lineas que han cambiado:

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

Es asi como pasamos de 'terminar abruptamente en un error' a 'efectivamente manejar el error', a través del cambio de `ok().expect()` a una sentencia `match`. El `Result` retornado por `parse()` es un enum justo como `Ordering`, pero en este caso cada variante tiene data asociada: `Ok` es exito, y `Err` es una falla. Cada uno contiene mas información: el entero parseado en el caso exitoso, o un tipo de error. En este caso hacemos `match` en `Ok(num)`, el cual asigna el valor interno del `Ok` a el nombre `num`, y seguidamente retorna en el lado derecho. En el caso de `Err`, no nos importa que tipo de error es, es por ello que usamos `_` en lugar de un nombre. Esto ignora el error y `continue` nos mueve a la siguiente iteración del ciclo (`loop`).

Ahora deberiamos estar bien! Probemos:

```
$ cargo run
   Compiling adivinanzas v0.1.0 (file:///home/tu/proyectos/adivinanzas)
     Running `target/adivinanzas`
Adivina el numero!
El numero secreto es: 61
Por favor introduce tu corazonada.
10
Tu corazonada fue: 10
Muy pequeño!
Por favor introduce tu corazonada.
99
Tu corazonada fue: 99
Muy pequeño!
Por favor introduce tu corazonada.
foo
Por favor introduce tu corazonada.
61
Tu corazonada fue: 61
Haz ganado!
```

Genial! Con una ultima mejora, finalizamos el juego de las advinanzas. Te imaginas cual es? Es correcto, no queremos imprimir el numero secreto. Era bueno para las pruebas, pero arruina nuestro juego. He aqui nuestro código fuente final:

```
extern crate rand;

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

fn main() {
    println!("Adivina el numero!");

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

    loop {
        println!("Por favor introduce tu corazonada.");

        let mut corazonada = String::new();

        io::stdin().read_line(&mut corazonada)
            .ok()
            .expect("Fallo al leer linea");

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

        println!("Tu corazonada fue: {}", corazonada);

        match corazonada.cmp(&numero_secreto) {
            Ordering::Less    => println!("Muy pequeño!"),
            Ordering::Greater => println!("Muy grande!"),
            Ordering::Equal   => {
                println!("Haz ganado!");
                break;
            }
        }
    }
}
```

Completado!

En este punto, has terminado satisfactoriamente el juego de las adivinanza! Felicitaciones!

Este primer proyecto te enseno un montón: `let`, `match`, metodos, funciones asociadas, usar crates externos, y mas. Nuestro siguiente proyecto demostrara aun mas.

% La Cena de los Filósofos

Para nuestro segundo proyecto, echemos un vistazo a un problema clásico de concurrencia. Se llama ‘La cena de los filósofos’. Fue originalmente concebido por Dijkstra en 1965, pero nosotros usaremos una version ligeramente adaptada de este paper por Tony Hoare en 1985.

> En tiempos ancestrales, un filántropo adinerado preparo una universidad para alojar a cinco filósofos eminentes. Cada filósofo tenia una habitación en la cual podía desempeñar su actividad profesional del pensamiento: también había un comedor en común, amoblado con una mesa circular, rodeada por cinco sillas, cada una identificada con el nombre del filosofo que se sentaba en ella. Los filósofos se sentaban en sentido anti-horario alrededor de la mesa. A la izquierda de cada filósofo yacía un tenedor dorado, y en el centro un tazón de espagueti, el cual era constantemente rellenado. Se esperaba que un filósofo empleara la mayoría de su tiempo pensando; pero cuando se sintieran con hambre, se dirigiera a el comedor tomara el tenedor que estaba a su izquierda y lo sumieran en el espagueti. Pero tal era naturaleza enredada del espagueti que un segundo tenedor era requerido para llevarlo a la boca. El filósofo por ende tenia que también tomar el tenedor a su derecha. Cuando terminaban debían bajar ambos tenedores, levantarse de la silla y continuar pensando. Por supuesto, un tenedor puede ser usado por un solo filósofo a la vez. Si otro filósofo lo desea, tiene que esperar hasta que el tenedor este disponible nuevamente.

Este problema clásico exhibe algunos elementos de la concurrencia. La razón de ello es que es una solución efectivamente difícil de implementar: una implementación simple puede generar un deadlock. Por ejemplo, consideremos un algoritmo simple que podría resolver este problema:

1. Un filósofo toma el tenedor a su izquierda.
2. Después toma el tenedor en a su derecha.
3. Come.
4. Baja los tenedores.

Ahora, imaginemos esta secuencia de eventos:

1. Filosofo 1 comienza el algoritmo, tomando el tenedor a su izquierda.
2. Filosofo 2 comienza el algoritmo, tomando el tenedor a su izquierda.
3. Filosofo 3 comienza el algoritmo, tomando el tenedor a su izquierda.
4. Filosofo 4 comienza el algoritmo, tomando el tenedor a su izquierda.
5. Filosofo 5 comienza el algoritmo, tomando el tenedor a su izquierda.
6. ... ? Todos los tenedores han sido tomados, pero nadie puede comer!

Existen diferentes formas de resolver este problema. Te guiaremos a través de la solución de este tutorial. Por ahora, comencemos modelando el problema. Empecemos con los filósofos:

```
struct Filosofo {
    nombre: String,
}

impl Filosofo {
    fn new(nombre: &str) -> Filosofo {
        Filosofo {
            nombre: nombre.to_string(),
        }
    }
}

fn main() {
    let f1 = Filosofo::new("Judith Butler");
    let f2 = Filosofo::new("Gilles Deleuze");
    let f3 = Filosofo::new("Karl Marx");
    let f4 = Filosofo::new("Emma Goldman");
    let f5 = Filosofo::new("Michel Foucault");
}
```

Acá, creamos una `estructura` (struct) para representar un filósofo. Por ahora el nombre es todo lo que necesitamos. Elegimos el tipo `String` para el nombre, en vez de `&str`. Generalmente hablando, trabajar con tipo que es dueño (posee pertenencia) de su data es mas fácil que trabajar con uno que use referencias.

Continuemos:

```
# struct Filosofo {
#     nombre: String,
# }
impl Filosofo {
    fn new(nombre: &str) -> Filosofo {
        Filosofo {
            nombre: nombre.to_string(),
        }
    }
}
```

Este bloque `impl` nos permite definir cosas en estructuras `Filosofo`. En este caso estamos definiendo una ‘función asociada’ llamada `new`. La primera linea luce así:

```
# struct Filosofo {
#     nombre: String,
# }
# impl Filosofo {
fn new(nombre: &str) -> Filosofo {
#         Filosofo {
#             nombre: nombre.to_string(),
#         }
#     }
# }
```

Recibimos un argumento, `nombre`, de tipo `&str`. Una referencia a otra cadena de caracteres. Esta retorna una instancia de nuestra estructura `Filosofo`.

```
# struct Filosofo {
#     nombre: String,
# }
# impl Filosofo {
#    fn new(nombre: &str) -> Filosofo {
Filosofo {
    nombre: nombre.to_string(),
}
#     }
# }
```

Lo anterior crea un nuevo `Filosofo`, y asigna nuestro argumento `nombre` a el campo `nombre`. No a el argumento en si mismo, debido a que llamamos `.to_string()` en el. Lo cual crea una copia de la cadena a la que apunta nuestro `&str`, y nos da un nuevo `String`, que es del tipo del campo `nombre` de `Filosofo`.

Porque no aceptar un `String` directamente? Es mas fácil de llamar. Si recibiéramos un `String` pero quien nos llama tuviese un `&str` ellos se verían en la obligación de llamar `.to_string()` de su lado. La desventaja de esta flexibilidad es que *siempre* hacemos una copia. Para este pequeño programa, esto no es particularmente importante, y que sabemos que estaremos usando cadenas cortas de cualquier modo.

Una ultima cosas que habrás notado: solo definimos un `Filosofo`, y no parecemos hacer nada con el. Rust es un lenguaje 'basado en expresiones', lo que significa que casi cualquier cosa en Rust es una expresión que retorna un valor. Esto es cierto para las funciones también, la ultima expresión es retornada automáticamente. Debido a que creamos un nuevo `Filosofo` como la ultima expresión de esta función, terminamos retornándolo.

El nombre `new()`, no es nada especial para Rust, pero es una convención para funciones que crean nuevas instancias de estructuras. Antes que hablemos del porque, echamos un vistazo a `main()` otra vez:

```
# struct Filosofo {
#     nombre: String,
# }
#
# impl Filosofo {
#     fn new(nombre: &str) -> Filosofo {
#         Filosofo {
#             nombre: nombre.to_string(),
#         }
#     }
# }
#
fn main() {
    let f1 = Filosofo::new("Judith Butler");
    let f2 = Filosofo::new("Gilles Deleuze");
    let f3 = Filosofo::new("Karl Marx");
    let f4 = Filosofo::new("Emma Goldman");
    let f5 = Filosofo::new("Michel Foucault");
}
```

Acá, creamos cinco variables con cinco nuevos filósofos. Estos son mis cinco favoritos, pero puedes substituirlos con quienes prefieras. De no haber definido la función `new()`, `main()` luciría así:

```
# struct Filosofo {
#     nombre: String,
# }
fn main() {
    let f1 = Filosofo { nombre: "Judith Butler".to_string() };
    let f2 = Filosofo { nombre: "Gilles Deleuze".to_string() };
    let f3 = Filosofo { nombre: "Karl Marx".to_string() };
    let f4 = Filosofo { nombre: "Emma Goldman".to_string() };
    let f5 = Filosofo { nombre: "Michel Foucault".to_string() };
}
```

Un poco mas ruidoso. Usar `new` tiene también posee otras ventajas, pero incluso en este simple caso termina por ser de mejor utilidad.

Ahora que tenemos lo básico en su lugar, hay un numero de maneras en las cuales podemos atacar el problema mas amplio. A mi me gusta comenzar por el final: creemos una forma para que cada filosofo pueda finalizar de comer. Como un paso pequeño, hagamos un método, y luego iteremos a través de todos los filósofos llamándolo:

```
struct Filosofo {
    nombre: String,
}

impl Filosofo {
    fn new(nombre: &str) -> Filosofo {
        Filosofo {
            nombre: nombre.to_string(),
        }
    }

    fn comer(&self) {
        println!("{} ha finalizado de comer.", self.nombre);
    }
}

fn main() {
    let filosofos = vec![
        Filosofo::new("Judith Butler"),
        Filosofo::new("Gilles Deleuze"),
        Filosofo::new("Karl Marx"),
        Filosofo::new("Emma Goldman"),
        Filosofo::new("Michel Foucault"),
    ];

    for f in &filosofos {
        f.comer();
    }
}
```

Primero veamos a `main()`. En lugar de tener cinco variables individuales para nuestros filósofos, creamos un `Vec<T>`. `Vec<T>` es llamado también un 'vector', y es un arreglo capaz de crecer. Después usamos un ciclo [`for`][for] para iterar a través del vector, obteniendo un referencia a cada filosofo a la vez.

En el cuerpo del bucle, llamamos `f.comer();`, que esta definido como:

```
fn comer(&self) {
    println!("{} ha finalizado de comer.", self.nombre);
}
```

En Rust, los métodos reciben un parámetro explícito `self`. Es por ello que `comer()` es un método y `new` es una función asociada: `new()` no tiene `self`. Para nuestra primera version de `comer()`, solo imprimimos el nombre del filósofo, y mencionamos que ha finalizado de comer. Ejecutar este programa deber generar la siguiente salida:

```
Judith Butler ha finalizado de comer.
Gilles Deleuze ha finalizado de comer.
Karl Marx ha finalizado de comer.
Emma Goldman ha finalizado de comer.
Michel Foucault ha finalizado de comer.
```

Muy fácil, todos han terminado de comer! Pero no hemos implementado el problema real todavía, así que aun no terminamos!

A continuación, no solo queremos solo finalicen de comer, sino que efectivamente coman. He aquí la siguiente versión:

```
use std::thread;

struct Filosofo {
    nombre: String,
}

impl Filosofo {
    fn new(nombre: &str) -> Filosofo {
        Filosofo {
            nombre: nombre.to_string(),
        }
    }

    fn comer(&self) {
        println!("{} esta comiendo.", self.nombre);

        thread::sleep_ms(1000);

        println!("{} ha finalizado de comer.", self.nombre);
    }
}

fn main() {
    let filosofos = vec![
        Filosofo::new("Judith Butler"),
        Filosofo::new("Gilles Deleuze"),
        Filosofo::new("Karl Marx"),
        Filosofo::new("Emma Goldman"),
        Filosofo::new("Michel Foucault"),
    ];

    for f in &filosofos {
        f.comer();
    }
}
```

Solo unos pocos cambios. Analicémoslos parte por parte.

```
use std::thread;
```

`use` hace disponibles nombres en nuestro ámbito (scope). Comenzaremos a usar el modulo `thread` de la biblioteca estándar, y es por ello que necesitamos hacer `use` en el.

```
    fn comer(&self) {
        println!("{} esta comiendo.", self.nombre);

        thread::sleep_ms(1000);

        println!("{} ha finalizado de comer.", self.nombre);
    }
```

Ahora estamos imprimiendo dos mensajes, con un `sleep_ms()` en el medio. Lo cual simulara el tiempo que tarda un filosofo en comer.

Si ejecutas este programa, deberias ver comer a cada filosofo a la vez:

```
Judith Butler esta comiendo.
Judith Butler ha finalizado de comer.
Gilles Deleuze esta comiendo.
Gilles Deleuze ha finalizado de comer.
Karl Marx esta comiendo.
Karl Marx ha finalizado de comer.
Emma Goldman esta comiendo.
Emma Goldman ha finalizado de comer.
Michel Foucault esta comiendo.
Michel Foucault ha finalizado de comer.
```

Excelente! Estamos avanzando. Solo hay un detalle: no estamos operando de manera concurrente, lo cual es parte central de nuestro problema!

Para hacer a nuestros filósofos comer de manera concurrente, necesitamos hacer un pequeño cambio.

He aqui la siguiente iteración:

```
use std::thread;

struct Filosofo {
    nombre: String,
}

impl Filosofo {
    fn new(nombre: &str) -> Filosofo {
        Filosofo {
            nombre: nombre.to_string(),
        }
    }

    fn comer(&self) {
        println!("{} esta comiendo.", self.nombre);

        thread::sleep_ms(1000);

        println!("{} ha finalizado de comer.", self.nombre);
    }
}

fn main() {
    let filosofos = vec![
        Filosofo::new("Judith Butler"),
        Filosofo::new("Gilles Deleuze"),
        Filosofo::new("Karl Marx"),
        Filosofo::new("Emma Goldman"),
        Filosofo::new("Michel Foucault"),
    ];

    let handles: Vec<_> = filosofos.into_iter().map(|f| {
        thread::spawn(move || {
            f.comer();
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

Todo lo que hemos hecho es cambiar el ciclo en `main()`, y agregado un segundo! Este es el primer cambio:

```
let handles: Vec<_> = filosofos.into_iter().map(|f| {
    thread::spawn(move || {
         f.comer();
    })
}).collect();
```

Aun así son solo cinco lineas, son cinco densas lineas. Analicemos por partes.

```
let handles: Vec<_> =
```

Introducimos una nueva variable, llamada `handles`. Le hemos dado este nombre porque crearemos algunos nuevos hilos, que resultaran en algunos handles (agarradores, manillas) a esos dichos hilos los cuales nos permitirán controlar su operación. Necesitamos anotar el tipo explícitamente, debido a algo que haremos referencia mas adelante. El `_` es un marcador de posición para un tipo. Estamos diciendo “ `handles` es un vector de algo, pero tu, Rust, puedes determinar que es ese algo.”

```
filosofos.into_iter().map(|f| {
```

Tomamos nuestra lista de filósofos y llamamos `into_iter()` en ella. Esto crea un iterador que se adueña (toma pertenencia) de cada filosofo. Necesitamos hacer esto para poder pasar los filósofos a nuestros hilos. Luego tomamos ese iterador y llamamos `map` en el, método que toma un closure como argumento y llama dicho closure en cada uno de los elementos a la vez.

```
    thread::spawn(move || {
        f.comer();
    })
```

Es aquí donde la concurrencia ocurre. La función `thread::spawn` toma un closure como argumento y ejecuta ese closure en un nuevo hilo. El closure necesita una anotación extra, `move`, para indicar que el closure va a adueñarse de los valores que esta capturando. Principalmente, la variable `f` de la función `map`.

Dentro del hilo, todo lo que hacemos es llamar a `comer();` en `f`.

```
}).collect();
```

Finalmente, tomamos el resultado de todos esas llamadas a `map` y los coleccionamos. `collect()` los convertirá en una colección de alguna tipo, que es el porque anotamos el tipo de retorno: queremos un `Vec<T>`. Los elementos son los valores retornados de las llamadas a `thread::spawn`, que son handles a esos hilos. Whew!

```
for h in handles {
    h.join().unwrap();
}
```

Al final de `main()`, iteramos a través de los handles llamando `join()` en ellos, lo cual bloquea la ejecución hasta que el hilo haya completado su ejecución. Esto asegura que el hilo complete su ejecución antes que el programa termine.

Si ejecutas este programa, veras que los filósofos comen sin orden! Tenemos multi-hilos!

```
Gilles Deleuze esta comiendo.
Gilles Deleuze ha finalizado de comer.
Emma Goldman esta comiendo.
Emma Goldman ha finalizado de comer.
Michel Foucault esta comiendo.
Judith Butler esta comiendo.
Judith Butler ha finalizado de comer.
Karl Marx esta comiendo.
Karl Marx ha finalizado de comer.
Michel Foucault ha finalizado de comer.
```

Pero que acerca de los tenedores no los hemos modelado del todo todavía.

Para hacerlo, creemos un nuevo `struct`:

```
use std::sync::Mutex;

struct Mesa {
    tenedores: Vec<Mutex<()>>,
}
```

Esta `Mesa` contiene un vector de `Mutex` es. Un mutex es una forma de controlar concurrencia, solo un hilo puede acceder el contenido a la vez. Esta es la exactamente la propiedad que necesitamos para nuestros tenedores. Usamos una dupla vacía, `()`, dentro del mutex, debido a que no vamos a usar el valor, solo nos aferraremos a el.

Modifiquemos el programa para hacer uso de `Mesa`:

```
use std::thread;
use std::sync::{Mutex, Arc};

struct Filosofo {
    nombre: String,
    izquierda: usize,
    derecha: usize,
}

impl Filosofo {
    fn new(nombre: &str, izquierda: usize, derecha: usize) -> Filosofo {
        Filosofo {
            nombre: nombre.to_string(),
            izquierda: izquierda,
```

```
            derecha: derecha,
        }
    }

    fn comer(&self, mesa: &Mesa) {
        let _izquierda = mesa.tenedores[self.izquierda].lock().unwrap();
        let _derecha = mesa.tenedores[self.derecha].lock().unwrap();

        println!("{} esta comiendo.", self.nombre);

        thread::sleep_ms(1000);

        println!("{} ha finalizado de comer.", self.nombre);
    }
}

struct Mesa {
    tenedores: Vec<Mutex<()>>,
}

fn main() {
    let mesa = Arc::new(Mesa { tenedores: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});

    let filosofos = vec![
        Filosofo::new("Judith Butler", 0, 1),
        Filosofo::new("Gilles Deleuze", 1, 2),
        Filosofo::new("Karl Marx", 2, 3),
        Filosofo::new("Emma Goldman", 3, 4),
        Filosofo::new("Michel Foucault", 0, 4),
    ];

    let handles: Vec<_> = filosofos.into_iter().map(|f| {
        let mesa = mesa.clone();

        thread::spawn(move || {
            f.comer(&mesa);
        })
    }).collect();

    for h in handles {
        h.join().unwrap();
    }
}
```

Muchos cambios! Sin embargo, con esta iteración, hemos obtenido un programa funcional. Veamos los detalles:

```
use std::sync::{Mutex, Arc};
```

Usaremos otra estructura del paquete `std::sync`: `Arc<T>`.

Hablaremos mas acerca de ella cuando la usemos.

```
struct Filosofo {
    nombre: String,
    izquierda: usize,
    derecha: usize,
}
```

Vamos a necesitar agregar dos campos mas a nuestra estructura `Filosofo`. Cada filosofo tendrá dos tenedores: el de la izquierda, y el de la derecha. Usaremos el tipo `usize` para indicarlos, debido a que este es el tipo con el cual se indexan los vectores. Estos dos valores serán indices en los `tenedores` que nuestra `Mesa` posee.

```
fn new(nombre: &str, izquierda: usize, derecha: usize) -> Filosofo {
    Filosofo {
        nombre: nombre.to_string(),
        izquierda: izquierda,
        derecha: derecha,
    }
}
```

Ahora necesitamos construir esos valores `izquierda` y `derecha`, de manera que podamos agregarlos a `new()`.

```
fn comer(&self, mesa: &Mesa) {
    let _izquierda = mesa.tenedores[self.izquierda].lock().unwrap();
    let _derecha = mesa.tenedores[self.derecha].lock().unwrap();

    println!("{} esta comiendo.", self.nombre);

    thread::sleep_ms(1000);

    println!("{} ha finalizado de comer.", self.nombre);
}
```

Tenemos dos nuevas lineas, también hemos agregado un argumento, `mesa`. Accedemos a la lista de tenedores de la `Mesa`, y después usamos `self.izquierda` y `self.derecha` para acceder al tenedor en un indice en particular. Eso nos da acceso al `Mutex` en ese indice, en donde llamamos `lock()`. Si el mutex esta siendo accedido actualmente por alguien mas, nos bloquearemos hasta que este disponible.

La llamada a `lock()` puede fallar, y si lo hace, queremos terminar abruptamente. En este caso el error que puede ocurrir es que el mutex este 'envenenado' ('poisoned'), que es lo que ocurre cuando el hilo hace pánico mientras el mantiene el bloqueo. Debido a que esto no debería ocurrir, simplemente usamos `unwrap()`.

Otra cosa extraña acerca de esta lineas: hemos nombrado los resultados `_izquierda` and `_derecha`. Que hay con ese sub-guion? Bueno, en realidad no planeamos *usar* el valor dentro del bloqueo. Solo queremos adquirirlo. A consecuencia, Rust nos advertirá que nunca usamos el valor. A través del uso del sub-guion le decimos a Rust que es lo que quisimos, de esa manera no generara la advertencia.

Que acerca de soltar el bloqueo?, Bueno, esto ocurrirá cuando `_izquierda` y `_derecha` salgan de ámbito, automáticamente.

```
    let mesa = Arc::new(Mesa { tenedores: vec![
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
        Mutex::new(()),
    ]});
```

A continuación, en `main()`, creamos una nueva `Mesa` y la envolvemos en un `Arc<T>`. 'arc' proviene de 'atomic reference count' (cuenta de referencias atómica), necesitamos compartir nuestra `Mesa` entre multiples hilos. A media que la compartimos, la cuenta de referencias subirá, y cuando cada hilo termine, ira bajando.

```
let filosofos = vec![
    Filosofo::new("Judith Butler", 0, 1),
    Filosofo::new("Gilles Deleuze", 1, 2),
    Filosofo::new("Karl Marx", 2, 3),
    Filosofo::new("Emma Goldman", 3, 4),
    Filosofo::new("Michel Foucault", 0, 4),
];
```

Necesitamos pasar nuestros valores `izquierda` and `derecha` a los constructores de nuestros `Filosofo`s. Pero hay un detalle mas aquí, y es *muy* importante. Si observas al patrón, es consistente hasta el final, Monsieur Foucault debe tener `4, 0` como argumentos, pero en vez de esto tiene `0, 4`. Esto es lo que previene deadlocks, en efecto: uno de los filósofos es zurdo! Esa es una forma de resolver el problema, y en mi opinion, es la mas simple.

```
let handles: Vec<_> = filosofos.into_iter().map(|f| {
    let mesa = mesa.clone();

    thread::spawn(move || {
        f.comer(&mesa);
    })
}).collect();
```

Finalmente, dentro de nuestro ciclo `map()` / `collect()`, llamamos `mesa.clone()`. El método `clone()` en `Arc<T>` es lo que incrementa la cuenta de referencias, y cuando sale de ámbito, la decrementa. Notaras que podemos introducir una nueva variable `mesa`, y esta sobre escribirá (shadow) la anterior. Esto es frecuentemente usado de manera tal de no tener que inventar dos nombres únicos.

Con todo esto, nuestro programa funciona! Solo dos filosofo pueden comer en un momento dado y en consecuencia tendrás salida se vera así:

```
Gilles Deleuze esta comiendo.
Emma Goldman esta comiendo.
Emma Goldman ha finalizado de comer.
Gilles Deleuze ha finalizado de comer.
Judith Butler esta comiendo.
Karl Marx esta comiendo.
Judith Butler ha finalizado de comer.
Michel Foucault esta comiendo.
Karl Marx ha finalizado de comer.
Michel Foucault ha finalizado de comer.
```

Felicitaciones! Haz implementado un problema clásico de concurrencia en Rust.

% Rust Dentro de Otros Lenguajes

Para nuestro tercer proyecto, elegiremos algo que demuestra una de las mayores fortalezas de Rust: la ausencia de un entorno de ejecución.

A medida que las organizaciones crecen, estas de manera progresiva, hacen uso de una multitud de lenguajes de programación. Diferentes lenguajes de programación poseen diferentes fortalezas y debilidades, y una arquitectura poliglota permite usar un lenguaje en particular donde sus fortalezas hagan sentido y otro lenguaje sea débil.

Un area muy común en donde muchos lenguajes de programación son débiles es el performance en tiempo de ejecución. Frecuentemente, usar un lenguaje que es lento, pero ofrece mayor productividad para el programador, es un equilibrio que vale la pena. Para ayudar a mitigar esto, dichos lenguajes proveen una manera de escribir una parte de tu sistema en C y luego llamar ese código como si hubiese sido escrito en un lenguaje de mas alto nivel. Esta facilidad es denominada ‘interfaz de funciones foráneas’ (‘foreign function interface’), comúnmente abreviando a ‘FFI’.

Rust posee soporte para FFI en ambas direcciones: puede llamar código en C de manera fácil, pero crucialmente puede ser *llamado* tan fácilmente como C. Combinado con la ausencia de un recolector de basura y bajos requerimientos en tiempo de ejecución, Rust es un candidato para ser embebido dentro de otros lenguajes cuando necesites esos ooKmh extra.

Existe un capitulo completo dedicado a FFI y sus detalles en otra parte del libro, pero en este capitulo, examinaremos el uso particular de FFI con ejemplos en Ruby, Python, y JavaScript.

El problema

Existen muchos problemas que podríamos haber escogido, pero elegiremos un ejemplo en el cual Rust tiene una ventaja clara por encima de otros lenguajes: computación numérica e hilos.

Muchos lenguajes, en honor a la consistencia, colocan números en el montículo, en vez de en la pila. Especialmente en lenguajes enfocados en programación orientada a objetos y el uso de un recolector de basura, la asignación de memoria desde el montículo es el comportamiento por defecto. Algunas veces optimizaciones pueden colocar ciertos números en la pila, pero en vez de confiar en un optimizador para realizar este trabajo, podríamos querer asegurarnos que siempre estemos usando numero primitivos en vez de algun tipo de objetos.

Segundo, muchos lenguajes poseen un 'bloqueo global del interprete' ('global interpreter lock') (GIL), que limita la concurrencia en muchas situaciones. Esto es hecho en el nombre de la seguridad lo cual es un efecto positivo, pero limita la cantidad de trabajo que puede ser llevado a cabo de manera concurrente, lo cual es un gran negativo.

Para enfatizar estos 2 aspectos, crearemos un pequeño proyecto que usa estos dos aspectos en gran medida. Debido a que el foco del ejemplo es embeber Rust en otros lenguajes, en vez de el problema en si mismo, usaremos un ejemplo de juguete:

> Inicia diez hilos. Dentro de cada hilo, cuenta desde uno hasta cinco millones. Después que todos los hilos hayan finalizado, imprime "completado!".

He escogido cinco millones basado en mi computador en particular. He aquí un ejemplo de este código en Ruby:

```
threads = []

10.times do
  threads << Thread.new do
    count = 0

    5_000_000.times do
      count += 1
    end
  end
end

threads.each { |t| t.join }
puts "completado!"
```

Intenta ejecutar este ejemplo, y escoge un numero que corra por unos segundos. Dependiendo en el hardware de tu computador, tendrás que incrementar o decrementar el numero.

En mi sistema, ejecutar este programa toma `2.156`. Si uso alguna tipo de herramienta de monitoreo de procesos, como `top`, puedo ver que solo usa un núcleo en mi maquina. El GIL presente haciendo su trabajo.

Si bien es cierto que este en un programa sintético, uno podría imaginar muchos problemas similares a este en el mundo real. Para nuestros propósitos, levantar unos pocos hilos y ocuparlos representa una especie de computación paralela y costosa.

Una biblioteca Rust

Escribamos este problema en Rust. Primero, creemos un proyecto nuevo con Cargo:

```
$ cargo new embeber
$ cd embeber
```

Este programa es fácil de escribir en Rust:

```
use std::thread;

fn procesar() {
    let handles: Vec<_> = (0..10).map(|_| {
        thread::spawn(|| {
            let mut _x = 0;
            for _ in (0..5_000_000) {
                _x += 1
            }
        })
    }).collect();

    for h in handles {
        h.join().ok().expect("No se pudo unir un hilo!");
    }
}
```

Algo de esto debe lucir familiar a ejemplos anteriores. Iniciamos diez hilos, colectandolos en un vector `handles`. Dentro de cada hilo, iteramos cinco millones de veces, agregando uno a `_x` en cada iteración. Porque el sub-guion? Bueno, si lo removemos y luego compilamos:

```
$ cargo build
  Compiling embeber v0.1.0 (file:///Users/goyox86/Code/rust/embeber)
src/lib.rs:3:1: 16:2 warning: function is never used: `procesar`, #[warn(dead_code)] on b
src/lib.rs:3 fn procesar() {
src/lib.rs:4     let handles: Vec<_> = (0..10).map(|_| {
src/lib.rs:5         thread::spawn(|| {
src/lib.rs:6             let mut x = 0;
src/lib.rs:7             for _ in (0..5_000_000) {
src/lib.rs:8                 x += 1
             ...
src/lib.rs:6:17: 6:22 warning: variable `x` is assigned to, but never used, #[warn(unused
src/lib.rs:6             let mut x = 0;
                             ^~~~~
```

La primera advertencia es debido es a consecuencia de estar construyendo una biblioteca. Si tuviéramos una prueba para esta función, la advertencia desaparecería. Pero por ahora nunca es llamada.

La segunda esta relacionada a `x` versus `_x`. Como producto de que efectivamente *no hacemos nada* con `x` obtenemos una advertencia. Eso, en nuestro caso, esta perfectamente bien, puesto que queremos desperdiciar ciclos de CPU. Usando un sub-guión de prefijo eliminamos la advertencia.

Finalmente, hacemos join en cada uno de los hilos.

Hasta el momento, sin embargo, es una biblioteca Rust, y no expone nada que pueda ser llamado desde C. Si quisiéramos conectarla con otro lenguaje, en su estado actual, no funcionaria. Solo necesitamos hacer unos pequeños cambios para arreglarlo. Lo primero es modificar el principio de nuestro código:

```
#[no_mangle]
pub extern fn procesar() {
```

Debemos agregar un nuevo atributo, `no_mangle`. Cuando creamos una biblioteca Rust, este cambia el nombre de la función en la salida compilada. Las razones de esto escapan del alcance de este tutorial, pero para que otros lenguajes puedan saber como llamar a la función, debemos evitar que el compilador cambie el nombre en la salida compilada. Este atributo desactiva ese comportamiento.

El otro cambio es el `pub extern`. El `pub` significa que esta función puede ser llamada desde afuera de este modulo, y el `extern` dice que esta puede ser llamada desde C. Eso es todo! No muchos cambios.

La segunda cosa que necesitamos hacer es cambiar una configuración en nuestro `Cargo.toml`. Agrega esto al final:

```
[lib]
name = "embeber"
crate-type = ["dylib"]
```

Estas lineas le informan a Rust que queremos compilar nuestra biblioteca en una biblioteca dinámica estándar. Rust compila un ‘rlib’, un formato especifico de Rust.

Ahora construyamos el proyecto:

```
$ cargo build --release
   Compiling embeber v0.1.0 (file:///Users/goyox86/Code/rust/embeber)
```

Hemos elegido `cargo build --release`, lo cual construye el proyecto con optimizaciones. Queremos que sea lo mas rápido posible! Puedes encontrar la salida de la biblioteca en `target/release`:

```
$ ls target/release/
build  deps  examples  libembeber.dylib native
```

Esa `libembeber.dylib` es nuestra biblioteca de 'objetos compartidos'. Podemos usar esta biblioteca como cualquier biblioteca de objetos compartido escrita en C! Como nota, esta podría ser `libembeber.so` o `libembeber.dll` , dependiendo la plataforma.

Ahora que tenemos nuestra biblioteca Rust, usémosla desde Ruby.

Ruby

Crea un archivo `embeber.rb` dentro de nuestro proyecto, y coloca esto dentro:

```
require 'ffi'

module Hola
  extend FFI::Library
  ffi_lib 'target/release/libembeber.dylib'
  attach_function :procesar, [], :void
end

Hola.procesar

puts 'completado!'
```

Antes de que podamos ejecutarlo, necesitamos instalar la gema `ffi` :

```
$ gem install ffi # esto puede necesitar sudo
Fetching: ffi-1.9.8.gem (100%)
Building native extensions.  This could take a while...
Successfully installed ffi-1.9.8
Parsing documentation for ffi-1.9.8
Installing ri documentation for ffi-1.9.8
Done installing documentation for ffi after 0 seconds
1 gem installed
```

Finalmente, intentemos ejecutarlo:

```
$ ruby embeber.rb
completado!
$
```

Whoa, eso fue rápido! En mi sistema, tomo `0.086` segundos, a diferencia de los dos segundos que la version en Ruby puro. Analicemos este código Ruby:

```
require 'ffi'
```

Primero necesitamos requerir la gema `ffi`. Nos permite interactuar con una biblioteca Rust como una biblioteca en C.

```
module Hola
  extend FFI::Library
  ffi_lib 'target/release/libembeber.dylib'
```

El modulo `Hola` es usado para adjuntar las funciones nativas de la biblioteca compartida. Dentro, `extend` emos el modulo `FFI::Library` y luego llamamos el método `ffi_lib` para cargar nuestra biblioteca de objetos compartidos. Simplemente pasamos la ruta en la cual nuestra biblioteca esta almacenada, la cual, como vimos anteriormente, es `target/release/libembeber.dylib`.

```
attach_function :procesar, [], :void
```

El método `attach_function` es proporcionado por la gema FFI. Es lo que conecta nuestra función `procesar()` en Rust a un método en Ruby con el mismo nombre. Debido a que `procesar()` no recibe argumentos, el segundo parámetro es un arreglo vacío, y ya que no retorna nada, pasamos `:void` como argumento final.

```
Hola.procesar
```

Esta es la llamada a Rust. La combinación de nuestro modulo y la llamada a `attach_function` han configurado todo. Se ve como un método Ruby pero es en realidad código Rust!

```
puts 'completado!'
```

Finalmente, y como requerimiento de nuestro proyecto, imprimimos `completado!`.

Eso es todo! Como hemos visto, hacer un puente entre los dos lenguajes es realmente fácil, y nos compra mucho performance.

A continuación, probemos Python!

Python

Crea un archivo `embeber.py` en este directorio, y coloca esto en el:

```
from ctypes import cdll

lib = cdll.LoadLibrary("target/release/libembeber.dylib")

lib.procesar()

print("completado!")
```

Aun mas fácil! Usamos `cdll` del modulo `ctypes`. Una llamada rápida a `LoadLibrary` después, y luego podemos llamar `procesar()`.

En mi sistema, toma `0.017` segundos. Rápidillo!

Node.js

Node no es un lenguaje, pero es actualmente la implementación de Javascript dominante del lado del servidor.

Para hacer FFI con Node, primero necesitamos instalar la biblioteca:

```
$ npm install ffi
```

Después de que este instalada, podemos usarla:

```
var ffi = require('ffi');

var lib = ffi.Library('target/release/libembeber', {
  'procesar': ['void', []]
});

lib.procesar();

console.log("completado!");
```

Luce mas parecido al ejemplo Ruby que al de Python. Usamos el modulo `ffi` para obtener acceso a `ffi.Library()`, la cual nos permite cargar nuestra biblioteca de objetos compartidos. Necesitamos anotar el tipo de retorno y los tipos de los argumentos de la

función, que son `void` para el retorno y un arreglo vacío para representar ningún argumento. De allí simplemente llamamos a la función `procesar()` e imprimimos el resultado.

En my sistema, este ejemplo toma unos rápidos `0.092` segundos.

Conclusion

Como puedes ver, las bases de hacer FFI son *muy* fáciles. Por supuesto hay mucho mas que podríamos hacer aquí. Echa un vistazo al capitulo FFI para mas detalles.

% Rust Efectivo

Entonces, haz aprendido como escribir algo de código Rust. Pero hay una diferencia entre escribir *cualquier* código Rust y escribir *buen* código Rust.

Esta sección consiste de tutoriales relativamente independientes que te muestran como llevar tu Rust al siguiente nivel. Patrones comunes y características de la biblioteca estándar serán presentados. Puedes leer dichas secciones en el orden que prefieras.

% La Pila y el Montículo

Como un lenguaje de sistemas, Rust opera a un bajo nivel. Si provienes de un lenguaje de alto nivel, hay algunos aspectos de los lenguajes de programación de sistemas con los cuales puedas no estar familiarizado. El mas importante es el funcionamiento de la memoria, con la pila y el montículo. Si estas familiarizado con el como lenguajes como C usan asignación desde la pila, este capitulo sera un repaso. Si no lo estas, aprenderás acerca de este concepto general, pero con un enfoque Rustero.

Manejo de memoria

Estos dos términos hacen referencia a el manejo de la memoria. La pila y el montículo son abstracciones que ayudan a determinar cuando asignar y liberar memoria.

He aquí una comparación de alto nivel:

La pila es muy rápida, y es de donde la memoria es asignada por defecto en Rust. Pero la asignación es local a una llamada a función, y es limitada en tamaño. El montículo por otro lado, es mas lento, y es asignado por tu programa. Pero es efectivamente de un tamaño ilimitado, y es globalmente accesible.

La Pila

Hablemos acerca de este programa Rust:

```
fn main() {
    let x = 42;
}
```

Este programa posee una variable (variable binding), `x`. La memoria tiene que ser asignada desde algún sitio. Rust asigna desde la pila por defecto, lo que se traduce en que los valores básicos ‘van a la pila’. Pero, que significa esto?

Veamos, cuando una función es llamada, algo de memoria es asignada para sus variables locales y otra información extra. Dicha memoria es llamada ‘registro de activación’ (‘stack frame’), para el propósito de este tutorial, ignoraremos la información extra y solo consideraremos las variables locales a las que estamos asignando memoria. Así que en este caso, cuando `main()` es ejecutada, asignamos un entero de 32 bits para nuestro registro de activación. Todo esto es manejado automáticamente, como has podido ver, no tuvimos que escribir ningún código Rust especial o alguna otra cosa.

Cuando la función termina, su registro de activación es liberado o desasignado. Esto ocurre de manera automática, no tuvimos que hacer nada especial acá.

Eso es todo para este simple programa. Lo clave a entender aquí es que la asignación de memoria desde la pila es muy, muy rápida. Debido a que conocemos por adelantado todas las variables locales, podemos obtener toda la memoria de una sola vez. Y debido a que la desecharemos toda completa, podemos deshacernos de ella muy rápido, también.

La desventaja es que no podemos mantener valores rondando por allí si los necesitamos por un periodo mas largo que el tiempo de vida de una función. Tampoco hemos hablado acerca de que significa ese nombre, 'pila'. Para hacerlo necesitamos un ejemplo ligeramente mas complejo:

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
    let x = 42;

    foo();
}
```

Este programa tiene tres variables en total: dos en `foo()`, una en `main()`. Al igual que antes, cuando `main()` es llamada, un solo entero es asignado para su registro de activación. Pero antes que demostremos que es lo que pasa cuando `foo()` es llamada, necesitamos visualizar que es lo que esta pasando en memoria. Tu sistema operativo presenta a tu programa una visión muy simple: una lista inmensa de direcciones, desde 0 hasta un numero muy grande, que representa cuanta memoria RAM posee la maquina. Por ejemplo si tienes un gigabyte de RAM, tus direcciones irán desde 0 hasta `1,073,741,824`, numero que proviene de 2^{30}, el numero de bytes en un gigabyte.

Esta memoria es una especie de arreglo gigante: las direcciones comienzan en cero y se incrementan hasta el numero final. Entonces, he aquí un diagrama de nuestro primer registro de activación:

Dirección	Nombre	Valor
0	x	42

Hemos colocado a `x` en la dirección `0`, con el valor `42`

Cuando `foo()` es llamada un nuevo registro de activación es asignado:

Dirección	Nombre	Valor
2	z	100
1	y	5
0	x	42

Debido a que `0` fue reservado para la primera frame, `1` y `2` son usados para el registro de activación de `foo()`. La pila crece hacia arriba, a medida que llamamos a mas funciones.

Hay algunas cosas importantes que debemos notar aquí. Los números 0, 1 y 2 existen solo para propósitos ilustrativos, y no poseen ninguna relación con los números que una computadora realmente usaría. En particular, la serie de direcciones están separadas por un numero de bytes, y esa separación puede incluso exceder el tamaño del valor que esta siendo almacenado.

Después que `foo()` termina, su registro de activación es liberado:

Dirección	Nombre	Valor
0	x	42

Y luego después de que `main()` finaliza, este ultimo valor se va. Fácil!

Es llamada una pila (‘stack’) debido a que funciona como una pila de platos: el primer plato que colocas es el ultimo plato que sacarás. Las pilas son algunas veces llamadas ‘colas ultimo que entra, primero que sale’ (‘last in, first out queues’), por estas razones el ultimo valor que pusiste en la pila será el primero que obtendrás de ella.

Probemos un ejemplo de tres niveles:

```
fn bar() {
    let i = 6;
}

fn foo() {
    let a = 5;
    let b = 100;
    let c = 1;

    bar();
}

fn main() {
    let x = 42;

    foo();
}
```

Bien, en primera instancia, llamamos a `main()` :

Dirección	Nombre	Valor
0	x	42

Acto seguido, `main()` llama a `foo()` :

Dirección	Nombre	Valor
3	c	1
2	b	100
1	a	5
0	x	42

Luego `foo()` llama a `bar()` :

Dirección	Nombre	Valor
4	i	6
3	c	1
2	b	100
1	a	5
0	x	42

Uff! Nuestra pila esta creciendo.

Después que `bar()` termina, su registro de activación es liberado, dejando solo a `foo()` y `main()` :

Dirección	Nombre	Valor
3	c	1
2	b	100
1	a	5
0	x	42

Después `foo()` termina, dejando solo a `main()`

Dirección	Nombre	Valor
0	x	42

Hemos terminado entonces. Se entiende? Es como apilar platos: agregas al tope y sacas de el.

El Montículo

Ahora, todo esto trabaja bien, pero no todo funciona de esa manera. Algunas veces, necesitas pasar memoria entre diferentes funciones, o mantener memoria viva por un tiempo mayor que la ejecución de una función. Para esto usamos el montículo.

En Rust, puedes asignar memoria desde el montículo con el tipo `Box<T>` (caja).

He aqui un ejemplo:

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

Acá, lo que sucede cuando `main()` es llamada:

Dirección	Nombre	Valor
1	y	42
0	x	??????

Asignamos espacio para dos variables en la pila. `y` es `42` , como conocemos hasta ahora, pero que acerca de `x` ? Bueno, `x` es un `Box<i32>` , y las cajas (boxes) asignan memoria desde el montículo. El valor de la caja en cuestión es una estructura que posee un

apuntador a 'el montículo'. Cuando comienza la ejecución de la función, y `Box::new()` es llamada, esta asigna algo de memoria para el montículo y coloca `5` allí. La memoria ahora luce así:

Dirección	Nombre	Valor
2^{30}		5
...	...	...
1	y	42
0	x	2^{30}

Tenemos 2^{30} en nuestra computadora hipotética con 1GB de RAM. Y debido a que nuestra pila crece desde cero, la forma mas fácil para asignar memoria es desde el otro extremo. Entonces nuestro primer valor esta en el lugar mas alto en la memoria. Y el valor de la estructura en `x` tiene un apuntador plano (raw pointer) a el lugar que hemos asignado en el montículo, entonces el valor de `x` es 2^{30}, la dirección de la memoria que hemos solicitado.

No hemos hablado mucho acerca de que significa en realidad asignar y liberar memoria en estos contextos. Entrar en el profundo detalle de ello esta fuera del alcance de este tutorial, lo importante a resaltar es que el montículo no es una simple pila que crece desde el lado opuesto. Tendremos un ejemplo de esto mas adelante en el libro, pero debido a que el montículo puede ser asignado y liberado en cualquier orden, puede terminar con 'vacios'. He aquí un diagrama de la distribución de la memoria de un programa que ha estado corriendo por algún tiempo:

Dirección	Nombre	Valor
2^{30}		5
$(2^{30}) - 1$		
$(2^{30}) - 2$		
$(2^{30}) - 3$		42
...	...	...
3	y	$(2^{30}) - 3$
2	y	42
1	y	42
0	x	2^{30}

En este caso, hemos asignado cuatro cosas en el montículo, pero hemos liberado dos de ellas. Hay un vacío entre 2^{30} y (2^{30}) - 3 que no esta siendo usado actualmente. El detalle especifico acerca de como y porque esto sucede depende de la estrategia usada para manejar el montículo. Diferentes programas pueden usar diferentes 'asignadores de memoria' ('memory allocators'), que son bibliotecas encargadas de manejar la asignación de memoria por ti. Los programas en Rust usan jemalloc para dicho propósito.

De cualquier modo, y de vuelta a nuestro ejemplo. Debido a que esta memoria esta en el montículo, puede permanecer viva mas tiempo que la función que crea la caja (box). Sin embargo, en este caso, esto no sucede. [moving] cuando la función termina, necesitamos liberar el registro de activación de `main()`. `Box<T>`, sin embargo, tienen un truco bajo la manga: Drop. La implementación de `Drop` para `Box` libera la memoria que ha sido asignada cuando la caja es creada. Grandioso! Así que cuando `x` se va (sale de contexto), primero libera la memoria asignada desde el montículo:

Dirección	Nombre	Valor
1	y	42
0	x	??????

> [moving]. Podemos hacer que la memoria permanezca viva mas tiempo transfiriendo la pertenencia (ownership), algunas veces llamado 'moviendo fuera de la caja' ('moving out of the box'). Ejemplos mas complejos serán cubiertos mas adelante. ↩

Luego el registro de activación se va, liberando toda nuestra memoria.

Argumentos y prestamo (borrowing)

Hemos llevado a cabo algunos ejemplos básicos con la pila y el montículo, pero que hay acerca de los argumentos a funciones y el préstamo (borrowing)? He aquí un pequeño programa Rust:

```
fn foo(i: &i32) {
    let z = 42;
}

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

    foo(y);
}
```

Cuando entramos a `main()`, la memoria luce de la siguiente manera:

Dirección	Nombre	Valor
1	y	0
0	x	5

`x` es un simple `5`, y `y` es una referencia a `x`. Entonces, el valor de `y` es la dirección de memoria en la que `x` vive, que en este caso es `0`.

Que sucede cuando llamamos a `foo()` pasando a `y` como argumento?

Dirección	Nombre	Valor
3	z	42
2	i	0
1	y	0
0	x	5

Los registros de activación no son solo para variables locales, son también para argumentos. En este caso, necesitamos tener ambos `i`, nuestro argumento, y `z` nuestra variable local. `i` es una copia del argumento, `y`. Debido a que el valor de `y` es `0` entonces ese es el valor de `i`.

Esta es una razón por la cual tomar prestada una variable no libera ninguna memoria: el valor de la referencia es solo un apuntador a una dirección de memoria. Si nos deshiciéramos de la memoria subyacente, las cosas no irían del todo bien.

Un ejemplo complejo

Bien, vayamos a través de este programa complejo paso-a-paso:

```
fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
    let d = Box::new(5);
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

Primero, llamamos a `main()`:

Dirección	Nombre	Valor
2^{30}		20
...	...	...
2	j	0
1	i	2^{30}
0	h	3

Asignamos memoria para `j`, `i`, y `h`. `i` esta en el montículo, es por ello que su valor apunta hacia el.

A continuation, al final de `main()`, `foo()` es llamada:

Dirección	Nombre	Valor
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Es asignado espacio para `x`, `y`, y `z`. El argumento `x` tiene el mismo valor que `j`, debido a que eso fue lo que le proporcionamos a la función. Es un apuntador a la dirección `0`, puesto que `j` apunta a `h`.

Seguidamente, `foo()` llama a `baz()`, pasándole `z`:

Dirección	Nombre	Valor
2^{30}		20
...	...	...
7	g	100
6	f	4
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Hemos asignado memoria para `f` y `g`. `baz()` es muy corta, así que cuando termina, nos deshacemos de su registro de activación:

Dirección	Nombre	Valor
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Después, `foo()` llama a `bar()` con `x` y `z`:

Dirección	Nombre	Valor
2^{30}		20
(2^{30}) - 1		5
...	...	...
10	e	9
9	d	(2^{30}) - 1
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Terminamos asignando otro valor en el montículo, así que tenemos que restar uno a 2^{30}. Es mas fácil escribir eso que `1,073,741,823`. En cualquier caso, seteamos las variables como ya es usual.

Al final de `bar()`, esta llama a `baz()`:

Dirección	Nombre	Valor
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
12	g	100
11	f	9
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Con esto, estamos en nuestro punto mas profundo! Wow! Felicitaciones por haber seguido todo esto y haber llegado tan lejos.

Luego `baz()` termina, nos deshacemos de `f` y `g`:

Dirección	Nombre	Valor
2^{30}		20
$(2^{30}) - 1$		5
...	...	...
10	e	9
9	d	$(2^{30}) - 1$
8	c	5
7	b	4
6	a	0
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

A continuación, retornamos de `bar()`. `d` en este caso es un `Box<T>`, entonces también libera a lo que apunta: $(2^{30}) - 1$.

Dirección	Nombre	Valor
2^{30}		20
...	...	...
5	z	4
4	y	10
3	x	0
2	j	0
1	i	2^{30}
0	h	3

Después, `foo()` retorna:

Dirección	Nombre	Valor
2^{30}		20
...	...	...
2	j	0
1	i	2^{30}
0	h	3

Entonces, finalmente `main()` retorna, lo cual limpia el resto. Cuando `i` es liberada (a través de `Drop`) esta limpiara también lo restante en el montículo.

Que hacen otros lenguajes?

La mayoría de los lenguajes con un recolector de basura asignan desde el montículo por defecto. Esto significa que todos los valores están dentro de cajas (boxed). Existen un numero de razones por la cuales esto se hace de esta manera, pero están fuera del alcance de este tutorial. También, existen algunas optimizaciones que hacen que esto no sea 100% verdad todo el tiempo. En vez de confiar en la pila y `Drop` para limpiar la memoria, el recolector de basura es el encargado de administrar el montículo.

Cual usar?

Si la pila es mas rápida y mas fácil de usar, porque necesitamos el montículo? Una gran razón es que la asignación desde la pila significa que solo tienes semántica LIFO para reclamar almacenamiento. La asignación desde el montículo es estrictamente mas general, permitiendo que el almacenamiento pueda ser tomado y retornado a el pool en orden arbitrario, pero con un costo en complejidad.

Generalmente, deberías preferir asignación desde la pila, es por ello que Rust asigna desde la pila por defecto. El modelo LIFO de la pila es mas simple, a nivel fundamental. Esto tiene dos grandes impactos: eficiencia en tiempo de ejecución e impacto semántico.

Eficiencia en tiempo de Ejecucion.

Administrar la memoria para la pila es trivial: La maquina simplemente incrementa un solo valor, el llamado "apuntador a la pila" (“stack pointer”). La administración de memoria para el montículo no lo es: La memoria asignada desde el montículo es liberada en puntos

arbitrarios, y cada bloque de memoria asignada desde el montículo pude ser de un tamaño arbitrario, el administrador de memoria generalmente debe trabajar mucho mas duro para identificar memoria que pueda ser reusada.

Si quisieras sumergirte mas en este tópico con mayor detalle, este paper es una muy buena introducción.

Impacto semantico

Stack-allocation impacts the Rust language itself, and thus the developer's mental model. The LIFO semantics is what drives how the Rust language handles automatic memory management. Even the deallocation of a uniquely-owned heap-allocated box can be driven by the stack-based LIFO semantics, as discussed throughout this chapter. The flexibility (i.e. expressiveness) of non LIFO-semantics means that in general the compiler cannot automatically infer at compile-time where memory should be freed; it has to rely on dynamic protocols, potentially from outside the language itself, to drive deallocation (reference counting, as used by Rc and Arc, is one example of this).

La asignación desde la pila impacta a Rust como lenguaje, y con ello el modelo mental del desarrollador. La semántica LIFO es lo que conduce como el lenguaje Rust maneja el manejo automático de memoria. Incluso la liberación de una caja asignada desde el montículo con un único dueño puede ser manejada por la semántica LIFO, tal y como se ha discutido en este capitulo. La flexibilidad (e.j. expresividad) de la semántica no-LIFO significa que en general el compilador no puede inferir de manera automática y en tiempo de compilación donde la memoria debería ser liberada; tiene que apoyarse en protocolos dinámicos, potencialmente externos a el lenguaje, para efectuar liberación de memoria (conteo de referencias, como el usado en `Rc<T>` y `Arc<T>`, es un ejemplo).

Cuando se lleva al extremo, el mayor poder expresivo de la asignación desde el montículo viene a costo de bien sea soporte significativo en tiempo de ejecución (e.j. en la forma de un recolector de basura) o esfuerzo significativo por parte del programador (en la forma de llamadas manuales explícitas que requieren verificación no proporcionada por el compilador de Rust).

% Pruebas

> Probar programas puede ser una forma efectiva de mostrar la presencia de bugs, pero es desesperanzadamente inadecuada para mostrar su ausencia. Edsger W. Dijkstra, "The Humble Programmer" (1972)

Hablaremos acerca de como probar código Rust. De lo que no estaremos hablando es acerca de la manera correcta de probar código Rust. Hay muchas escuelas de pensamiento en relación a la forma correcta e incorrecta de escribir pruebas. Todos esos enfoques usan las mismas herramientas básicas, en esta sección te mostraremos la sintaxis para hacer uso de ellas.

El atributo `test`

En esencia, una prueba en Rust es una función que esta anotada con el atributo `test`. Vamos a crear un nuevo proyecto llamado `sumador` con Cargo:

```
$ cargo new sumador
$ cd adder
```

Cargo generara automáticamente una prueba simple cuando creas un proyecto nuevo. He aqui el contenido de `src/lib.rs`:

```
#[test]
fn it_works() {
}
```

Nota el `#[test]`. Este atributo indica que esta es una función de prueba. Actualmente no tiene cuerpo. Pero eso es suficiente para pasar! Podemos ejecutar los tests con `cargo test`:

```
$ cargo test
  Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests sumador

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Cargo compilo y ejecuto nuestros tests. Hay dos conjuntos de salida aquí: uno para las pruebas que nosotros escribimos, y otro para los tests de documentación. Hablaremos acerca de estos mas tarde. Por ahora veamos esta linea:

```
test it_works ... ok
```

Nota el `it_works`. Proviene del nombre de nuestra función:

```
fn it_works() {
# }
```

También obtenemos una linea de resumen:

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

Entonces, porque nuestras pruebas vacías pasan? Cualquier prueba que no haga `panic!` pasa, y cualquier prueba que hace `panic!` falla. Hagamos fallar a nuestra prueba:

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` es una macro proporcionada por Rust la cual toma un argumento: si el argumento es `true`, nada pasa. Si el argumento es `false`, `assert!` hace `panic!`. Ejecutemos nuestras pruebas otra vez:

```
$ cargo test
   Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
    thread 'it_works' panicked at 'assertion failed: false', src/lib.rs:3



failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed', /Users/rustbuild/src/rust-buildbot/slave
```

Rust nos indica que nuestra prueba ha fallado:

```
test it_works ... FAILED
```

Y se refleja en la linea de resumen:

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

También obtenemos un valor de retorno diferente a cero:

```
$ echo $?
101
```

Esto es muy útil para integrar `cargo test` con otras herramientas.

Podemos invertir la falla de nuestras pruebas con otro atributo: `should_panic` :

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

Estas pruebas tendrán éxito si hacemos `panic!` y fallaran si se completan. Probemos:

```
$ cargo test
   Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests sumador

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust proporciona otra macro, `assert_eq!`, que compara dos argumentos para verificar igualdad:

```
#[test]
#[should_panic]
fn it_works() {
    assert_eq!("Hola", "mundo");
}
```

Esta prueba pasa o falla? Debido a la presencia del atributo `should_panic`, pasa:

```
$ cargo test
   Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests sumador

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Las pruebas `should_panic` pueden ser frágiles, es difícil garantizar que la prueba no falló por una razón inesperada. Para ayudar en esto, un parámetro opcional `expected` puede ser agregado a el atributo `should_panic`. La prueba se asegurara que el mensaje de error contenga el mensaje proporcionado. Una versión mas segura de la prueba seria:

```
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hola", "mundo");
}
```

Eso fue todo para lo básico! Escribamos una prueba 'real':

```
pub fn suma_dos(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, suma_dos(2));
}
```

Este es un uso muy común de `assert_eq!` : llamar alguna función con algunos argumentos conocidos y comparar la salida de dicha llamada con la salida esperada.

El modulo `tests`

Hay una forma en la cual nuestro ejemplo no es idiomático: le falta el modulo `tests` . La manera idiomática de escribir nuestro ejemplo luce así:

```
pub fn suma_dos(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::suma_dos;

    #[test]
    fn it_works() {
        assert_eq!(4, suma_dos(2));
    }
}
```

Hay unos cuantos cambios acá. El primero es la inclusion de un `mod tests` con un atributo `cfg` . El modulo nos permite agrupar todas nuestras pruebas, y también nos permite definir funciones de soporte de ser necesario, todo eso no forma parte de nuestro crate. El atributo

`cfg` solo compila nuestro código de pruebas si estuviéramos intentando correr las pruebas. Esto puede ahorrar tiempo de compilación, también se asegura que nuestras pruebas queden completamente excluidas de una compilación normal.

El segundo cambio es la declaración `use` . Debido a que estamos en un modulo interno, necesitamos hace disponible a nuestra prueba dentro de nuestro ámbito actual. Esto puede ser molesto si posees un modulo grande, y es por ello que es común el uso de la facilidad `glob` . Cambiemos nuestro `src/lib.rs` para que haga uso de ello:

```
pub fn suma_dos(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, suma_dos(2));
    }
}
```

Nota la linea `use` diferente. Ahora ejecutamos nuestras pruebas:

```
$ cargo test
   Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests sumador

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Funciona!

La convención actual es usar el modulo `tests` para contener tus pruebas de "estilo-unitario". Cualquier cosa que solo pruebe un pequeño pedazo de funcionalidad va aquí. Pero que acerca de las pruebas "estilo-integracion"? Para ellas, tenemos el directorio `tests` .

El directorio `tests`

Para escribir una prueba de integración, creemos un directorio `tests` y coloquemos un archivo `tests/lib.rs` dentro, con el siguiente de contenido:

```
extern crate sumador;

#[test]
fn it_works() {
    assert_eq!(4, sumador::suma_dos(2));
}
```

Luce similar a nuestras pruebas anteriores, pero ligeramente diferente. Ahora tenemos un `extern crate sumador` al principio. Esto es debido a que las pruebas en el directorio son un crate separado, entonces debemos importar nuestra biblioteca. Esto es también el porque `tests` es un lugar idóneo pera escribir tests de integración: estas pruebas usan la biblioteca justo como cualquier otro consumidor lo haría.

Ejecutemoslas:

```
$ cargo test
   Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/lib-f71036151ee98b04

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests sumador

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Ahora tenemos tres secciones: nuestras pruebas anteriores también fueron ejecutadas, junto con la nueva prueba de integración.

Eso fue todo para el directorio `tests`. El modulo `tests` no es necesario aquí, debido a que el modulo completo esta dedicado a pruebas.

Finalmente echemos un vistazo a esa tercera sección: pruebas de documentación.

Pruebas de documentación

Nada es mejor que documentación con ejemplos. Nada es peor que ejemplos que no funcionan, debido a que el código a cambiado desde que la documentación fue escrita. Respecto a esto, Rust soporta la ejecución automática de los ejemplos presentes en tu documentación. He aquí un `src/lib.rs` pulido con ejemplos:

```
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//!
```

//! assert_eq!(4, adder::add_two(2)); //! ```

/// This function adds two to its argument. /// /// # Examples /// /// `/// use adder::add_two; /// /// assert_eq!(4, add_two(2)); ///` pub fn add_two(a: i32) -> i32 { a + 2 }

[cfg(test)]

mod tests { use super::*;

```
#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

}

```
Nota la documentación a nivel de modulo con `//!` y la documentación a nivel de función c

Ejecutemos las pruebas nuevamente:

```bash
$ cargo test
  Compiling sumador v0.1.0 (file:///Users/goyox86/Code/rust/sumador)
     Running target/debug/lib-f71036151ee98b04

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/debug/sumador-ba17f4f6708ca3b9

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests sumador

running 2 tests
test _0 ... ok
test suma_dos_0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Ahora tenemos los tres tipos de pruebas corriendo! Nota los nombres de las pruebas de documentación: el `_0` es generado para la prueba del modulo, y `suma_dos_0` para la prueba de función. Estos números se auto incrementaran con nombres como `suma_dos_1` a medida que mas ejemplos son agregados.
```

% Compilación Condicional

Rust posee un atributo especial, `#[cfg]`, que te permite compilar código basado en una opción proporcionada al compilador. Tiene dos formas:

```
#[cfg(foo)]
# fn foo() {}

#[cfg(bar = "baz")]
# fn bar() {}
```

También posee algunos helpers:

```
#[cfg(any(unix, windows))]
# fn foo() {}

#[cfg(all(unix, target_pointer_width = "32"))]
# fn bar() {}

#[cfg(not(foo))]
# fn not_foo() {}
```

Los cuales pueden ser anidados de manera arbitraria:

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
# fn foo() {}
```

Para activar o desactivar estos switches, si estas usando Cargo, se configuran en la sección `[features]` de tu `Cargo.toml`

```
[features]
# No features por defecto
default = []

# El feature “secure-password” depende en el paquete bcrypt.
secure-password = ["bcrypt"]
```

Cuando hacemos esto, Cargo pasa una opción a `rustc`:

```
--cfg feature="${feature_name}"
```

La suma de esas opciones `cfg` determinara cuales son activadas, y en consecuencia, cual código sera compilado. Tomemos este código:

```
#[cfg(feature = "foo")]
mod foo {
}
```

Si lo compilamos con `cargo build --features "foo"`, Cargo enviara la opción `--cfg feature="foo"` a `rustc`, y la salida tendrára el `mod foo`. Si compilamos con un `cargo build` normal, ninguna opción extra sera proporcionada, y debido a esto, ningún modulo `foo` existirá.

cfg_attr

También puedes configurar otro atributo basado en una variable `cfg` con `cfg_attr`:

```
#[cfg_attr(a, b)]
# fn foo() {}
```

Sera lo mismo que `#[b]` si `a` es configurado por un atributo `cfg`, y nada de cualquier otra manera.

cfg!

La extension de sintaxis te permite también usar este tipo de opciones en cualquier otro punto de tu código:

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

Estos serán reemplazado por `true` o `false` en tiempo de compilación, dependiendo en las opciones de la configuración.

% Documentación

La documentación es una parte importante de cualquier proyecto de software y un ciudadano de primera clase en Rust. Hablemos acerca de las herramientas que Rust te proporciona para documentar tus proyectos.

Acerca de `rustdoc`

La distribución de Rust incluye una herramienta, `rustdoc`, encargada de generar la documentación. `rustdoc` es también usada por Cargo a través de `cargo doc`.

La documentación puede ser generada de dos formas: desde el código fuente, o desde archivos Markdown.

Documentando código fuente

La principal forma de documentar un proyecto Rust es a través de la anotación del código fuente. Para este propósito, puedes usar comentarios de documentación:

```
/// Construye un nuevo `Rc`.
///
/// # Examples
///
///
```

/// use std::rc::Rc; /// /// let cinco = Rc::new(5); /// ``` pub fn new(value: T) -> Rc { // la implementación va aqui }

```
El código anterior genera documentación que luce como [esta][rc-new](ingles). He dejado l

Los comentarios de documentación están escritos en formato Markdown.

Rust mantiene un registro de dichos comentarios, registro que usa al momento de generar l

```rust
/// El tipo `Option`. Vea [la documentación a nivel de modulo](../) para mas información.
enum Option<T> {
    /// Ningún valor
    None
    /// Algún valor `T`
    Some(T),
}
```
```

Lo anterior funciona, pero esto, no:

```
/// El tipo `Option`. Vea [la documentación a nivel de modulo](../) para mas información.
enum Option {
    None, ///  Ningún valor
    Some(T), /// Algún valor `T`
}
```

Obtendrás un error:

```
hola.rs:4:1: 4:2 error: expected ident, found `}`
hola.rs:4 }
          ^
```

Este desafortunado error es correcto: los comentarios de documentación aplican solo a lo que este después de ellos, y no hay nada después del ultimo comentario.

Escribiendo comentarios de documentación

De cualquier modo, cubramos cada parte de este comentario en detalle:

```
/// Construye un nuevo `Rc<T>`.
# fn foo() {}
```

La primera linea de un comentario de documentación debe ser un resumen corto de sus funcionalidad. Una oración. Solo lo básico. De alto nivel.

```
///
/// Otros detalles acerca de la construcción de `Rc<T>`s, quizás describiendo semántica
/// complicada, tal vez opciones adicionales, cualquier cosa extra.
///
# fn foo() {}
```

Nuestro ejemplo original solo tenia una linea de resumen, pero si hubiésemos tenido mas cosas que decir, pudimos haber agregado mas explicación en un párrafo nuevo.

Secciones especiales

```
/// # Examples
# fn foo() {}
```

A continuación están las secciones especiales. Estas son indicadas con una cabecera, `#`. Hay tres tipos de cabecera que se usan comúnmente. Estos no son sintaxis especial, solo convención, por ahora.

```
/// # Panics
# fn foo() {}
```

Malos e irrecuperables usos de una función (e.j. Errores de programación) en Rust son usualmente indicados por pánicos (panics), los cuales matan el hilo actual como mínimo. Si tu función posee un contrato no trivial como este, que es detectado/impuesto por pánicos, documentarlo es muy importante.

```
/// # Failures
# fn foo() {}
```

Si tu función o método retorna un `Result<T, E>`, entonces describir las condiciones bajo las cuales retorna `Err(E)` es algo bueno por hacer. Esto es ligeramente menos importante que `Panics`, a consecuencia de que es codificado en el sistema de tipos, pero es aun, algo que se recomienda hacer.

```
/// # Safety
# fn foo() {}
```

Si tu función es `unsafe` (insegura), deberías explicar cuales son las invariantes que deben ser mantenidas por el llamador.

```
/// # Examples
///
///
```

/// use std::rc::Rc; /// /// let cinco = Rc::new(5); /// ```

fn foo() {}

```
Tercero, `Examples`, incluye uno o mas ejemplos del uso de tu función o método, y tus usu


```rust
/// # Examples
///
/// Patrones `&str` simples:
///
///
```

/// let v: Vec<&str> = "Mary tenia un corderito".split(' ').collect(); /// assert_eq!(v, vec!["Mary", "tenia", "un", "corderito"]); /// `/// /// Patrones mas complejos con lambdas: /// ///` /// let v: Vec<&str> = "abc1def2ghi".split(|c: char| c.is_numeric()).collect(); /// assert_eq!(v, vec!["abc", "def", "ghi"]); /// ```

fn foo() {}

```
Discutamos los detalles de esos bloques de código.

#### Anotaciones de bloques de código

Para escribir alguna código Rust en un comentario, usa los graves triples:

```rust
///
```

/// println!("Hola, mundo"); /// ```

fn foo() {}

```
Si quieres código que no sea Rust, puedes agregar una anotación:

```rust
/// ```c
/// printf("Hola, mundo\n");
///
```
```

fn foo() {}

```
La sintaxis de esta sección sera resaltada de acuerdo al lenguaje que estés mostrando. Si

Acá, es importante elegir la anotación correcta, debido a que `rustdoc` la usa de una man

## Documentación como pruebas

Discutamos nuestra documentación de ejemplo:

```rust
///
```

/// println!("Hola, mundo"); /// ```

fn foo() {}

```
Notaras que no necesitas una `fn main()` o algo mas. `rustdoc` agregara un main() automát

```rust
///
```

/// use std::rc::Rc; /// /// let cinco = Rc::new(5); /// ```

fn foo() {}

```
Se convertirá en la prueba:

```rust
fn main() {
    use std::rc::Rc;
    let cinco = Rc::new(5);
}
```

He aquí el algoritmo completo que `rustdoc` usa para post-procesar los ejemplos:

1. Cualquier atributo `#![foo]` sobrante es dejado intacto como atributo del crate.
```

2. Algunos atributos comunes son insertados, incluyendo `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, y `dead_code`. Ejemplos pequeños ocasionalmente disparan estos lints.
3. Si el ejemplo no contiene `extern crate`, entonces el `extern crate <micrate>;` es insertado.
4. Finalmente, si el ejemplo no contiene `fn main`, el texto es envuelto en `fn main() { tu_codigo }`

Algunas veces, todo esto no es suficiente. Por ejemplo, todos estos ejemplos de código con `///` de los que hemos estado hablando? El texto plano:

```
/// Algo de documentación.
# fn foo() {}
```

Luce diferente a la salida:

```
/// Algo de documentación.
# fn foo() {}
```

Si, es correcto: puedes agregar lineas que comiencen con `#`, y estas serán eliminadas de la salida, pero serán usadas en la compilación de tu código. Puedes usar esto como ventaja. En este caso, los comentarios de documentación necesitan aplicar a algún tipo de función, entonces si quiero mostrar solo un comentario de documentación, necesito agregar una pequeña definición de función debajo. Al mismo tiempo, esta allí solo para satisfacer al compilador, de manera tal que esconderla hace el ejemplo mas limpio. Puedes usar esta técnica para explicar ejemplos mas largos en detalle, preservando aun la capacidad de tu documentación para ser probada. Por ejemplo, este código:

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

He aquí una explicación, renderizada:

Primero, asignamos a `x` el valor de cinco:

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

A continuación, asignamos seis a `y`:

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

Finalmente, imprimimos la suma de `x` y `y`:

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```

He aquí la misma explicación, en texto plano:

> Primero, asignamos a `x` el valor de cinco:
>
> ```
> let x = 5;
> # let y = 6;
> # println!("{}", x + y);
> ```
>
> A continuación, asignamos seis a `y`:
>
> ```
> # let x = 5;
> let y = 6;
> # println!("{}", x + y);
> ```
>
> Finalmente, imprimimos la suma de `x` y `y`:
>
> ```
> # let x = 5;
> # let y = 6;
> println!("{}", x + y);
> ```

Al repetir todas las partes del ejemplo, puedes asegurarte que tu ejemplo aun compila, mostrando solo las partes relevantes a tu explicación.

Documentando macros

He aquí un ejemplo de la documentación a una macro:

```
/// Panic con un mensaje proporcionado a menos que la expression sea evaluada a true.
///
/// # Examples
///
///
```

/// # #[macro_use] extern crate foo; /// # fn main() { /// panic_unless!(1 + 1 == 2, “Las mathematicas estan rotas.”); /// # } /// `/// ///` should_panic /// # #[macro_use] extern crate foo; /// # fn main() { /// panic_unless!(true == false, “Yo estoy roto.”); /// # } /// ```

[macro_export]

macro_rules! panic_unless { ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic! ($($rest),+); } }); }

fn main() {}

```
Notaras tres cosas: necesitamos agregar nuestro propia linea `extern crate`, de tal maner

### Ejecutando pruebas de documentación

Para correr las pruebas puedes:

```bash
$ rustdoc --test ruta/a/mi/crate/root.rs
# ó
$ cargo test
```

Correcto, `cargo test` prueba la documentación embebida también. Sin embargo, `cargo test`, no probara crates binarios, solo bibliotecas. Esto debido a la forma en la que `rustdoc` funciona: enlaza con la biblioteca a ser probada, pero en el caso de un binario, no hay nada a lo cual enlazar.

Hay unas pocas anotaciones mas que son útiles para ayudar a `rustdoc` a hacer la cosa correcta cuando pruebas tu código:

```
/// ```ignore
/// fn foo() {
///
```

fn foo() {}
```

```
La directiva `ignore` le dice a Rust que ignore el codigo. Esta es la forma que casi nunc

```rust
/// ```should_panic
/// assert!(false);
///
```

fn foo() {}

```
`should_panic` le dice a `rustdoc` que el código debe compilar correctamente, pero sin la

```rust
/// ```no_run
/// loop {
///     println!("Hola, mundo");
/// }
///
```

fn foo() {}

```
El atributo `no_run` compilara tu código, pero no lo ejecutara. Esto es importante para e

### Documentando módulos

Rust posee otro tipo de comentario de documentación, `//!`. Este comentario no documenta

```rust
mod foo {
    //! Esta es documentation para el modulo `foo`.
    //!
    //! # Examples

    // ...
}
```

Es aquí en donde veras `//!` usado mas a menudo: para documentación de módulos. Si tienes un modulo en `foo.rs`, frecuentemente al a abrir su código veras esto:
```

```
//! Un modulo para usar `foo`s.
//!
//! El modulo `foo` contiene un monton de funcionalidad bla bla bla
```

Estilo de comentarios de documentación

Echa un vistazo a el RFC 505 para un listado completo de convenciones acerca del estilo y formato de la documentación (ingles)

Otra documentación

Todo este comportamiento funciona en archivos no Rust también. Debido a que los comentarios son escritos en Markdown, frecuentemente son archivos `.md`.

Cuando escribes documentación en archivos Markdown, no necesitas prefijar la documentación con comentarios. Por ejemplo:

```
/// # Examples
///
///
```

/// use std::rc::Rc; /// /// let cinco = Rc::new(5); /// ```

fn foo() {}

```
es solo

~~~markdown
# Examples
```

use std::rc::Rc;

let cinco = Rc::new(5);

```
~~~

cuando esta en un archivo Markdown. Solo hay un detalle, los archivos markdown necesitan


```markdown
% Titulo

Esta es la documentación de ejemplo
```

Esta linea `%` deber estar ubicada en la primera linea del archivo.

atributos `doc`

A un nivel mas profundo, los comentarios de documentación son otra forma de escribir atributos de documentación:

```
/// this
# fn foo() {}

#[doc="this"]
# fn bar() {}
```

son lo mismo que estos:

```
//! this

#![doc="/// this"]
```

No veras frecuentemente este atributo siendo usado para escribir documentación, pero puede ser util cuando se esten cambiando ciertas opciones, o escribiendo una macro.

Re-exports

`rustdoc` mostrara la documentación para un re-export publico en ambos lugares:

```
extern crate foo;

pub use foo::bar;
```

Lo anterior creara documentación para bar dentro de la documentación para el crate `foo`, así como la documentación para tu crate. Sera la misma documentación en ambos lugares.
```

Este comportamiento puede ser suprimido con `no_inline`:

```
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

Controlando HTML

Puedes controlar algunos aspectos de el HTML que `rustdoc` genera a través de la versión `#![doc]` del atributo:

```
#![doc(html_logo_url = "http://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "http://www.rust-lang.org/favicon.ico",
       html_root_url = "http://doc.rust-lang.org/")]
```

Esto configura unas pocas opciones, con un logo, favicon, y URL raíz.

Opciones de generación

`rustdoc` también contiene unas pocas opciones en la linea de comandos, para mas personalización:

- `--html-in-header FILE`: incluye el contenido de FILE al final de la sección `<head>...</head>`.
- `--html-before-content FILE`: incluye el contenido de FILE después de `<body>`, antes del contenido renderizado (incluyendo la barra de búsqueda).
- `--html-after-content FILE`: incluye el contenido de FILE después de todo el contenido renderizado.

Nota de seguridad

El Markdown en los comentarios de documentación es puesto sin procesar en la pagina final. Se cuidadoso con HTML literal:

```
/// <script>alert(document.cookie)</script>
# fn foo() {}
```

% Iteradores

Hablemos de ciclos.

Recuerdas el ciclo `for` de Rust? He aqui un ejemplo:

```
for x in 0..10 {
    println!("{}", x);
}
```

Ahora que sabes mas Rust, podemos hablar en detalle acerca de como este código funciona. Los rangos (el `0..10`) son iteradores. Un iterador es algo en lo que podemos llamar el método `.next()` repetitivamente, y el iterador nos proporciona una secuencia de elementos.

Por ejemplo:

```
let mut rango = 0..10;

loop {
    match rango.next() {
        Some(x) => {
            println!("{}", x);
        },
        None => { break }
    }
}
```

Creamos un enlace (una variable) a el rango, nuestro iterador. Luego iteramos mediante el ciclo `loop`, con un `match` interno. Dicho `match` usa el resultado de `rango.next()`, que nos proporciona una referencia a el siguiente valor en el iterador. `next` retorna un `Option<i32>`, en este caso, que será `Some(i32)` cuando tenemos un valor y `None` cuando nos quedemos sin valores. Si obtenemos `Some(i32)`, lo imprimimos, y si obtenemos `None`, rompemos el ciclo, saliendo de el a través de `break`.

Este ejemplo de código es básicamente el mismo que nuestra version de un ciclo for `for`. El ciclo `for` es solo una forma practica de escribir una construcción `loop` / `match` / `break`.

Sin embargo, los ciclos `for` no son la única cosa que usa iteradores. Escribir tu propio iterador implica implementar el trait `Iterator`. Si bien hacerlo esta fuera del ámbito de esta guía, Rust provee a un numero de iteradores útiles para llevar a cabo diversas tareas. Antes de hablar de eso, debemos hablar acerca de un anti-patron. Dicho anti-patron es usar rangos de la manera expuesta anteriormente.

Si, acabamos de hablar acerca de cuan cool son los rangos. Pero son también muy primitivos. Por ejemplo, si necesitamos iterar a través del contenido de un vector, podríamos estar tentados a escribir algo como esto:

```
let nums = vec![1, 2, 3];

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

Esto no es estrictamente peor que usar un iterador. Se puede iterar en vectores directamente, escribe esto:

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", num);
}
```

Hay dos razones para hacerlo de esta manera. Primero, expresa lo que queremos de manera mas directa. Iteramos a través del vector completo, en vez de iterar a través de indices para luego indexar el vector. Segundo, esta version es mas eficiente: en la primera version tendremos chequeos de limites extra debido a que usa indexado, `nums[i]`. En el segundo ejemplo y debido a que con el iterador cedemos una referencia a cada elemento a la vez, ho hay chequeo de limites. Esto es muy común en iteradores: podemos ignorar chequeos de limites innecesarios, sabiendo al mismo tiempo que estamos seguros.

Hay otro detalle aquí que no esta el 100% claro debido a el funcionamiento `println!`. `num` es de tipo `&i32`. Una referencia a un `i32`, no un `i32`. `println!` maneja el dereferenciamiento por nosotros, es por ello que no lo vemos. Este código también es correcto:

```
let nums = vec![1, 2, 3];

for num in &nums {
    println!("{}", *num);
}
```

Ahora estamos dereferenciando a `num` de forma explicita. Porque `&nums` nos da referencias? Primeramente, porque lo solicitamos de manera explicita con `&`. Segundo, si nos diera la data en si misma, tendríamos que ser dueños de ella, lo cual implicaría la

creación de una copia de la data para después darnos esa copia. Con referencias, solo estamos haciendo un préstamo ('borrowing') de una referencia a la data, y por ello solo pasamos una referencia, sin necesidad de transferir la pertenencia.

Entonces, ahora que hemos establecido que los rangos a veces no son lo que queremos, hablemos de lo queremos.

Hay tres amplias clases de cosas que son relevantes: iteradores, *adaptadores de iteradores* y *consumidores*. He aquí algunas definiciones:

- *iteradores* proporcionan una sequencia de valores.
- *adaptadores de iteradores* operan en un iterador, produciendo un nuevo iterador con una secuencia diferente de salida.
- *consumidores* operan en un iterador, produciendo un conjunto final de valores.

Hablemos primeramente acerca de los consumidores, debido a que ya hemos visto un iterador, los rangos.

Consumidores

Un *consumidor* opera en un iterador, retornando algún tipo de valor o valores. El consumidor mas común es `collect()`. Este código no compila, pero muestra la intención:

```
let uno_hasta_cien = (1..101).collect();
```

Como puedes ver, llamamos `collect()` en el iterador. `collect()` toma tantos valores como el iterador le proporcione, retornando una colección de resultados. Entonces, porque este código no compilara? Rust no puede determinar que tipo de cosas quieres recolectar, y es por ello necesitas hacerle saber. Esta es la version que compila:

```
let uno_hasta_cien = (1..101).collect::<Vec<i32>>();
```

Si recuerdas, la sintaxis `::<>` te permite dar una indicio acerca del tipo, en nuestro caso estamos diciendo que queremos un vector de enteros. No siempre es necesario usar el tipo completo. `_` te permitirá dar un indicio parcial acerca del tipo:

```
let uno_hasta_cien = (1..101).collect::<Vec<_>>();
```

Esto dice "Recolecta en un `Vec<T>`, por favor, pero infiere que es `T` por mi.". `_` es por esta razón llamado algunas veces "marcador de posición de tipo".

`collect()` es el consumidor mas común, pero hay otros. `find()` es uno de ellos:

```
let mayores_a_cuarenta_y_dos = (0..100)
                               .find(|x| *x > 42);

match mayores_a_cuarenta_y_dos {
    Some(_) => println!("Tenemos algunos números!"),
    None => println!("No se encontraron números :("),
}
```

`find` recibe un closure, y trabaja en una referencia a cada elemento de un iterador. Dicho closure retorna `true` si el elemento es el que estamos buscando y `false` de lo contrario. Debido a que podríamos no encontrar un elemento que satisfaga nuestro criterio, `find` retorna un `Option` en lugar de un elemento.

Otro consumidor importante es `fold`. Luce asi:

```
let suma = (1..4).fold(0, |suma, x| suma + x);
```

`fold(base, |acumulador, elemento| ...)`. Toma dos argumentos: el primero es un elemento llamado *base*. El segundo es un closure que a su vez toma dos argumentos: el primero es llamado el *acumulador*, y el segundo es un *elemento*. En cada iteración, el closure es llamado, y el resultado es usado como el valor del acumulador en la siguiente iteración. En la primera iteración, la base es el valor del acumulador.

Bien, eso es un poco confuso. Examinemos los valores de todas las cosas en este iterador:

base	acumulador	elemento	resultado del closure
0	0	1	1
0	1	2	3
0	3	3	6

Hemos llamado a `fold()` con estos argumentos:

```
# (1..4)
.fold(0, |suma, x| suma + x);
```

Entonces, `0` es nuestra base, `suma` es nuestro acumulador, y `x` es nuestro elemento. En la primera iteración, asignamos `sum` a `0` y `x` es el primer elemento de nuestro rango, `1`. Después sumamos `sum` y `x` lo que nos da `0 + 1 = 1`. En la segunda iteración, ese valor se convierte en el valor de nuestro acumulador, `sum`, y el elemento es el segundo elemento del rango, `2`. `1 + 2 = 3` y de igual manera se convierte en el valor del

acumulador para la ultima iteración. En esa iteración, `x` es el ultimo elemento, `3`, y `3 + 3 = 6`, resultado final para nuestro `suma`. `1 + 2 + 3 = 6`, ese es el resultado que obtenemos.

Whew. `fold` puede ser un poco extraño a primera vista, pero una vez hace click, puedes usarlo en todos lados. Cada vez que tengas una lista de cosas, y necesites un único resultado, `fold` es apropiado.

Los consumidores son importantes debido a una propiedad adicional de los iteradores de la que no hemos hablado todavia: pereza (laziness). Hablemos mas acerca de los iteradores y veras porque los consumidores son importantes.

Iteradores

Como hemos dicho antes, un iterador es algo en lo que podemos llamar el método `.next()` repetidamente, y este nos devuelve una secuencia de elementos. Debido a que necesitamos llamar a el método, los iteradores pueden ser *perezosos* y no generar todos los valores por adelantado. Este código, por ejemplo, no genera los números `1-99`. En su lugar crea un valor que representa la secuencia:

```
let nums = 1..100;
```

Debido a que no hicimos nada con el rango, este no genero la secuencia. Agreguemos un consumidor:

```
let nums = (1..100).collect::<Vec<i32>>();
```

Ahora `collect()` requerirá que el rango provea algunos números, y en consecuencia tendra que llevar a cabo la labor de generar la secuencia.

Los rangos son una de las dos formas básicas de iteradores que veras. La otra es `iter()`. `iter()` puede transformar un vector en un iterador simple que proporciona un elemento a la vez:

```
let nums = vec![1, 2, 3];

for num in nums.iter() {
   println!("{}", num);
}
```

Estos dos iteradores básicos deberian ser de utilidad. Existen iteradores mas avanzados, incluyendo aquellos que son infinitos.

Suficiente acerca de iteradores, los adaptadores de iteradores son el ultimo concepto relacionado a iteradores al que debemos hacer mención. Hagamoslo!

Adaptadores de iterador

Los *adaptadores de iteradores* toman un iterador y lo modifican de alguna manera, produciendo uno nuevo. El mas simple es llamado `map`:

```
(1..100).map(|x| x + 1);
```

`map` es llamado en otro iterador, `map` produce un iterador nuevo en el que cada referencia a un elemento posee el closure que se ha proporcionado como argumento. El código anterior nos dará los números `2-100`, Bueno, casi! Si compilas el ejemplo, obtendrás una advertencia:

```
warning: unused result which must be used: iterator adaptors are lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on by default
(1..100).map(|x| x + 1);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

La pereza ataca de nuevo! Ese closure nunca se ejecutara. Este ejemplo no imprime ningún numero:

```
(1..100).map(|x| println!("{}", x));
```

Si estas intentanto ejecutar un closure en un iterador para obtener sus efectos colaterales (side-effects) usa un `for`.

```
for i in (1..).take(5) {
    println!("{}", i);
}
```

Esto imprimira:

```
1
2
3
4
5
```

`filter()` es un adaptador que toma un closure como argumento. Dicho closure retorna `true` o `false`. El nuevo iterador que `filter()` produce solo elementos para los que el closure retorna `true`:

```
for i in (1..100).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

Esto imprimirá todos los números pares entre uno y cien. (Nota que debido a que `filter` no consume los elementos que están siendo iterados, a este se le pasa una referencia a cada elemento, debido a ello, el predicado usa el patron `&x` para extraer el entero.)

```
(1..)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<i32>>();
```

Lo anterior te un vector conteniendo `6`, `12`, `18`, `24`, y `30`.

Esta es una pequeña muestra de las cosas en las cuales los iteradores, adaptadores de iteradores, y consumidores pueden ayudarte. Existe una variedad de iteradores realmente útiles, junto al hecho que puedes escribir tus propios. Los iteradores proporcionan una manera segura y eficiente de manipular todo tipo de listas. Son un poco inusuales a primera vista, pero si juegas un poco con ellos, quedaras enganchado. Para una lista de los diferentes iteradores y consumidores echa un vistazo a la documentacion del modulo iterator (ingles).

% Concurrencia

La concurrencia y el paralelismo son dos tópicos increíblemente importantes en las ciencias de la computación, también son un tópico caliente en la industria hoy en día. Las computadoras cada vez poseen mas y mas núcleos, aun así, todavía algunos desarrolladores no están preparados para utilizarnos completamente.

La seguridad en el manejo de memoria de Rust también aplica a su historia de concurrencia. Incluso los programas concurrentes deben ser seguros en el manejo de memoria, sin condiciones de carrera. El sistema de tipos de Rust esta a la altura del desafío, y te dota de poderosas vías para razonar acerca de código concurrente en tiempo de compilación.

Antes de que hablemos de las características de concurrencia que vienen con Rust, es importante entender algo: Rust es de bajo nivel, suficientemente bajo al punto que todas estas facilidades están implementadas en la biblioteca estándar. Esto significa que si no te gusta algún aspecto de la manera en la que Rust maneja la concurrencia, puedes implementar una forma alternativa de hacerlo. mio es un vivo ejemplo de este principio en acción.

Bases: `Send` y `Sync`

La concurrencia es algo sobre lo que es difícil razonar. En Rust tenemos un poderoso, sistema de tipos estático que nos ayuda a razonar acerca de nuestro código. Como tal, Rust nos provee de dos traits para ayudarnos a darle sentido a código que pueda posiblemente ser concurrente.

`Send`

El primer trait del cual hablaremos es `Send` (ingles). Cuando un tipo `T` implementa `Send`, le indica al compilador que algo de este tipo puede transferir la pertenencia entre hilos de forma segura.

Esto es importante para hacer cumplir ciertas restricciones. Por ejemplo si tenemos un canal, conectando dos hilos, deberíamos querer transferir algunos datos a el otro hilo través del canal. Por lo tanto, nos aseguramos de que `Send` haya sido implementado para ese tipo.

De manera opuesta, si estamos envolviendo una biblioteca con FFI que no es threadsafe, no deberíamos querer implementar `Send`, de manera tal que el compilador nos ayude a asegurarnos que esta no pueda abandonar el hilo actual.

Sync

El segundo de estos traits es llamado `Sync` (ingles). Cuando un tipo `T` implementa `Sync`, le indica al el compilador que algo de este tipo no tiene posibilidad de introducir inseguridad en memoria cuando es usado de manera concurrente por multiples hilos de ejecución.

Por ejemplo, compartir data inmutable con una cuenta de referencias atómica es threadsafe. Rust provee dicho tipo, `Arc<T>`, el cual implementa `Sync`, y es por ello que es seguro de compartir entre hilos.

Estos dos traits te permiten usar el sistema de tipos para hacer garantías fuertes acerca de las propiedades de tu código bajo concurrencia. En primer lugar, antes de demostrar porque, necesitamos aprender como crear un programa concurrente en Rust!

Hilos

La biblioteca estándar de Rust provee una biblioteca para el manejo de hilos, que te permite ejecutar código rust de forma paralela. He aqui un ejemplo basico del uso de `std::thread`:

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hola desde un hilo!");
    });
}
```

El método `thread::spawn()` acepta un closure como argumento, closure que es ejecutado en un nuevo hilo. `thread::spawn()` retorna un handle a el nuevo hilo, que puede ser usado para esperar a que el hilo finalice y luego extraer su resultado:

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        "Hola desde un hilo!"
    });

    println!("{}", handle.join().unwrap());
}
```

Muchos lenguajes poseen la habilidad de ejecutar hilos, pero es salvajemente inseguro. Existen libros enteros acerca de como prevenir los errores que ocurren como consecuencia de compartir estado mutable. Rust ayuda con su sistema de tipos, previniendo condiciones de carrera en tiempo de compilación. Hablemos acerca de como efectivamente puedes compartir cosas entre hilos.

Estado Mutable Compartido Seguro

Debido a el sistema de tipos de Rust, tenemos un concepto que suena como una mentira: "estado mutable compartido seguro". Muchos programadores concuerdan en que el estado mutable compartido es muy, muy malo.

Alguien dijo alguna vez:

> El estado mutable compartido es la raíz de toda maldad. La mayoría de los lenguajes intentan lidiar con este problema a través de la parte 'mutable', pero Rust lo enfrenta resolviendo la parte 'compartida'.

La misma pertenencia que previene el uso incorrecto de apuntadores también ayuda a eliminar las condiciones de carrera, uno de los peores bugs relacionados con concurrencia.

Como ejemplo, un programa Rust que tendría una condición de carrera en muchos lenguajes. En Rust no compilaría:

```
use std::thread;

fn main() {
    let mut data = vec![1u32, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

Lo anterior produce un error:

```
8:17 error: capture of moved value: `data`
        data[i] += 1;
        ^~~~
```

En este caso, sabemos que nuestro código *deberia* ser seguro, pero Rust no esta seguro de ello. En realidad no es seguro, si tuviéramos una referencia a `data` en cada hilo, y el hilo toma pertenencia de la referencia, tendríamos tres dueños! Eso esta mal. Podemos arreglar esto a través del uso del tipo `Arc<T>`, un apuntador con conteo atómico de referencias. La parte 'atómico' significa que es seguro compartirlo entre hilos.

`Arc<T>` asume una propiedad mas acerca de su contenido para asegurarse que es seguro compartirlo entre hilos: asume que su contenido es `Sync`. Pero en nuestro caso, queremos mutar el valor. Necesitamos un tipo que pueda asegurarse que solo una persona a la vez pueda mutar lo que este dentro. Para eso, podemos usar el tipo `Mutex<T>`. He aquí la segunda version de nuestro código. Aun no funciona, pero por una razón diferente:

```
use std::thread;
use std::sync::Mutex;

fn main() {
    let mut data = Mutex::new(vec![1u32, 2, 3]);

    for i in 0..3 {
        let data = data.lock().unwrap();
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

He aqui el error:

```
:9:9: 9:22 error: the trait `core::marker::Send` is not implemented for the type `std::sy
:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
:9:9: 9:22 note: `std::sync::mutex::MutexGuard<'_, collections::vec::Vec>` cannot be sent
:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
```

Lo ves, `Mutex` posee un metodo `lock` que posee esta firma:

```
fn lock(&self) -> LockResult>
```

Debido a que `Send` no esta implementado para `MutexGuard<T>`, no podemos transferir el `MutexGuard<T>` entre hilos, lo que se traduce en el error.

Podemos usar `Arc<T>` para corregir el error. He aqui la version funcional:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[i] += 1;
        });
    }

    thread::sleep_ms(50);
}
```

Ahora llamamos `clone()` en nuestro `Arc`, lo cual incrementa el contador interno. Este handle es entonces movido dentro del nuevo hilo. Examinemos el cuerpo del hilo mas de cerca:

```
# use std::sync::{Arc, Mutex};
# use std::thread;
# fn main() {
#     let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));
#     for i in 0..3 {
#         let data = data.clone();
thread::spawn(move || {
    let mut data = data.lock().unwrap();
    data[i] += 1;
});
#     }
#     thread::sleep_ms(50);
# }
```

Primero, llamamos `lock()`, lo cual obtiene el bloqueo de exclusion mutua. Debido a que esta operación puede fallar, este método retorna un `Result<T, E>`, y debido a que esto es solo un ejemplo, hacemos `unwrap()` en el para obtener una referencia a la data. Código real tendría un manejo de errores mas robusto en este lugar. Somos libres de mutarlo, puesto que tenemos el bloqueo.

Por ultimo, mientras que los hilos se ejecutan, esperamos por la culminación de un temporizador corto. Esto no es ideal: pudimos haber escogido un tiempo razonable para esperar pero lo mas probable es que esperemos mas de lo necesario, o no lo suficiente, dependiendo de cuanto tiempo los hilos toman para terminar la computación cuando el programa corre.

Una alternativa mas precisa a el temporizador seria el uso de uno de los mecanismos proporcionados por la biblioteca estándar de Rust para la sincronización entre hilos. Hablemos de ellos: los canales.

Canales

He aquí una version nuestro código que usa canales para la sincronización, en lugar de esperar por un tiempo especifico:

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc;

fn main() {
    let data = Arc::new(Mutex::new(0u32));

    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            tx.send(());
        });
    }

    for _ in 0..10 {
        rx.recv();
    }
}
```

Hacemos uso del método `mpsc::channel()` para construir un canal nuevo. Enviamos (a través de `send`) un simple `()` a través del canal, y luego esperamos por el regreso diez de ellos.

Mientras que este canal esta solo enviando una senal genérica, podemos enviar cualquier data que sea `Send` a través del canal!

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let respuesta = 42u32;

            tx.send(respuesta);
        });
    }

   rx.recv().ok().expect("No se ha podido recibir la respuesta");
}
```

Un `u32` es `Send` porque podemos hacer una copia de el. Entonces creamos un hilo, y le solicitamos que calcule la respuesta, este luego nos envía la respuesta de regreso (usando `send()`) a través del canal.

Panicos

Un `panic!` causara la finalización abrupta (crash) del hilo de ejecución actual. Puedes usar los hilos de Rust como un mecanismo de aislamiento sencillo:

```
use std::thread;

let resultado = thread::spawn(move || {
    panic!("ups!");
}).join();

assert!(resultado.is_err());
```

Nuestro `Thread` nos devuelve un `Result` , el cual nos permite chequear si el hilo ha hecho pánico o no.

% Manejo de Errores

> Los planes mejor establecidos por ratones y hombres a menudo se tuercen. "Tae a Moose", Robert Burns

Algunas veces, las cosas simplemente salen mal. Es importante tener un plan para cuando lo inevitable suceda. Rust posee un soporte rico para el manejo de errores que podrían (seamos honestos: ocurrirán) ocurrir en tus programas.

Existen dos tipos de errores que pueden ocurrir en tus programas: fallas y pánicos. Hablaremos de las diferencias entre los dos, y luego discutiremos como manejar cada uno. Después discutiremos como promover fallas a pánicos.

Falla vs. Pánico

Rust usa dos términos para diferenciar entre las dos formas de error: falla, y pánico. Una *falla* es un error del cual nos podemos recuperar de alguna manera. Un *pánico* es un error irrecuperable.

Que queremos decir con "recuperar"? Bueno, en la mayoría de los casos, la posibilidad de un error es esperada. Por ejemplo, considera la función `parse`:

```
"5".parse();
```

Este método convierte una cadena de caracteres a otro tipo. Pero debido a que es una cadena de caracteres, no se puede estar seguro de que la conversion efectivamente tenga éxito. Por ejemplo, a que debería ser convertido esto?:

```
"hola5mundo".parse();
```

Lo anterior no funciona. Sabemos que el método `parse()` solo tendrá éxito para algunas entradas. Es un comportamiento esperado. Es por ello que llamamos a este error una *falla*.

Por otro lado, algunas veces, hay errores que son inesperados, o de los cuales no nos podemos recuperar. Un ejemplo clásico es un `assert!`:

```
# let x = 5;
assert!(x == 5);
```

Usamos `assert!` para declarar que algo es cierto (true). Si la declaración no es verdad, entonces algo esta muy mal. Suficientemente mal como para no poder continuar la ejecución en el estado actual. Otro ejemplo es el uso de la macro `unreachable!()` :

```
use Evento::NuevoLanzamiento;

enum Evento {
    NuevoLanzamiento,
}

fn probabilidad(_: &Evento) -> f64 {
    // la implementación real seria mas compleja, por supuesto
    0.95
}

fn probabilidad_descriptiva(evento: Evento) -> &'static str {
    match probabilidad(&evento) {
        1.00 => "cierto",
        0.00 => "imposible",
        0.00 ... 0.25 => "muy poco probable",
        0.25 ... 0.50 => "poco probable",
        0.50 ... 0.75 => "probable",
        0.75 ... 1.00 => "muy probable",
    }
}

fn main() {
     println!("{}", probabilidad_descriptiva(NuevoLanzamiento));
}
```

Lo anterior resultara en un error:

```
error: non-exhaustive patterns: `_` not covered [E0004]
```

Si bien sabemos que hemos cubierto todos los casos posibles, Rust no puede saberlo. No sabe cual es la probabilidad entre 0.0 y 1.0. Es por ello que agregamos otro caso:

```
use Evento::NuevoLanzamiento;

enum Evento {
    NuevoLanzamiento,
}

fn probabilidad(_: &Evento) -> f64 {
    // la implementación real seria mas compleja, por supuesto
    0.95
}

fn probabilidad_descriptiva(evento: Evento) -> &'static str {
    match probabilidad(&evento) {
        1.00 => "cierto",
        0.00 => "imposible",
        0.00 ... 0.25 => "muy poco probable",
        0.25 ... 0.50 => "poco probable",
        0.50 ... 0.75 => "probable",
        0.75 ... 1.00 => "muy probable",
        _ => unreachable!()
    }
}

fn main() {
    println!("{}", probabilidad_descriptiva(NuevoLanzamiento));
}
```

Nunca deberíamos alcanzar el caso `_`, debido a esto hacemos uso de la macro para indicarlo. `unreachable!()` produce un tipo diferente de error que `Result`. Rust llama a ese tipo de errores *pánicos*.

Manejando errores con `Option` y `Result`

La manera mas simple de indicar que una función puede fallar es usando el tipo `Option<T>`. Por ejemplo, el método find en las cadenas de caracteres intenta localizar un patrón en la cadena, retorna un `Option`:

```
let s = "foo";

assert_eq!(s.find('f'), Some(0));
assert_eq!(s.find('z'), None);
```

Esto es apropiado para casos simples, pero no nos da mucha información en el caso de una falla. Que tal si quisiéramos saber el *porque* la función falló? Para ello, podemos usar el tipo `Result<T, E>`. Que luce así:

```
enum Result<T, E> {
   Ok(T),
   Err(E)
}
```

Esta enum es proporcionada por Rust, es por ello que no necesitas definirla si deseas hacer uso de ella. La variante `Ok(T)` representa éxito, y la variante `Err(E)` representa una falla. Retornar un `Result` en lugar de un `Option` es recomendable para la mayoría de los casos no triviales:

He aquí un ejemplo del uso de `Result`:

```
#[derive(Debug)]
enum Version { Version1, Version2 }

#[derive(Debug)]
enum ErrorParseo { LongitudCabeceraInvalida, VersionInvalida }

fn parsear_version(cabecera: &[u8]) -> Result<Version, ErrorParseo> {
    if cabecera.len() < 1 {
        return Err(ErrorParseo::LongitudCabeceraInvalida);
    }
    match cabecera[0] {
        1 => Ok(Version::Version1),
        2 => Ok(Version::Version2),
        _ => Err(ErrorParseo::LongitudCabeceraInvalida)
    }
}

fn main() {
    let version = parsear_version(&[1, 2, 3, 4]);
    match version {
    Ok(v) => {
      println!("trabajando con la version: {:?}", v);
    }
    Err(e) => {
      println!("error parseando cebecera: {:?}", e);
    }
    }
}
```

Esta función hace uso de un enum, `ErrorParseo`, para enumerar los errores que pueden ocurrir.

El trait `Debug` es el que nos permite imprimir el valor del enum usando la operación de formato `{:?}`.

Errores no recuperables con `panic!`

En el caso de un error inesperado del cual no se pueda recuperar, la macro `panic!` se utiliza para inducir un pánico. Dicho pánico terminara abruptamente el hilo actual de ejecución proporcionando un mensaje de error:

```
panic!("boom");
```

resulta en

```
thread '
' panicked at 'boom', hello.rs:2
```

cuando lo ejecutas.

Debido a que estas situaciones son relativamente raras, usa los pánicos con moderación.

Promoviendo fallas a pánicos

En ciertas circunstancias, aun sabiendo que una función puede fallar, podríamos querer tratar la falla como un pánico. Por ejemplo, `io::stdin().read_line(&mut buffer)` retorna un `Result<usize>`, cuando hay un error leyendo la linea. Esto nos permite manejar y posiblemente recuperarnos en caso de error.

Si no queremos manejar el error, y en su lugar simplemente abortar el programa, podemos usar el método `unwrap()`:

```
io::stdin().read_line(&mut buffer).unwrap();
```

`unwrap()` hará un pánico (`panic!`) si el `Result` es `Err`. Esto básicamente dice "Dame el valor, y si algo sale mal, simplemente aborta la ejecución". Esto es menos confiable que hacer match en el error y tratar de recuperarnos, pero al mismo tiempo es significativamente mas corto. Algunas veces, la terminación abrupta es apropiada.

Hay una manera que de hacer lo anterior que es un poco mejor que `unwrap()`:

```
let mut bufer = String::new();
let bytes_leidos = io::stdin().read_line(&mut bufer)
                      .ok()
                      .expect("Fallo al leer linea");
```

`ok()` convierte el `Result` en un `Option`, y `expect()` hace lo mismo que `unwrap()`, pero recibe un mensaje como argumento. Este mensaje es pasado a el `panic!` subyacente, proporcionando un mejor mensaje de error.

Usando `try!`

Cuando escribimos código que llama a muchas funciones que retornan el tipo `Result`, el manejo de errores se puede tornar tedioso. La macro `try!` esconde algo de el código repetitivo correspondiente a la propagación de errores en la pila de llamadas.

`try!` reemplaza:

```
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    nombre: String,
    edad: i32,
    grado: i32,
}

fn escribir_info(info: &Info) -> io::Result<()> {
    let mut archivo = File::create("mis_mejores_amigos.txt").unwrap();

    if let Err(e) = writeln!(&mut archivo, "nombre: {}", info.nombre) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut archivo, "edad: {}", info.edad) {
        return Err(e)
    }
    if let Err(e) = writeln!(&mut archivo, "grado: {}", info.rgrado) {
        return Err(e)
    }

    return Ok(());
}
```

Con:

```
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Info {
    nombre: String,
    edad: i32,
    grado: i32,
}

fn escribir_info(info: &Info) -> io::Result<()> {
    let mut archivo = File::create("mis_mejores_amigos.txt").unwrap();

    try!(writeln!(&mut archivo, "nombre: {}", info.nombre));
    try!(writeln!(&mut archivo, "edad: {}", info.edad));
    try!(writeln!(&mut archivo, "grado: {}", info.grado));

    return Ok(());
}
```

Envolver una expresión con `try!` resultara en el valor (`Ok`) exitoso desenvuelto, a menos que el resultado sea `Err` , caso en el cual `Err` es retornado de manera temprana por la función que envuelve al try.

Es importante hacer mención a el hecho de que solo puedes usar `try!` desde una función que retorna un `Result` , lo que se traduce en que no puedes usar `try!` dentro de `main()` , debido a que `main()` no retorna nada.

`try!` hace uso de `From<Error>` (ingles) para determinar que retornar en el caso de error.

% Interfaz de Funciones Foráneas/Externas

Introducción

Esta guía usara la biblioteca de compresión/descompresión snappy como introducción a la escritura de bindings a código externo. Rust actualmente no puede llamar código en una biblioteca C++ de manera directa, pero snappy incluye una interfaz en C (documentada en `snappy-c.h`).

El siguiente es un ejemplo minino de como llamar una función foránea que compilara asumiendo que snappy esta instalada:

```
# #![feature(libc)]
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("longitud maxima de un buffer the 100 bytes comprimido: {}", x);
}
```

El bloque `extern` es una lista de firmas de función en una biblioteca foránea, en este caso con la interfaz binaria de aplicación C (ABI en ingles) de la plataforma. El atributo `#[link(...)]` es usado para instruir al enlazador a enlazar con la biblioteca snappy de modo que los símbolos puedan ser resueltos.

Se asume que las interfaces a funciones foráneas son inseguras, es por ello que las llamadas a ellas deben estar dentro de un bloque `unsafe {}` como una promesa al compilador de que todo lo contenido en el es realmente seguro. Bibliotecas en C a algunas veces exponen interfaces que no son thread-safe, y casi cualquier función que toma un apuntador como argumento no es valida para todas las entradas posibles debido a que el apuntador podría ser un apuntador colgante, y los apuntadores planos quedan por fuera del modelo de memoria segura de Rust.

Cuando se declaran los tipos de los argumentos de una función foránea, el compilador de Rust no puede chequear si la declaración es correcta, a consecuencia de esto, especificarla de manera correcta forma parte de mantener el binding correcto en tiempo de ejecución.

El bloque `extern` puede ser extendido para cubrir la API completa de snappy:

```
# #![feature(libc)]
extern crate libc;
use libc::{c_int, size_t};

#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
# fn main() {}
```

Creando una interfaz segura

La interfaz plana en C necesita ser envuelta con el objetivo de proveer seguridad en el manejo de memoria así como el uso de conceptos de alto nivel tales como vectores. Una biblioteca puede escoger entre exponer la interfaz segura de alto nivel y esconder los detalles inseguros internos.

Envolver las funciones que esperan bufers involucra el uso de el modulo `slice::raw` para la manipulación de vectores Rust como apuntadores a memoria. Los vectores de Rust están garantizados a ser un bloque de memoria contiguo. La longitud es el numero de elementos actualmente contenidos, y la capacidad es el tamaño total de la memoria asignada. La longitud es menor o igual a la capacidad.

```
# #![feature(libc)]
# extern crate libc;
# use libc::{c_int, size_t};
# unsafe fn snappy_validate_compressed_buffer(_: *const u8, _: size_t) -> c_int { 0 }
# fn main() {}
pub fn validar_buffer_comprimido(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

La función envoltorio `validar_buffer_comprimido` hace uso de un bloque `unsafe`, pero hace la garantía de que llamarla es segura para todas las entradas a través de la exclusion del `unsafe` de la firma de la función.

Las funciones `snappy_compress` y `snappy_uncompress` son mas complejas, debido a que un bufer tiene que ser asignado para mantener la salida.

La función `snappy_max_compressed_length` puede ser usada para asignar un vector con la capacidad maxima requerida para almacenar la salida comprimida. El vector entonces puede ser pasado a la función `snappy_compress` como un parámetro de salida. Un parámetro de salida es también pasado para obtener la longitud real después de la compresión para asignar la longitud.

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_compress(a: *const u8, b: size_t, c: *mut u8,
#                           d: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_max_compressed_length(a: size_t) -> size_t { a }
# fn main() {}
pub fn comprimir(orig: &[u8]) -> Vec<u8> {
    unsafe {
        let long_orig = orig.len() as size_t;
        let porig = orig.as_ptr();

        let mut long_dest = snappy_max_compressed_length(long_orig);
        let mut dest = Vec::with_capacity(long_dest as usize);
        let pdest = dest.as_mut_ptr();

        snappy_compress(porig, long_orig, pdest, &mut long_dest);
        dest.set_len(long_dest as usize);
        dest
    }
}
```

La descompresion es similar, debido a que snappy almacena la longitud descomprimida como parte del formato de compresión y `snappy_uncompressed_length` obtendrá el tamaño exacto del bufer requerido.

```
# #![feature(libc)]
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_uncompress(compressed: *const u8,
#                             compressed_length: size_t,
#                             uncompressed: *mut u8,
#                             uncompressed_length: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_uncompressed_length(compressed: *const u8,
#                                      compressed_length: size_t,
#                                      result: *mut size_t) -> c_int { 0 }
# fn main() {}
pub fn descomprimir(orig: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let long_orig = orig.len() as size_t;
        let porig = orig.as_ptr();

        let mut long_dest: size_t = 0;
        snappy_uncompressed_length(porig, long_orig, &mut long_dest);

        let mut dest = Vec::with_capacity(long_dest as usize);
        let pdest = dest.as_mut_ptr();

        if snappy_uncompress(porig, long_orig, pdest, &mut long_dest) == 0 {
            dest.set_len(long_dest as usize);
            Some(dest)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

Como referencia, los ejemplos usados aquí están disponibles también como una biblioteca en Github.

Destructores

La bibliotecas externas usualmente transfieren la pertenencia de los recursos a el código llamador. Cuando esto ocurre, debemos usar los destructores de Rust para proveer seguridad y la garantía de la liberación de dichos recursos (especialmente en el caso de un pánico).

Para mas información acerca de los destructores, echa un vistazo a la sección del trait Drop.

Callbacks desde código C a funciones

Rust

Algunas bibliotecas externas requieren el uso de callbacks para reportar de vuelta su estado o data parcial a el llamador. Es posible pasar funciones definidas en Rust a una biblioteca externa. El requerimiento para esto es que la función callback este marcada como `extern` y con la convención de llamadas correcta para hacer posible su llamado desde código C.

La función callback puede entonces ser enviada a través de una llamada de registro a la biblioteca en C y su posterior invocación desde allá.

Un ejemplo básico es:

Código Rust:

```
extern fn callback(a: i32) {
    println!("He sido llamado desde C con el valor {0}", a);
}

#[link(name = "extlib")]
extern {
   fn registrar_callback(cb: extern fn(i32)) -> i32;
   fn disparar_callback();
}

fn main() {
    unsafe {
        registrar_callback(callback);
        disparar_callback(); // Dispara el callback
    }
}
```

Código C:

```
typedef void (*callback_rust)(int32_t);
callback_rust cb;

int32_t registrar_callback(callback_rust callback) {
    cb = callback;
    return 1;
}

void disparar_callback() {
  cb(7); // Llamara a callback(7) en Rust
}
```

En este ejemplo el `main()` de Rust llamara a `disparar_callback()` en C, que a su vez llamara de vuelta a `callback()` en Rust.

Apuntando callbacks a objetos Rust

El ejemplo anterior demostró como una función global puede ser llamada desde código en C. Si embrago a veces se desea que el callback apunte a un objeto Rust especial. Este podría ser el objeto que representa el envoltorio para el objeto respectivo en C.

Todo esto puede ser logrado a través del paso de un apuntador plano a la biblioteca en C. La biblioteca en C entonces puede incluir el apuntador a el objeto Rust en la notificación. Esto permitirá a el callback acceder de manera insegura el objeto Rust referenciado.

Código Rust:

```
#[repr(C)]
struct ObjetoRust {
    a: i32,
    // otros miembros
}

extern "C" fn callback(objetivo: *mut ObjetoRust, a: i32) {
    println!("He sido llamado desde C con el valor {0}", a);
    unsafe {
        // Actualiza el valor en ObjetoRust con el valor recibido desde el callback
        (*objetivo).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn registrar_callback(objetivo: *mut ObjetoRust,
                         cb: extern fn(*mut ObjetoRust, i32)) -> i32;
   fn disparar_callback();
}

fn main() {
    // Creando el objeto que sera referenciado en el callback
    let mut objeto_rust = Box::new(ObjetoRust { a: 5 });

    unsafe {
        registrar_callback(&mut *objeto_rust, callback);
        disparar_callback();
    }
}
```

Código C:

```
typedef void (*callback_rust)(void*, int32_t);
void* objetivo_cb;
callback_rust cb;

int32_t registrar_callback(void* objetivo_callback, callback_rust callback) {
    objetivo_cb = objetivo_callback;
    cb = callback;
    return 1;
}

void disparar_callback() {
  cb(objetivo_cb, 7); // Llamare a callback(&ObjetoRust, 7) in Rust
}
```

Callbacks Asíncronos

En los ejemplos anteriores los callbacks son invocados como una reacción directa a una llamada a función a la biblioteca externa en C. El control sobre el hilo actual es cambiado de Rust a C para la ejecución del callback, pero al final el callback es ejecutado en el mismo hilo que llamo a la función que disparo el callback.

Las cosas se vuelven mas complicadas cuando la biblioteca externa crea sus propios hilos e invoca los callbacks desde ellos. En estos casos el acceso a las estructuras de datos Rust dentro de los callbacks es especialmente inseguro y mecanismos de sincronización apropiados deben ser usados. Ademas de los mecanismos clásicos de sincronización como mutexes, una posibilidad en Rust es el uso de canales (en `std::sync::mpsc`) para transferir data desde el hilo C que invoco el callback a un hilo Rust.

Si un callback asíncrono tiene como objetivo un objeto especial en el espacio de direcciones de Rust es absolutamente necesario también que ningún otro callback sea ejecutados por la biblioteca en C después de que el respectivo objeto Rust haya sido destruido. Esto puede ser llevado a cabo de-registrando el callback en el destructor del objeto y diseñando la biblioteca de manera tal que garantice que ningún callback sera ejecutado después de la de-registracion.

Enlace

El atributo `link` en bloques `extern` proporciona el bloque de construcción basico para instruir a rustc como enlazar con bibliotecas nativas. Hoy en dia hay dos formas del atributo link:

- `#[link(name = "foo")]`

- `#[link(name = "foo", kind = "bar")]`

En ambos casos, `foo` es el nombre de la biblioteca nativa con la cual estamos enlazando, y en el segundo caso `bar` es el tipo de biblioteca nativa a la cual esta enlazando el compilador. Actualmente hay tres tipos de bibliotecas nativas conocidos:

- Dinamicas - `#[link(name = "readline")]`
- Estaticas - `#[link(name = "my_build_dependency", kind = "static")]`
- Frameworks - `#[link(name = "CoreFoundation", kind = "framework")]`

Nota que los frameworks solo están disponibles en objetivos OSX.

Los diferentes valores `kind` tienen como objeto diferenciar como la biblioteca nativa participa en el enlace. Desde la perspectiva del enlace, el compilador de Rust crea dos tipos de artefactos: parciales (rlib/staticlib) y finales (dylib/binary). Las dependencias en bibliotecas nativas y frameworks son propagadas a la frontera de artefacto final, mientras que las dependencias estáticas no son propagadas del todo, debido a que las bibliotecas estáticas están integradas directamente dentro del artefacto subsecuente.

Unos pocos ejemplos de como este modelo puede ser usado son:

- Una dependencia de construcción nativa: Algunas veces algún C/C++ de pegamento es necesario cuando se escribe un tipo especifico de código Rust, pero la distribución del código en un formato de biblioteca es simplemente una carga. En este caso, el código sera archivado en un `libfoo.a` y luego el crate rust declarara una dependencia via `#[link(name = "foo", kind = "static")]` .

Independientemente del tipo de salida para el crate, la biblioteca nativa estática sera incluida en la salida, significando que la distribución de la biblioteca estática nativa no es necesaria.

- Una dependencia dinámica normal: Bibliotecas comunes del sistema (como `readline`) están disponibles en un gran numero de sistemas, y a menudo una copia estática de esas biblioteca puede no existir. Cuando esta dependencia es incluida en un crate Rust, objetivos parciales (como rlibs) no enlazaran a la biblioteca, pero cuando el rlib es incluido en un objetivo final (como un binario), la biblioteca sera enlazada.

En OSX, los frameworks se comportan con la misma semantica que una biblioteca dinámica.

Bloques Unsafe

Algunas operaciones, como dereferenciar punteros planos o llamadas a funciones que han sido marcadas como inseguras solo son permitidas dentro de bloques unsafe. Los bloques unsafe aíslan la inseguridad y son una promesa al compilador que la inseguridad no se filtrara hacia el exterior del bloque.

Las funciones unsafe, por otro lado, hacen publicidad de la inseguridad a el mundo exterior. Una función insegura se escribe de la siguiente manera:

```
unsafe fn kaboom(ptr: *const i32) -> i32 { *ptr }
```

Esta función puede ser invocada solo desde un bloque `unsafe` u otra función `unsafe`.

Accediendo globales externas

APIs foráneas algunas veces exportan una variable global que podría hacer algo como llevar registro de algun estado global. Con la finalidad de acceder dichas variables, debes declararlas en bloques `extern` con la palabra reservada `static`:

```
# #![feature(libc)]
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("Tienes la version {} de readline.",
             rl_readline_version as i32);
}
```

Alternativamente, podrías necesitar alterar estado global proporcionado por una interfaz foránea. Para hacer esto, los estáticos pueden ser declarados con `mut` con la finalidad de poder mutarlos.

```
# #![feature(libc)]
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::new("[my-awesome-shell] $").unwrap();
    unsafe {
        rl_prompt = prompt.as_ptr();

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

        rl_prompt = ptr::null();
    }
}
```

Nota que toda la interacción con un `static mut` es insegura, ambos lectura y escritura. Lidiar con estado global mutable requiere de un gran cuidado.

Convenciones de llamadas foráneas

La mayoría de el código foráneo expone un ABI C, y Rust usa por defecto la convención de llamadas de la plataforma al momento de llamar funciones externas. Algunas funciones foráneas, notablemente el API de Windows, usan otra convenión de llamadas. Rust provee una forma de informar al compilador acerca de cual convención debe ser usada:

```
# #![feature(libc)]
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
# fn main() { }
```

Esto aplica a el bloque `extern` completo. La lista de restricciones ABI soportadas son:

- `stdcall`

- `aapcs`
- `cdecl`
- `fastcall`
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`

La mayoría de las abis son auto-explicativas, pero el abi `system` puede parecer un poco raro. Esta restricción selecciona cualquiera que sea el ABI apropiado para interoperar con las bibliotecas objetivo. Por ejemplo, en win32 con una arquitectura x86, significa que el abi usado sera `stdcall`. En x86_64, sin embargo, Windows usa la convención de llamadas `C`, entonces se usa `C`. Esto se traduce a que en nuestro ejemplo anterior, pudimos haber hecho uso de `extern "system" { ... }` para definir un bloque para todos los sistemas Windows, no solo los x86.

Interoperabilidad con código externo

Rust garantiza que la distribución de un `struct` sea compatible con la representación de la plataforma en C solo si el atributo `#[repr(C)]` es aplicado. `#[repr(C, packed)]` puede ser usado para distribuir los miembros sin padding. `#[repr(C)]` puede ser aplicado también a un enum.

Los boxes Rust (`Box<T>`) usan apuntadores no-nulables como handles que apuntan a el objeto contenido. Sin embargo, no deben ser creados manualmente debido que son manejados por asignadores internos. La referencias pueden ser asumidas de manera segura como apuntadores no-nulables directos al tipo. Sin embargo, romper el chequeo de préstamo (borrowing) o las reglas de mutabilidad no se garantiza ser seguro, es por ello que se prefiere el uso de apuntadores planos (`*`) de ser necesario debido a que el compilador no puede asumir muchas cosas acerca de ellos.

Los vectores y las cadenas de caracteres comparten la misma distribución en memoria, y utilidades para interactuar con APIs C están disponibles en los módulos `vec` y `str`. Sin embargo, las cadenas de caracteres no son terminadas en `\0`. Si necesitas una cadena de caracteres que termine en NUL para interactuar con C, debes entonces hacer uso de el tipo `CString` en el modulo `std::ffi`.

La biblioteca estándar incluye alias de tipo y definiciones de funciones para la biblioteca estándar de C en el modulo `libc`, y Rust enlaza por defecto con `libc` y `libm`.

La "optimizacion de apuntador nulable"

Ciertos tipos están definidos para no ser `null`. Esto incluye referencias (`&T`, `&mut T`), boxes (`Box<T>`), y apuntadores a funciones (`extern "abi" fn()`). Cuando haces interfaz con C, apuntadores que pueden ser null son usados algunas veces. Como un caso especial, un `enum` genérico que contiene exactamente dos variantes, de las cuales una no contiene data y la otra contiene un solo campo, es elegible para la "optimizacion de apuntador nulable". Cuando dicha enum es instanciada con uno de los tipos no-nulables, es representada como un solo apuntador, y la variante sin data es representada como el apuntador null. Entonces `Option<extern "C" fn(c_int) -> c_int>` es como uno representa un apuntador a función nullable usando el ABI de C.

Llamando coding Rust desde C

Podrías querer compilar código Rust de modo que permita ser llamado desde C. Esto es fácil, pero requiere ciertas cosas:

```
#[no_mangle]
pub extern fn hola_rust() -> *const u8 {
    "Hola, mundo!\0".as_ptr()
}
# fn main() {}
```

El `extern` hace que esta función se adhiera a la convención de llamadas de C, tal y como se discute en "Convenciones de llamadas foráneas". El atributo `no_mangle` desactiva el estropeo (mangling) de nombres de Rust, de manera tal que sea mas facil de enlazar.

FFI y pánicos

Es importante estar consciente de los `panic!` os cuando se trabaja con FFI. Un `panic!` en al limite de FFI es comportamiento indefinido. Si estas escribiendo código que pueda hacer pánico, debes ejecutarlo en otro hilo, de esta forma el pánico no se propagara hasta C:

```
use std::thread;

#[no_mangle]
pub extern fn oh_no() -> i32 {
    let h = thread::spawn(|| {
        panic!("Oops!");
    });

    match h.join() {
        Ok(_) => 1,
        Err(_) => 0,
    }
}
# fn main() {}
```

% Borrow y AsRef

Lost traits `Borrow` and `AsRef` son muy similares, pero diferentes. He aquí un breve repaso acerca de lo que significan dichos traits:

Borrow

El trait `Borrow` se usa cuando estamos escribiendo un estructura de datos, y deseamos usar un tipo owned o un tipo borrowed como sinónimo por alguna razón.

Por ejemplo, `HashMap` posee un metodo `get` que usa `Borrow` :

```
fn get(&self, k: &Q) -> Option<&V>
    where K: Borrow“,
          Q: Hash + Eq
”
```

Esta firma es complicada. El parametro `K` es en lo que estamos interesados ahora. Se refiere a un parametro del `HashMap` en si mismo:

```
struct HashMap {
```

El parametro `K` es el tipo de la *clave* que usa el `HashMap` . Al ver de nuevo la firma de `get()` , solo podemos usar `get()` cuando la clave implementa `Borrow<Q>` . De esta manera, podemos crear un `HashMap` que use claves `String` pero al momento de buscar use `&str` s:

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

let mut map = HashMap::new();
map.insert("Foo".to_string(), 42);

assert_eq!(map.get("Foo"), Some(&42));
```

Esto es debido a que la biblioteca estándar posee una `impl Borrow<str> for String` .

Para la mayoría de los tipos, cuando desees tomar un tipo ya sea owned o borrowed, un &T es suficiente. Pero un area en la cual `Borrow` es efectivo es cuando hay mas de un tipo de valor borrowed. Esto es especialmente cierto en referencias y slices: puedes tener ambos un `&T` un `&mut T` . Si queremos aceptar ambos tipos `Borrow` es lo que necesitamos:

```
use std::borrow::Borrow;
use std::fmt::Display;

fn foo<T: Borrow<i32> + Display>(a: T) {
    println!("a es un borrowed: {}", a);
}

let mut i = 5;

foo(&i);
foo(&mut i);
```

This will print out `a es un borrowed: 5` dos veces.

AsRef

El trait `AsRef` es un trait de conversion. Es usado para convertir algún valor a una referencia en código generico. Como esto:

```
let s = "Hola".to_string();

fn foo<T: AsRef<str>>(s: T) {
    let slice = s.as_ref();
}
```

Cual debería usar?

Podemos ver como ambos son una especie de lo mismo: ambos lidian con las versiones owned y borrowed de algún tipo. Sin embargo, son diferentes.

Escoge `Borrow` cuando quieras abstraerte por encima de diferentes tipos de borrowing, o cuando estas construyendo una estructura de datos que trata los valores owned y borrowed de formas equivalentes, como el hashing y la comparación.

Usa `AsRef` cuando desees convertir algo directamente en una referencia, y estés escribiendo código genérico.

% Canales de Distribución

El proyecto Rust usa un concepto denominado ‘canales de distribución’ para administrar los releases.

Es importante entender este proceso para así poder decidir cual version de Rust debe usar tu proyecto.

Visión general

Hay tres canales para los releases de Rust:

- Nocturno (nightly)
- Beta
- Estable (stable)

Los releases nocturnos son creados una vez al día. Cada seis semanas, el ultimo release nocturno es promovido a 'Beta'. En este punto, solo recibirá parches que arreglen errores serios. Seis semanas después el beta es promovido a 'Estable', y se convierte en el nuevo release de `1.x`.

Este proceso ocurre en paralelo. Cada seis semanas, en el mismo día el nocturno sube a beta, el beta es promovido a estable. Cuando `1.x` es liberado, al mismo tiempo, `1.(x + 1)-beta` es liberado, y el nocturno se vuelve la primera version de `1.(x + 2)-nightly`.

Escogiendo una versión

Generalmente hablando, a menos que tengas una razón especifica, deberías usar el canal de distribución estable. Esos releases tienen como objetivo una audiencia general.

Sin embargo, dependiendo de tu interés en Rust, podrías escoger el nocturno. El balance es el siguiente: en el canal nocturno, puedes hacer uso de nuevas características de Rust. Sin embargo, las características inestables están sujetas al cambio, y es por ello que cualquier nocturno tiene la posibilidad de romper tu código. Si usas el release estable, no tienes acceso a características experimentales, pero el siguiente release de Rust no causara problemas significativos a causa de cambios.

Ayudando a el ecosistema a traves de CI

Que hay acerca de beta? Nosotros recomendamos a todos los usuarios Rust que usan el canal de distribución estable a que también prueben en sus sistemas de integración continua con el canal beta. Esto ayudara a advertir al equipo en caso de que exista una regresión accidental.

Adicionalmente, probar contra el nocturno puede detectar regresiones mucho mas temprano, es por ello que si no te importa tener un tercer build, apreciamos que pruebes contra todos los canales.

% Sintaxis y Semántica

Esta sección divide a Rust en pequeños pedazos, uno para cada concepto.

Si deseas aprender Rust de abajo hacia arriba, leer esta sección en orden es una muy buena forma de hacerlo.

Estas secciones forman una referencia para cada concepto, si estas leyendo otro tutorial y encuentras algo confuso, puedes encontrarlo explicado en algún lugar de esta sección.

% Enlaces a Variable

Virtualmente cualquier programa no-'Hola Mundo' usa *enlaces a variables*. Dichos enlaces a variables lucen así:

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

Colocar `fn main() {` en cada ejemplo es un poco tedioso, así que en el futuro lo omitiremos. Si estas siguiendo paso a paso, asegurate de editar tu función `main()`, en lugar de dejarla por fuera. De otro modo obtendrás un error.

En muchos lenguajes, esto es llamado una *variable*, pero los enlaces a variable de Rust tienen un par de trucos bajo la manga. Por ejemplo el lado izquierdo de una expresión `let` es un 'patron', no un simple nombre de variable. Esto se traduce en que podemos hacer cosas como:

```
let (x, y) = (1, 2);
```

Después de que esta expresión es evaluada, `x` sera uno, y `y` sera dos. Los patrones son realmente poderosos, y tienen su propia sección en el libro. No necesitamos esas facilidades por ahora, solo mantengamos esto en nuestras mentes mientras avanzamos.

Rust es un lenguaje estáticamente tipificado, lo que significa que especificamos nuestros tipos por adelantado, y estos son chequeados en tiempo de compilación. Entonces, porque nuestro primer ejemplo compila? Bueno, Rust tiene esta cosa llamada 'inferencia de tipos'. Si puede determinar el tipo de algo, Rust no requiere que escribas el tipo.

Podemos agregar el tipo si lo deseamos. Los tipos vienen después de dos puntos (`:`):

```
let x: i32 = 5;
```

Si te pidiera leer esto en voz alta al resto de la clase, dirías " `x` es un enlace con el tipo `i32` y el valor `cinco`."

En este caso decidimos representar `x` como un entero con signo de 32 bits. Rust posee muchos tipos de enteros primitivos diferentes. Estos comienzan con `i` para los enteros con signo y con `u` para los enteros sin signo. Los tamaños posibles para enteros son 8, 16, 32, y 64 bits.

En ejemplos futuros, podríamos anotar el tipo en un comentario. El ejemplo luciría así:

```
fn main() {
    let x = 5; // x: i32
}
```

Nota las similitudes entre la anotación y la sintaxis que usas con `let`. Incluir esta clase de comentarios no es idiomático en Rust, pero los incluiremos ocasionalmente para ayudarte a entender cuales son los tipos que Rust infiere.

Por defecto, los enlaces son *inmutables*. Este código no compilara:

```
let x = 5;
x = 10;
```

Dara como resultado el siguiente error:

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~
```

Si deseas que un enlace a variable sea mutable, puedes hacer uso de `mut`:

```
let mut x = 5; // mut x: i32
x = 10;
```

No hay una razón única por la cual los enlaces a variable son inmutables por defecto, pero podemos pensar acerca de ello a través de uno de los focos principales de Rust: seguridad. Si olvidas decir `mut`. el compilador lo notará, y te hará saber que has mutado algo que no tenias intención de mutar. Si los enlaces a variables fuesen mutables por defecto, el compilador no seria capaz de decirte esto. Si tenias la intención de mutar algo, entonces la solución es muy fácil: agregar `mut`.

Hay otras buenas razones para evitar estado mutable siempre que sea posible, pero estan fuera del alcance de esta guisa. En general, puedes frecuentemente evitar mutación explicita, y esto es preferible en Rust. Dicho esto, algunas veces, la mutación es justo lo que necesitas, es por ello que no esta prohibida.

Volvamos a los enlaces a variables. Los enlaces a variable en Rust poseen un aspecto mas que difiere de otros lenguajes: se requiere esten inicializados a un valor antes que puedas usarlos.

Pongamos esto a prueba. Cambia tu archivo `src/main.rs` para que luzca así:

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

    println!("Hola, mundo!");
}
```

Puedes usar `cargo build` en la linea de comando para compilarlo. Obtendrás una advertencia pero aun así imprimirá "Hola, mundo!":

```
   Compiling hola_mundo v0.0.1 (file:///home/tu/proyectos/hola_mundo)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)]
   on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust nos advierte que nunca hacemos uso del enlace a variable, pero debido a que nunca la usamos, no hay peligro, no hay falta. Sin embargo, las cosas cambian si efectivamente intentamos usar `x`. Hagamos eso. Cambia tu programa para que luzca de la siguiente manera:

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

    println!("El valor de x es: {}", x);
}
```

Intenta compilarlo. Obtendrás un error:

```
$ cargo build
   Compiling hola_mundo v0.0.1 (file:///home/tu/proyectos/hola_mundo)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("El valor de x es: {}", x);
                                                    ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust no te permitirá usar un valor que no haya sido inicializado previamente. A continuación hablemos de lo que le hemos agregado a `println!`.

Si incluyes un par de llaves (`{}` , algunas personas los llaman bigotes/moustaches...) en la cadena de caracteres a imprimir, Rust lo interpretara como una petición para interpolar alguna clase de valor. *La interpolación en cadenas de caracteres* es un termino en ciencias de la computación que significa "coloca esto dentro de la cadena de caracteres". Agregamos una coma, y luego `x` , para indicar que queremos que este sea el valor interpolado. La coma es usada para separar los argumentos que pasamos a las funciones y los macros, en el caso de pasar mas de uno.

Cuando usas las llaves, Rust intentará mostrar el valor de una forma que tenga sentido después de chequear su tipo. Si deseamos especificar un formato mas detallado, existen un amplio numero de opciones disponible. Por ahora, nos apegaremos al comportamiento por defecto: los números enteros no son muy complicados de imprimir.

% Funciones

Todo programa Rust posee al menos una función, la función `main`:

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

Esta es la declaración de función mas simple posible. Como hemos mencionado anteriormente, `fn` dice 'esto es una función', seguido del nombre, paréntesis debido a que esta función no recibe ningún argumento, y luego llaves para indicar el cuerpo de la función. He aquí una función llamada `foo`:

```
fn foo() {
}
```

Entonces, que hay acerca de recibir argumentos? A continuación una función que imprime un numero:

```
fn imprimir_numero(x: i32) {
    println!("x es: {}", x);
}
```

Acá un programa completo que hace uso de `imprimir_numero`:

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

fn imprimir_numero(x: i32) {
    println!("x es: {}", x);
}
```

Como podrás ver, los argumentos de función trabajan de manera muy similar a las declaraciones `let`: agregas un tipo a el nombre del argumento, después de dos puntos.

He aquí un programa completo que suma dos números y luego los imprime:

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

fn imprimir_suma(x: i32, y: i32) {
    println!("la suma es: {}", x + y);
}
```

Los argumentos son separados por una coma, en ambos casos, cuando llamas a la función así como cuando la declaras.

A diferencia de `let`, *debes* declarar los tipos de los argumentos. Lo siguiente, no compilara:

```
fn imprimir_suma(x, y) {
    println!("suma es: {}", x + y);
}
```

Obtendrías el siguiente error:

```
expected one of `!`, `:`, or `@`, found `)`
fn imprimir_suma(x, y) {
```

Esto es una deliberada decisión de diseño. Aunque la inferencia del programa completo es possible, los lenguajes que la poseen, como Haskell, a menudo sugieren que documentar tus tipos de manera explicita es una buena practica. Nosotros coincidimos en que obligar a las funciones a declarar los tipos al mismo tiempo permitiendo inferencia dentro de la función es un maravilloso punto medio entre inferencia completa y la ausencia de inferencia.

Que hay acerca de retornar un valor? A continuación una función que suma uno a un entero:

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

La funciones en Rust retornan exactamente un valor, y el tipo es declarado después de una ‘flecha’, que es un guión (`-`) seguido por un signo mayor-que (`>`). La ultima linea de una función determina lo que esta retorna. Notaras la ausencia de un punto y coma aquí. De agregarlo:

```
fn suma_uno(x: i32) -> i32 {
    x + 1;
}
```

Obtendríamos un error:

```
error: not all control paths return a value
fn suma_uno(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

Lo anterior revela dos cosas interesantes acerca de Rust: es un lenguaje basado en expresiones, y los puntos y coma son diferentes a los de otros lenguajes basados en 'llaves y puntos y coma'. Estas dos cosas están relacionadas.

Expresiones vs. Sentencias

Rust en un lenguaje principalmente basado en expresiones. Existen solo dos tipos de sentencias, todo lo demás es una expresión.

Entonces, cual es la diferencia? Las expresiones retornan un valor, y las sentencias no. Es por ello que terminamos con el error 'not all control paths return a value' aquí: la sentencia `x + 1;` no retorna ningún valor.Hay dos tipos de sentencias en Rust: 'sentencias de declaración' and 'sentencias de expresión'. Todo lo demás es una expresión. Hablemos primero de las sentencias de declaración.

En algunos lenguajes, los enlaces a variable pueden ser escritos como expresiones, no solo sentencias. Como en Ruby:

```
x = y = 5
```

En Rust, sin embargo, el uso de `let` para introducir un enlace a variable *no es* una expresión. Lo siguiente producirá un error en tiempo de compilación:

```
let x = (let y = 5); // expected identifier, found keyword `let`
```

El compilador nos dice que estaba esperando el comienzo de una expresión, y un `let` puede ser solo una sentencia, no una expresión.

Nota que la asignación a una variable que ha sido previamente asignada (e.j. `y = 5`) es una expresión, aun así su valor no es particularmente util. A diferencia de otros lenguajes en los que una asignación es evaluada a el valor asignado (e.j. `5` en ejemplo anterior), en Rust el valor de una asignación es una tupla vacía `()` debido a que el valor asignado puede tener un solo dueño y cualquier otro valor de retorno seria sorpresivo:

```
let mut y = 5;

let x = (y = 6);  // x posee el valor `()`, no `6`
```

El segundo tipo de sentencia en Rust es la *sentencia de expresión*. Su propósito es convertir cualquier expresión en una sentencia. En términos prácticos, la gramática de Rust espera que una sentencia sea seguida por mas sentencias. Esto significa que puedes usar puntos y coma para separar una expresión de otra. Esto causa que Rust luzca muy similar a esos lenguajes en los cuales se requiere un punto y coma al final de cada linea, de hecho, observaras puntos y coma al final de casi todas las lineas de Rust que veras.

Entonces, Cual es la excepción que nos hace decir "casi"? Ya lo has visto en el código anterior:

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

Nuestra función clama retornar un `i32`, pero con un punto y coma, retornaría `()`. Rust determina que esto no es probablemente lo que queremos, y sugiere la remoción del punto y coma en el error que vimos anteriormente.

Retornos tempranos

Pero que hay acerca de los retornos tempranos? Rust proporciona una palabra reservada para ello, `return`:

```
fn foo(x: i32) -> i32 {
    return x;

    // nunca alcanzaremos este código!
    x + 1
}
```

Usar un `return` como la ultima linea de una función es valido, pero es considerado pobre estilo:

```
fn foo(x: i32) -> i32 {
    return x + 1;
}
```

La definición previa sin el `return` puede lucir un poco extraña si nunca has trabajado con un lenguaje basado en expresiones, pero esta se vuelve intuitiva con el tiempo.

Funciones divergentes

Rust tiene una sintaxis especial para las 'funciones divergentes', que son funciones que no retornan:

```
fn diverge() -> ! {
    panic!("Esta función nunca retorna!");
}
```

`panic!` es una macro, similar a la `println!()` que ya hemos visto. A diferencia de `println!()`, `panic!()` causa que el hilo de ejecución actual termine de manera abrupta mostrando el mensaje proporcionado.

Debido a que esta función causara una salida abrupta, esta nunca retornara, es por ello que posee el tipo '`!`', que se lee 'diverge'. Una función divergente puede ser usada como cualquier tipo:

```
# fn diverge() -> ! {
#     panic!("Esta función nunca retorna!");
# }
let x: i32 = diverge();
let x: String = diverge();
```

Apuntadores a función

También podemos crear enlaces a variables que apunten a funciones:

```
let f: fn(i32) -> i32;
```

`f` es un enlace a variable que apunta a una función que toma un `i32` como argumento y retorna un `i32`. Por ejemplo:

```
fn mas_uno(i: i32) -> i32 {
    i + 1
}

// sin inferencia de tipos
let f: fn(i32) -> i32 = mas_uno;

// inferencia de tipos
let f = mas_uno;
```

Podemos entonces hacer uso de `f` para llamar a la función:

```
# fn mas_uno(i: i32) -> i32 { i + 1 }
# let f = mas_uno;
let seis = f(5);
```

% Tipos Primitivos

Rust posee un conjunto de tipos que son considerados 'primitivos'. Esto significa que están integrados en el lenguaje. Rust esta estructurado de tal manera que la biblioteca estándar también provee una numero de tipos útiles basados en los primitivos, pero estos son los mas primitivos.

Booleanos

Rust posee un tipo booleano integrado, denominado `bool`. Tiene dos posibles valores `true` y `false`:

```
let x = true;

let y: bool = false;
```

Un uso común de los booleanos es en condicionales `if`.

Puedes encontrar mas documentación para los `bool`eanos en la documentación de la biblioteca estándar (ingles).

`char`

El tipo `char` representa un unico valor escalar Unicode. Puedes crear `char`s con comillas simples: (`'`)

```
let x = 'x';
let dos_corazones = '';
```

A diferencia de otros lenguajes, esto significa que `char` en Rust no es un solo byte, sino cuatro.

Puedes encontrar mas documentación para los `char`s en la documentación de la biblioteca estándar (ingles).

Tipos numéricos

Rust posee una variedad de tipos numéricos en unas pocas categorías: con signo y sin signo, fijos y variables, de punto flotante y enteros.

Dichos tipos consisten de dos partes: la categoría, y el tamaño. Por ejemplo, `u16` es un tipo sin signo con un tamaño de dieciséis bits. Mas bits te permiten almacenar números mas grandes.

Si un literal de numero no posee nada que cause la inferencia de su tipo, se usan tipos por defecto:

```
let x = 42; // x tiene el tipo i32

let y = 1.0; // y tiene el tipo f64
```

He aquí una lista de los diferentes tipos numéricos, con enlaces a su documentación en la biblioteca estándar (ingles):

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

Veamos cada una de las diferentes categorías.

Con signo y sin signo

Los tipos de enteros vienen en dos variedades: con signo y sin signo. Para entender la diferencia, consideremos un numero con cuatro bits de tamaño. Un numero de cuatro bits con signo te permitiría almacenar los números desde `-8` a `+7`. Los números con signo usan la “representación del complemento a dos”. Un numero de cuatros bits sin signo, debido a que no necesita guardar los valores negativos, puede almacenar valores desde `0` hasta `+15`.

Los tipos sin signo usan una `u` para su categoria, y los tipos con signo usan `i`. La `i` es de `integer` (entero). Entonces, `u8` es un numero de ocho bits sin signo, y un `i8` es un numero de ocho bits con signo.

Tipos de tamaño fijo

Los tipos de tamaño fijo poseen un numero especifico de bits en su representación. Los tamaños validos son `8`, `16`, `32`, and `64`. Entonces `u32` es un entero sin signo de 32 bits, e `i64` es un entero con signo de 64 bits.

Tipos de tamaño variable

Rust también provee tipos para los cuales el tamaño depende del tamaño del apuntador en la maquina subyacente. Dichos tipos poseen la categoría 'size', y vienen en variantes con y sin signo. Esto resulta en dos tipos `isize` y `usize`.

Tipos de punto flotante

Rust también posee dos tipos de punto flotante: `f32` y `f64`. Estos corresponden a los números IEEE-754 de simple y doble precision.

Arreglos

Como muchos lenguajes de programación, Rust posee tipos lista para representar una secuencia de cosas. El mas básico es el *arreglo*, una lista de elementos del mismo tipo y de tamaño fijo. Por defecto los arreglos son inmutables.

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

Los arreglos tienen el tipo `[T; N]`. Hablaremos de la notación `T` en la sección de genéricos. La `N` es una constante en tiempo de compilación, para la longitud del arreglo.

Hay un atajo para la inicialización de cada uno de los elementos del arreglo a el mismo valor. En este ejemplo, cada elemento de `a` sera inicializado a `0`:

```
let a = [0; 20]; // a: [i32; 20]
```

Puedes obtener el numero de elementos del arreglo `a` con `a.len()`

```
let a = [1, 2, 3];

println!("a tiene {} elementos", a.len());
```

Puedes acceder un elemento del arreglo en particular con la *notación de subindices*:

```
let nombres = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("El segundo nombre es: {}", nombres[1]);
```

Los subindices comienzan en cero, como en la mayoría de los lenguajes de programación, entonces el primer nombre es `nombres[0]` y es segundo es `nombres[1]`. El ejemplo anterior imprime: `El segundo nombre es: Brian`. Si intentas usar un subíndice que no esta en el arreglo, obtendrás un error: el chequeo de los limites en el acceso al arreglo se realiza en tiempo de ejecución. El acceso errante como ese es la fuente de muchos bugs en los lenguajes de programación de sistemas.

Puedes encontrar mas documentación para los `array`s en la documentación de la biblioteca estándar (ingles).

Slices

Un slice es una referencia a (o una “vista” dentro de) otra estructura de datos. Los slices son útiles para permitir acceso seguro y eficiente a una porción de un arreglo sin involucrar el copiado. Por ejemplo, podrías querer hacer referencia a una sola linea de una archivo que ha sido previamente leído en memoria. Por naturaleza, un slice no es creado directamente, estos son creados para una variable que ya existe. Los slices poseen una longitud, pueden ser mutables o inmutables, y en muchas formas se comportan como los arreglos:

```
let a = [0, 1, 2, 3, 4];
let middle = &a[1..4]; // Un slice de a: solo los elementos 1, 2, y 3
let complete = &a[..]; // Un slice conteniendo todos los elementos de a.
```

Los slices poseen el tipo `&[T]`. Hablaremos acerca de `T` cuando cubramos los genericos.

Puedes encontrar mas documentación para los slices en la documentación de la biblioteca estándar (ingles)

`str`

El `str` de Rust es el tipo de cadena de caracteres mas primitivo. Como un tipo sin tamaño, y no es muy util en si mismo, pero se vuelve muy util cuando es puesto detrás de una referencia, como `&str`. Es por ello que lo dejaremos hasta aquí.

Puedes encontrar mas documentación para str en la documentación de la biblioteca estándar (ingles)

Tuplas

Una tupla es una lista ordenada de tamaño fijo. Como esto:

```
let x = (1, "hola");
```

Los paréntesis y comas forman esta tupla de longitud dos. He aquí el mismo código pero con anotaciones de tipo:

```
let x: (i32, &str) = (1, "hola");
```

Como puedes ver, el tipo de una tupla es justo como la tupla, pero con cada posición teniendo el tipo en lugar del valor. Los lectores cuidadosos también notaran que las tuplas son heterogéneas: tenemos un `i32` y a `&str` en esta tupla. En los lenguajes de programación de sistemas, las cadenas de caracteres son un poco mas complejas que en otros lenguajes. Por ahora solo lee `&str` como un *slice de cadena de caracteres*. Aprenderemos mas dentro de poco.

Puedes asignar una tupla a otra, si estas tienen los mismo tipos contenidos y la misma aridad. La tuples poseen la misma aridad cuando estas tienen el mismo tamaño.

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

Puedes acceder a los campos de una tupla a través de un *let con destructuracion*. He aquí un ejemplo:

```
let (x, y, z) = (1, 2, 3);

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

Recuerdas anteriormente cuando dije que el lado izquierdo de una sentencia `let` era mas poderoso que la simple asignación de un binding? Henos aquí. Podemos colocar un patron en el lado izquierdo de un `let`, y si este concuerda con el lado derecho, podemos asignar multiples bindings a variable de una sola vez. En este caso, el `let` "destructura" o "parte" la tupla, y asigna las partes a los tres bindings a variable.

Este patron es muy poderoso, y lo veremos repetido con frecuencia en el futuro.

Puedes eliminar la ambigüedad entre una tupla de un solo elemento y un valor encerrado en paréntesis usando una coma:

```
(0,); // tupla de un solo elemento
(0); // cero encerrado en parentesis
```

Indexado en tuplas

Puedes también acceder los campos de una tupla con la sintaxis de indexado:

```
let tupla = (1, 2, 3);

let x = tupla.0;
let y = tupla.1;
let z = tupla.2;

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

Al igual que el indexado en arreglos, este comienza en cero, pero a diferencia de el indexado en arreglos, se usan `.`, en lugar de `[]`s.

Puedes encontrar mas documentación para tuplas en la documentación de la biblioteca estándar (ingles)

Funciones

Las funciones también tienen un tipo! Estas lucen así:

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

En este caso, `x` es un 'apuntador' a una función que recibe un `i32` y retorna un `i32`.

% Comentarios

Ahora que hemos vistos algunas funciones, es buena idea aprender sobre los comentarios. Los comentarios son notas que dejas a otros programadores con la finalidad de explicar algún aspecto de tu código. El compilador mayormente, los ignora.

Rust posee dos tipos de comentarios que te deben interesar: *comentarios de linea* y *comentarios de de documentación*

```
// Los comentarios de linea son cualquier cosa después de '///' y se extienden hasta el fi

let x = 5; // este es también un comentario de linea

// Si tienes una larga explicación acerca de algo, puedes colocar comentarios
// de linea unos juntos. Pon un espacio entre tus // y tu comentario con el
// fin de hacerlos mas legibles.
```

El otro tipo de comentario es un comentario de documentación (o comentario doc). Los comentarios doc usan `///` en lugar de `//` , y soportan notación Markdown en su interior:

```
/// Suma uno al numero proporcionado
///
/// # Examples
///
///
```

/// let cinco = 5; /// /// assert_eq!(6, suma_uno(5)); /// # fn suma_uno(x: i32) -> i32 { /// # x + 1 /// # } /// ``` fn suma_uno(x: i32) -> i32 { x + 1 }

```

Existe otro estilo de comentarios, `//!`, con la finalidad de comentar los items contened
```

//! # La Biblioteca Estándar de Rust //! //! La Biblioteca Estándar de Rust proporciona la funcionalidad //! en tiempo de ejecución esencial para la construcción de software //! Rust portable. ```

Cuando se escriben comentarios doc, proporcionar algunos ejemplos de uso es muy, muy util. Notaras que hemos hecho uso de una macro: `assert_eq!` . Esta compara dos valores, y hace `panic!` o si estos no son iguales. Es muy util en la documentación. También existe otra macro, `assert!` , la cual hace `panic!` o si el valor proporcionado es `false` .

Puedes usar la herramienta `rustdoc` para generar documentación en HTML a partir de dichos comentarios, así como ejecutar el código de los ejemplos como pruebas!

% if

El enfoque de Rust con relación a `if` no es particularmente complejo, pero es mas similar a el `if` que encontraras en lenguajes con tipificado dinámico que al de los lenguajes de sistemas tradicionales. Hablemos un poco acerca de este, para estar seguros de que entiendas todos los matices.

`if` es una forma especifica de a un concepto mas general, el 'branch' (rama). El nombre proviene de una rama en un árbol: es un punto de decisión, en el cual dependiendo de una opción, multiples caminos pueden ser tomados.

En el caso de `if`, hay una sola elección que conduce a dos caminos:

```
let x = 5;

if x == 5 {
    println!("x es cinco!");
}
```

De haber cambiado el valor de `x` a algo diferente, esta linea no hubiese sido impresa. Para ser mas especifico, si la expresión después del `if` es evaluada a `true`, entonces el bloque de código es ejecutado. Si es `false`, dicho bloque no se invoca.

Si deseas que algo ocurra en el caso de `false`, debes hacer uso de un `else`:

```
let x = 5;

if x == 5 {
    println!("x es cinco!");
} else {
    println!("x no es cinco :(");
}
```

De haber mas de un caso, usa un `else if`:

```
let x = 5;

if x == 5 {
    println!("x es cinco!");
} else if x == 6 {
    println!("x es seis!");
} else {
    println!("x no es ni cinco ni seis :(");
}
```

Todo esto es bien estándar. Sin embargo, también puedes hacer esto:

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

Lo cual podemos (y probablemente deberiamos) escribir así:

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

Esto funciona porque `if` es una expresión. El valor de la expresión es el valor de la ultima expresión en el bloque que haya sido seleccionado. Un `if` sin un `else` siempre resulta en `()` como valor.

% Ciclos

Rust actualmente provee tres enfoques para realizar actividad iterativa. `loop`, `while` y `for`. Cada uno de dichos enfoques tiene sus propios usos.

loop

El ciclo infinito `loop` es el ciclo mas simple disponible en Rust. A traves del uso de la palabra reservada `loop`, Rust proporciona una forma de iterar indefinidamente hasta que alguna sentencia de terminación sea alcanzada. El `loop` infinito de Rust luce así:

```
loop {
    println!("Itera por siempre!");
}
```

while

Rust también tiene un ciclo `while`. Luce de esta manera:

```
let mut x = 5; // mut x: i32
let mut completado = false; // mut completado: bool

while !completado {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        completado = true;
    }
}
```

Los ciclos `while` son la elección correcta cuando no estas seguro acerca de cuantas veces necesitas iterar.

De necesitar un ciclo infinito, podrías sentirte tentado a escribir algo como esto:

```
while true {
```

Sin embargo, `loop` es por lejos, el mejor para este caso:

```
loop {
```

El análisis de flujo de control de Rust trata esta construcción de manera diferente que a un `while true`, debido a que sabemos que iteraremos por siempre. En general, mientras mas información le proporcionemos al compilador, este podría desempeñarse mejor en relación a la seguridad y generación de código, es por ello que siempre deberías preferir `loop` cuando tengas planeado iterar de manera indefinida.

for

El ciclo `for` es usado para iterar un numero particular de veces. Los ciclos `for` de Rust, sin embargo, trabajan de manera diferente a los de otros lenguajes de programación de sistemas. El `for` de Rust no luce como este ciclo `for` al "estilo C"

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

En su lugar, el ciclo `for` de Rust luce así:

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

En términos ligeramente mas abstractos:

```
for var in expresion {
    código
}
```

La expresión es un iterador. El iterador retorna una serie de elementos. Cada elemento es una iteración del ciclo. Ese valor es a su vez asignado a el nombre `var`, el cual es valido en el cuerpo del ciclo. Una vez que el ciclo ha terminado, el siguiente valor es obtenido del iterador, y se itera una vez mas. Cuando no hay mas valores, el ciclo `for` termina.

En nuestro ejemplo, `0..10` es una expresión que toma una posición de inicio y una de fin, y devuelve un iterador por sobre esos valores. El limite superior es exclusivo, entonces, nuestro loop imprimirá de `0` hasta `9`, no `10`.

Rut no posee el ciclo `for` al estilo de C, a proposito. Controlar manualmente cada elemento del ciclo es complicado y propenso a errores, incluso para programadores C experimentados.

Enumerate

Cuando necesitas llevar registro de cuantas veces has iterado, puedes usar la función `.enumerate()`.

En rangos:

```
for (i,j) in (5..10).enumerate() {
    println!("i = {} y j = {}", i, j);
}
```

Salida:

```
i = 0 y j = 5
i = 1 y j = 6
i = 2 y j = 7
i = 3 y j = 8
i = 4 y j = 9
```

No olvides colocar los paréntesis alrededor del rango.

En iteradores:

```
# let lineas = "hola\nmundo".lines();
for (numero_linea, linea) in lineas.enumerate() {
    println!("{}: {}", numero_linea, linea);
}
```

Outputs:

```
0: Contenido de la linea uno
1: Contenido de la linea dos
2: Contenido de la linea tres
3: Contenido de la linea cuatro
```

Finalizando la iteración de manera temprana

Echemos un vistazo a ese ciclo `while` que vimos con anterioridad:

```
let mut x = 5;
let mut completado = false;

while !completado {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        completado = true;
    }
}
```

Necesitamos mantener un binding a variable `mut` , `completado` , para saber cuando debíamos salir del ciclo. Rust posee dos palabras clave para ayudarnos a modificar el proceso de iteración: `break` y `continue` .

En este caso, podemos escribir el ciclo de una mejor forma con `break` :

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

Ahora iteramos de manera indefinida con `loop` y usamos `break` para romper el ciclo de manera temprana. Usar un `return` explícito también sirve para la terminación temprana del ciclo.

`continue` es similar, pero en lugar de terminar el ciclo, nos hace ir a la siguiente iteración. Lo siguiente imprimirá los numeros impares:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

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

Etiquetas loop

También podrías encontrar situaciones en las cuales tengas ciclos anidados y necesites especificar a cual de los ciclos pertenecen tus sentencias `break` o `continue`. Como en la mayoría de los lenguajes, por defecto un `break` o `continue` aplica a el ciclo mas interno. En el caso de que desearas aplicar un `break` o `continue` para alguno de los ciclos externos, puedes usar etiquetas para especificar a cual ciclo aplica la sentencia `break` o `continue`. Lo siguiente solo imprimirá cuando ambos `x` y `y` sean impares:

```
'exterior: for x in 0..10 {
    'interior: for y in 0..10 {
        if x % 2 == 0 { continue 'exterior; } // continua el ciclo por encima de x
        if y % 2 == 0 { continue 'interior; } // continua el ciclo por encima de y
        println!("x: {}, y: {}", x, y);
    }
}
```

% Pertenencia

Esta guía es una de las tres presentando el sistema de pertenencia de Rust. Este es una de las características mas únicas e irresistibles de Rust, con la que desarrolladores Rust deben estar familiarizados. A través de la pertenencia es como Rust logra su objetivo mas importante, la seguridad. Existen unos pocos conceptos distintos, cada uno con su propio capitulo:

- pertenencia, la que lees actualmente
- préstamo, y su característica asociada ‘referencias’
- tiempo de vida, un concepto avanzado del préstamo

Estos tres capítulos están relacionados, en orden. Necesitaras los tres para entender completamente el sistema de pertenencia de Rust.

Meta

Antes de entrar en detalle, dos notas importantes acerca del sistema de pertenencia.

Rust tiene foco en seguridad y velocidad. Rust logra esos objetivos a travez de muchas ‘abstracciones de cero costo’, lo que significa que en Rust, las abstracciones cuestan tan poco como sea posible para hacerlas funcionar. El sistema de pertenencia es un ejemplo primordial de una abstracción de cero costo. Todo el análisis del que estaremos hablando en la presente guía es *llevado a cabo en tiempo de compilación*. No pagas ningún costo en tiempo de ejecución por ninguna de estas facilidades.

Sin embargo, este sistema tiene cierto costo: la curva de aprendizaje. Muchos usuarios nuevos Rust experimentan algo que nosotros denominamos ‘pelear con el comprobador de préstamo’ (‘fighting with the borrow checker’), situación en la cual el compilador de Rust se rehusa a compilar un programa el cual el autor piensa valido. Esto ocurre con frecuencia debido a que el modelo mental del programador acerca de como funciona la pertenencia no concuerda con las reglas actuales implementadas en Rust. Probablemente tu experimentes cosas similares al comienzo. Sin embargo, hay buenas noticias: otros desarrolladores Rust experimentados reportan que una vez que trabajan con las reglas del sistema de pertenencia por un periodo de tiempo, pelean cada vez menos con el comprobador de préstamo.

Con eso en mente, aprendamos acerca de la pertenencia.

Pertenencia

Los Bindings a variable poseen una propiedad en Rust: Estos 'poseen pertenencia' sobre lo que están asociados. Lo que se traduce a que cuando un binding a variable sale de ámbito, Rust libera los recursos asociados a este. Por ejemplo:

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

Cuando `v` entra en ámbito, un nuevo `Vec<T>` es creado. En este caso el vector también asigna algo de memoria desde el montículo, para los tres elementos. Cuando `v` sale de ámbito al final de `foo()`, Rust limpiara todo lo relacionado al vector, incluyendo la memoria asignada desde el montículo. Esto ocurre de manera determinista, al final de el ámbito.

Semantica de movimiento

Hay algo mas sutil acá, Rust se asegura que solo exista *exactamente un* binding a cualquier recurso en particular. Por ejemplo, si tenemos un vector, podemos asignarlo a otro binding a variable:

```
let v = vec![1, 2, 3];

let v2 = v;
```

Pero, si intentamos usar `v` después, obtenemos un error:

```
let v = vec![1, 2, 3];

let v2 = v;

println!("v[0] es: {}", v[0]);
```

El error luce como este:

```
error: use of moved value: `v` (uso de valor movido: `v`)
println!("v[0] es: {}", v[0]);
                        ^
```

Algo similar ocurre si definimos una función que tome pertenencia, y tratamos de usar algo después de habérselo pasado como argumento:

```
fn tomar(v: Vec) {
    // lo que sucede acá no es relevante
}

let v = vec![1, 2, 3];

tomar(v);

println!("v[0] es: {}", v[0]);
```

El mismo error: 'use of moved value' (uso de valor movido). Cuando transferimos la pertenencia a algo, decimos que hemos 'movido' la cosa a la cual nos estamos refiriendo. No necesitas ningún tipo de anotación especial para ello, es lo que Rust hace por defecto.

Los detalles

La razón por la cual no podemos usar un binding a variable después de haberlo movido es sutil, pero importante. Cuando escribimos código como este:

```
let v = vec![1, 2, 3];

let v2 = v;
```

La primera linea asigna memoria para el objeto vector, `v`, y para la data que contiene. El objeto vector es entonces almacenado en la pila pila y contiene un apuntador a el contenido (`[1, 2, 3]`) almacenado en el monticulo. Cuando movemos `v` a `v2`, se crea una copia de dicho apuntador para `v2`. Todo esto significa que existirían dos apuntadores para el contenido del vector en el montículo. Lo cual viola las garantías de seguridad de Rust, introduciendo una condición de carrera. Es por ello que Rust prohibe el uso de `v` después que el movimiento ha sido llevado a cabo.

Es importante destacar que algunas optimizaciones podrían remover la copia de los bytes en la pila, dependiendo de ciertas circunstancias. Así que puede no ser tan ineficiente a como luce inicialmente.

Tipos `Copy`

Hemos establecido que cuando transferimos la pertenencia a otro binding a variable, no podemos usar el binding original. Sin embargo, existe un trait trait que cambia este comportamiento, se llama `Copy`. No hemos discutido los traits (rasgos) todavía, pero por

ahora puedes verlos como una anotación hecha a un tipo en particular la cual agrega comportamiento extra. Por ejemplo:

```
let v = 1;

let v2 = v;

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

En este caso, `v` es un `i32`, que implementa el trait `Copy`. Esto significa que, justo como en un movimiento, cuando asignamos `v` a `v2`, una copia de la data es hecha. Pero, a diferencia de lo que ocurre en un movimiento, podemos hacer uso de `v` después. Esto es debido a que un `i32` no posee apuntadores a data en ningún otro lugar, copiarlos significa copiado completo.

Todos los tipos primitivos implementan el trait `Copy` y su pertenencia no es movida como uno podría asumir, siguiendo las `reglas de pertenencia`. Como ejemplo, los siguientes dos pedazos de código solo compilan porque los tipos `i32` y `bool` implementan el trait `Copy`.

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

    let _y = doblar(a);
    println!("{}", a);
}

fn doblar(x: i32) -> i32 {
    x * 2
}
```

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

    let _y = cambiar_verdad(a);
    println!("{}", a);
}

fn cambiar_verdad(x: bool) -> bool {
    !x
}
```

De haber tenido tipos que no implementasen el trait `Copy`, hubiésemos obtenido un error de compilación por tratar de usar un valor movido.

```
error: use of moved value: `a`
println!("{}", a);
               ^
```

Discutiremos como hacer que tus propios tipos sean `Copy` en la sección de traits

Mas que pertenencia

Por supuesto, si necesitaremos devolver la pertenencia con cada función que escribiésemos:

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // hacer algo con v

    // devolviendo pertenencia
    v
}
```

Las cosas se volverían bastante tediosas. Se pone aun peor mientras tengamos mas cosas sobre las cuales queramos tener pertenencia:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // hacer algo con v1 y v2

    // devolviendo pertenencia, así como el resultado de nuestra función
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, respuesta) = foo(v1, v2);
```

Ugh! El tipo de retorno, la linea de retorno, y el llamado a la función se vuelven mucho mas complicados.

Por suerte, Rust ofrece una facilidad, el préstamo, facilidad que nos sirve para solucionar este problema.

Es el tópico de la siguiente sección!

% Referencias y Préstamo

Esta guía es una de las tres presentando el sistema de pertenencia de Rust. Esta es una de las características mas únicas y atractivas de Rust, con la que los desarrolladores Rust deben estar bien familiarizados. La pertenencia es como Rust logra su objetivo mayor, seguridad en el manejo de memoria. Existen unos pocos conceptos distintos, cada uno con su propio capitulo:

- pertenencia, el concepto principal
- prestamo, el que lees ahora
- tiempos de vida, un concepto avanzado del préstamo

Estos tres capítulos están relacionados, y en orden. Necesitaras leer los tres para entender completamente el sistema de pertenencia.

Meta

Antes de entrar en detalle, dos notas importantes acerca del sistema de pertenencia.

Rust tiene foco en seguridad y velocidad. Rust logra esos objetivos a travez de muchas ‘abstracciones de cero costo’, lo que significa que en Rust, las abstracciones cuestan tan poco como sea posible para hacerlas funcionar. El sistema de pertenencia es un ejemplo primordial de una abstracción de cero costo. Todo el análisis del que estaremos hablando en la presente guía es *llevado a cabo en tiempo de compilación*. No pagas ningún costo en tiempo de ejecución por ninguna de estas facilidades.

Sin embargo, este sistema tiene cierto costo: la curva de aprendizaje. Muchos usuarios nuevos Rust experimentan algo que nosotros denominamos ‘pelear con el comprobador de préstamo’ (‘fighting with the borrow checker’), situación en la cual el compilador de Rust se rehusa a compilar un programa el cual el autor piensa valido. Esto ocurre con frecuencia debido a que el modelo mental del programador acerca de como funciona la pertenencia no concuerda con las reglas actuales implementadas en Rust. Probablemente tu experimentes cosas similares al comienzo. Sin embargo, hay buenas noticias: otros desarrolladores Rust experimentados reportan que una vez que trabajan con las reglas del sistema de pertenencia por un periodo de tiempo, pelean cada vez menos con el comprobador de préstamo.

Con eso en mente, aprendamos acerca de el préstamo.

Préstamo

Al final de la sección de pertenencia, teníamos una función fea que lucia así:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // hacer algo con v1 y v2

    // devolviendo pertenencia, así como el resultado de nuestra función
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, respuesta) = foo(v1, v2);
```

Lo anterior, sin embargo, no es Rust idiomatico, debido a que no se beneficia de las ventajas del préstamo. He aqui el primer paso:

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // hacer algo con v1 y v2

    // retornando la respuesta
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let respuesta = foo(&v1, &v2);

// podemos usar a v1 y v2 aqui
```

En lugar de tomar `Vec<i32>`s como argumentos, tomamos una referencia: `&Vec<i32>`. Y en lugar de pasar `v1` y `v2` directamente, pasamos `&v1` y `&v2`. Llamamos al tipo `&T` una 'referencia', y en vez de tomar pertenencia sobre el recurso, este la toma prestado. Un enlace a variable que hace un préstamo de algo no libera el recurso cuando sale de ámbito. Esto significa que después de la llamada a `foo()`, podemos nuevamente hacer uso de los enlaces a variable originales.

Las referencias son inmutables, justo como los enlaces a variable. Esto se traduce a que dentro de `foo()`, los vectores no pueden ser cambiados:

```
fn foo(v: &Vec) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

falla con:

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

Insertar un valor causa una mutación en el vector, y no tenemos permitido hacerlo.

referencias &mut

Existe un segundo tipo de referencia: `&mut T`. Una 'referencia mutable' que permite mutar el recurso que estas tomando prestado. Por ejemplo:

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

Lo anterior imprimirá `6`. Hacemos a `y` una referencia mutable a `x`, entonces sumamos uno a lo que sea que `y` apunta. Notaras que `x` también tuvo que haber sido marcado como `mut`, de lo contrario, no hubiésemos podido tomar un préstamo mutable a un valor inmutable.

Notaras también que hemos agregado un asterisco (`*`) al frente de `y`, tornándolo en `*y`, esto es debido a que `y` es una referencia `&mut`. También necesitaras hacer uso de ellos para acceder a el contenido de una referencia.

De otro modo, las referencias `&mut` son como las referencias. *Existe* una gran diferencia entre las dos, y como estas interactuan. Habrás notado que existe algo que no huele muy bien en el ejemplo anterior, puesto que necesitamos ese ámbito extra, con los `{` y `}`. Si los removemos, obtenemos un error:

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `x` until the borrow ends
        let y = &mut x;
                     ^
note: previous borrow ends here
fn main() {

}
^
```

Al parecer, hay reglas.

Las Reglas

He aqui las reglas acerca del préstamo en Rust:

Primero, cualquier préstamo debe vivir en un ámbito no mayor al de el dueño. Segundo, puedes tener uno u otro de estos tipos de préstamo, pero no los dos al mismo tiempo:

- una o mas referencias (`&T`) a un recurso,
- exactamente una referencia mutable (`&mut T`).

Posiblemente notes que esto es muy similar, pero no exactamente igual, a la definición de una condición de carrera:

> Hay una ‘condición de carrera’ cuando dos o mas apuntadores acceden a la misma locación en memoria al mismo tiempo, en donde al menos uno esta escribiendo, y las operaciones no están sincronizadas.

Con las referencias, puedes tener cuantas desees, debido a que ninguna de ellas esta escribiendo. Si estas escribiendo, y necesitas dos o mas apuntadores a la misma memoria, puedes tener solo un `&mut` a la vez. Es así como Rust previene las condiciones de carrera en tiempo de compilación: obtendremos errores si rompemos las reglas.

Con esto en mente, consideremos nuestro ejemplo otra vez.

Pensando en ámbitos

He aqui el código:

```
let mut x = 5;
let y = &mut x;

*y += 1;

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

El código anterior genera el siguiente error:

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
```

Esto es debido a que hemos violado las reglas: tenemos un `&mut T` apuntando a `x`, y en consecuencia no tenemos permitido crear ningún `&T` s. Una cosa u otra. La nota apunta a como pensar acerca de este problema:

```
note: previous borrow ends here
fn main() {

}
^
```

En otras palabras, el préstamo mutable es mantenido a lo largo de el resto de nuestro ejemplo. Lo que queremos es que nuestro préstamo mutable termine *antes* que intentemos llamar a `println!` y hagamos un préstamo inmutable. En Rust, el préstamo esta asociado al ámbito en el cual el préstamo es valido. Nuestros ámbitos lucen así:

```
let mut x = 5;

let y = &mut x;    // -+ préstamo &mut de x comienza aqui
                   //  |
*y += 1;           //  |
                   //  |
println!("{}", x); // -+ - intento de tomar prestado x aqui
                   // -+ préstamo &mut de x termina aqui
```

Los ámbitos entran en conflicto: no podemos crear un `&x` mientras `y` esta en ámbito.

Entonces cuando agregamos llaves:

```
let mut x = 5;

{
    let y = &mut x; // -+ préstamo &mut de x comienza aqui
    *y += 1;        //  |
}                   // -+ ... y termina aqui

println!("{}", x);  // <- intento de tomar prestado x aqui
```

No hay problema. Nuestro préstamo mutable sale de ámbito antes de que creemos un préstamo inmutable. El ámbito es clave para ver cuanto dura el préstamo.

Problemas que el préstamo previene

Porque tenemos estas reglas restrictivas? Bueno, como lo notamos, estas reglas previenen condiciones de carrera. Que tipos de problemas causan las condiciones de carrera? Acá unos pocos.

Invalidación de Iteradores

Un ejemplo es la 'invalidación de iteradores', que ocurre cuando tratas de mutar una colección mientras estas iterando sobre ella. El comprobador de prestamos de Rust evita que esto ocurra:

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
}
```

Lo anterior imprime desde uno hasta tres. A medida que iteramos los vectores, solo se nos proporcionan referencias a sus elementos. `v` en si mismo es tomado prestado de manera inmutable, lo que se traduce en que no podamos cambiarlo mientras lo iteramos:

```
let mut v = vec![1, 2, 3];

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

He aqui el error:

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!(“{}”, i);
    v.push(34);
}
^
```

No podemos modificar `v` debido a que esta tomado prestado por el ciclo.

uso despues de liberacion (use after free)

Las referencias no deben vivir por mas tiempo que el recurso al cual estas apuntan. Rust chequeara los ámbitos de tus referencias para asegurarse de que esto sea cierto.

Si Rust no verificara esta propiedad, podriamos accidentalmente usar una referencia invalida. Por ejemplo:

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

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

Obtenemos el siguiente error:

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

En otras palabras, `y` es valido solo para el ámbito en donde `x` existe. Tan pronto como `x` se va, se hace invalido hacerle referencia. Es por ello que el error dice que el préstamo, ‘no vive lo suficiente’ (‘doesn’t live long enough’) ya que no es valido por la cantidad de tiempo correcta.

El mismo problema ocurre cuando la referencia es declarada *antes* de la variable a la cual hace referencia. Esto es debido a que los recursos dentro del mismo ámbito son liberados en orden opuesto al orden en el que fueron declarados:

```
let y: &i32;
let x = 5;
y = &x;

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

Obtenemos este error:

```
error: `x` does not live long enough
y = &x;
     ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
    let y: &i32;
    let x = 5;
    y = &x;

    println!("{}", y);
}

note: ...but borrowed value is only valid for the block suffix following
statement 1 at 3:14
    let x = 5;
    y = &x;

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

En el ejemplo anterior, `y` es declarada antes que `x`, significando que `y` vive mas que `x`, lo cual no esta permitido.

% Tiempos de Vida

Esta guía es una de las tres presentando el sistema de pertenencia de Rust. Esta es una de las características mas únicas y atractivas de Rust, con la que los desarrolladores Rust deben estar bien familiarizados. La pertenencia es como Rust logra su objetivo mayor, seguridad en el manejo de memoria. Existen unos pocos conceptos distintos, cada uno con su propio capitulo:

- pertenencia, el concepto principal
- [préstamo][borrowing], y sus característica asociada 'referencias'
- tiempos de vida, la que lees ahora

Estos tres capítulos están relacionados, y en orden. Necesitaras leer los tres para entender completamente el sistema de pertenencia.

Meta

Antes de entrar en detalle, dos notas importantes acerca del sistema de pertenencia.

Rust tiene foco en seguridad y velocidad. Rust logra esos objetivos a travez de muchas 'abstracciones de cero costo', lo que significa que en Rust, las abstracciones cuestan tan poco como sea posible para hacerlas funcionar. El sistema de pertenencia es un ejemplo primordial de una abstracción de cero costo. Todo el análisis del que estaremos hablando en la presente guía es *llevado a cabo en tiempo de compilación*. No pagas ningún costo en tiempo de ejecución por ninguna de estas facilidades.

Sin embargo, este sistema tiene cierto costo: la curva de aprendizaje. Muchos usuarios nuevos Rust experimentan algo que nosotros denominamos 'pelear con el comprobador de préstamo' ('fighting with the borrow checker'), situación en la cual el compilador de Rust se rehusa a compilar un programa el cual el autor piensa valido. Esto ocurre con frecuencia debido a que el modelo mental del programador acerca de como funciona la pertenencia no concuerda con las reglas actuales implementadas en Rust. Probablemente tu experimentes cosas similares al comienzo. Sin embargo, hay buenas noticias: otros desarrolladores Rust experimentados reportan que una vez que trabajan con las reglas del sistema de pertenencia por un periodo de tiempo, pelean cada vez menos con el comprobador de préstamo.

Con eso en mente, aprendamos acerca de los tiempos de vida.

Tiempos de vida

Prestar una referencia a otro recurso del que alguien mas es dueño puede ser complicado. Por ejemplo, imagina este conjunto de operaciones:

- Obtengo un handle a algún tipo de recurso.
- Te presto una referencia a el recurso.
- Decido que he terminado con el recurso, y lo libero, mientras todavía tienes la referencia a el.
- Tu decides usar el recurso.

Oh no! Tu referencia esta apuntando a un recurso invalido. Esto es llamado un puntero colgante o `uso después de liberación`, cuando el recurso es memoria.

Para arreglar esto, tenemos que asegurarnos que el paso cuatro nunca ocurra después del paso tres. El sistema de pertenencia de Rust lleva esto a cabo a través de un concepto denominado tiempos de vida, los cuales describen el ámbito en el cual una referencia es valida.

Cuando tenemos una función que toma una referencia como argumento, podemos ser implícitos o explícitos acerca del tiempo de vida de la referencia:

```
// implicito
fn foo(x: &i32) {
}

// explicito
fn bar<'a>(x: &'a i32) {
}
```

El `'a` se lee ‘el tiempo de vida a’. Técnicamente, toda referencia posee un tiempo de vida asociado a ella, pero el compilador te permite omitirlas en casos comunes. Antes que lleguemos a eso, analicemos el pedazo de código explícito:

```
fn bar<'a>(...)
```

Anteriormente hablamos un poco acerca de la sintaxis de funciones, pero no discutimos los `<>`s después de un nombre de función. Una función puede tener ‘parámetros genéricos’ entre los `<>`s, y los tiempos de vida son un tipo de parámetro genérico. Discutiremos otros tipos de genericos mas tarde en el libro, pero por ahora, enfoquémonos solo en el aspecto de los tiempos de vida.

Usamos `<>` para declarar nuestros tiempos de vida. Esto dice que `bar` posee un tiempo de vida, `'a`. De haber tenido referencias como parámetros, hubiese lucido de esta manera:

```
fn bar<'a, 'b="">(...)
```

Entonces en nuestra lista de parámetros, usamos los tiempos de vida que hemos nombrado:

```
...(x: &'a i32)
```

De haber querido una referencia `&mut`, pudimos haber hecho lo siguiente:

```
...(x: &'a mut i32)
```

Si comparas `&mut i32` con `&'a mut i32`, son lo mismo, es solo que el tiempo de vida `'a` se ha metido entre el `&` y el `mut i32`. Leemos `&mut i32` como 'una referencia mutable a un `i32`' y `&'a mut i32` como 'una referencia mutable a un `i32` con el tiempo de vida `'a`'.

En `struct`s

También necesitaras tiempos de vida explícitos cuando trabajes con `struct`s:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // esto es lo mismo que `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

Como puedes ver, los `struct`s pueden también tener tiempos de vida. En una forma similar a las funciones,

```
struct Foo<'a> {
# x: &'a i32,
# }
```

declara un tiempo de vida, y

```
# struct Foo<'a> {
x: &'a i32,
# }
```

hace uso de el. Entonces, porque necesitamos un tiempo de vida aquí? Necesitamos asegurarnos que cualquier referencia a un `Foo` no pueda vivir mas que la referencia a un `i32` que este contiene.

bloques `impl`

Implementemos un metodo en `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
}

fn main() {
    let y = &5; // esto es lo mismo que `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

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

Como puedes ver, necesitamos declarar un tiempo de vida para `Foo` en la linea `impl`. Repetimos `'a` dos veces, justo como en funciones: `impl<'a>` define un tiempo de vida `'a`, y `Foo<'a>` hace uso de el.

Multiples tiempo de vida

Si posees multiples referencias, puedes hacer uso de el mismo tiempo de vida multiples veces:

```
fn x_o_y<'a>(x: &'a str, y: &'a str) -> &'a str {
#    x
# }
```

Lo anterior dice que ambos `x` y `y` viven por el mismo ámbito, y que el valor de retorno también esta vivo para dicho ámbito. Si hubieses querido que `x` y `y` tuviesen diferentes tiempos de vida, pudiste haber hecho uso de multiples parámetros de tiempos de vida:

```
fn x_o_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
#    x
# }
```

En este ejemplo, `x` y `y` tienen diferentes ámbitos validos, pero el valor de retorno tiene el mismo tiempo de vida que `x`.

Pensando en ámbitos

Una forma de pensar acerca de los tiempos de vida es visualizar el ámbito en el cual es valida una referencia. Por ejemplo:

```
fn main() {
    let y = &5;     // -+ y entra en ambito
                    //  |
    // stuff        //  |
                    //  |
}                   // -+ y sale de ambito
```

Agregando nuestro `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y entra en ambito
    let f = Foo { x: y }; // -+ f entra en ambito
    // stuff              //  |
                          //  |
}                         // -+ f y y salen de ambito
```

Nuestro `f` vive dentro de el ámbito de `y`, es por ello que todo funciona. Que pasaría de lo contrario? El siguiente código no funcionaria:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x entra en ambito
                              //  |
    {                         //  |
        let y = &5;           // ---+ y entra en ambito
        let f = Foo { x: y }; // ---+ f entra en ambito
        x = &f.x;             //  | | error aqui
    }                         // ---+ f y y salen de ambito
                              //  |
    println!("{}", x);        //  |
}                             // -+ x sale de ambito
```

Uff! Como puedes ver aqui, los ámbitos de `f` y `y` son menores que el ámbito de `x`. Pero cuando hacemos `x = &f.x`, hacemos a `x` una referencia a algo que estar por salir de ámbito.

Los tiempos de vida con nombre son una forma de darles a dichos ámbitos un nombre. Darle un nombre a algo es el primer paso hacia poder hablar acerca de el.

'static

El tiempo de vida denominado ‘static’ es un tiempo de vida especial. Este señala que algo posee el tiempo de vida de el programa completo. La mayoría de los desarrolladores Rust conocen a `'static` cuando lidian con cadenas de caracteres:

```
let x: &'static str = "Hola, mundo.";
```

Los literales de cadenas de caracteres poseen el tipo `&'static str` puesto que la referencia esta siempre viva: estos son colocados en el segmento de datos del binario final. Otro ejemplo son las globales:

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

Lo anterior agrega un `i32` a el segmento de datos de el binario, y `x` es una referencia a el.

Elision de tiempos de vida

Rust soporta una inferencia de tipos poderosa en los cuerpos de función, pero esta prohibido en las firmas de elementos permitir razonamiento basado únicamente en la firma. Sin embargo, por razones ergonomicas, una inferencia secundaria muy restricta llamada “elision de tiempos de vida” se aplica en las firmas de función. La “elision de tiempos de vida” infiere basandose solo en los componentes de la firma sin basarse en el cuerpo de la función, unicamnete infiere parámetros de tiempos de vida, y hace esto con solo tres reglas fácilmente memorizables e inambiguas. Todo esto hace a la elision de tiempos de vida un atajo para escribir una firma, sin necesidad de ocultar los tipos involucrados puesto a que inferencia local completa sera aplicada a ellos.

Cuando se habla de elision de tiempos de vida, usamos el termino *tiempo de vida de entrada* y *tiempo de vida de salida*. Un *tiempo de vida de entrada* es un tiempo de vida asociado con un parámetro de una función, y un *tiempo de vida de salida* es un tiempo de vida asociado con el valor de retorno de una función. Por ejemplo, la siguiente función tiene un tiempo de vida de entrada:

```
fn foo<'a>(bar: &'a str)
```

Esta posee un tiempo de vida de salida:

```
fn foo<'a>() -> &'a str
```

La siguiente tiene un tiempo de vida en ambas posiciones:

```
fn foo<'a>(bar: &'a str) -> &'a str
```

He aqui las tres reglas:

- Cada tiempo de vida elidido en los argumentos de una función se convierte en un parámetro de tiempo de vida distinto.
- Si existe exactamente un solo tiempo de vida de entrada, elidido o no, ese tiempo de vida es asignado a todos los tiempos de vida elididos en los valores de retorno de esa función.
- Si existen multiples tiempos de vida de entrada, pero una de ellos es `&self` o `&mut self`, el tiempo de vida de `self` es asignado a todos los tiempos de vida de salida elididos.

De lo contrario, es un error elidir un tiempo de vida de salida.

Ejemplos

He aqui algunos ejemplos de funciones con tiempos de vida elididos. Hemos pareado cada ejemplo de un tiempo de vida elidido con su forma expandida.

```
fn print(s: &str); // elidido
fn print<'a>(s: &'a str); // expandido

fn debug(lvl: u32, s: &str); // elidido
fn debug<'a>(lvl: u32, s: &'a str); // expandido

// En el ejemplo anterior, `lvl` no necesita un tiempo de vida debido a que no es una ref

fn substr(s: &str, until: u32) -> &str; // elidido
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expandido

fn get_str() -> &str; // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str; // ILEGAL, dos entradas
fn frob<'a, 'b="">(s: &'a str, t: &'b str) -> &str; // Expandido: Tiempo de vida de salid

fn get_mut(&mut self) -> &mut T; // elidido
fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded

fn args(&mut self, args: &[T]) -> &mut Command // elidido
fn args<'a, 'b,="" T:ToCStr="">(&'a mut self, args: &'b [T]) -> &'a mut Command // expand

fn new(buf: &mut [u8]) -> BufWriter; // elidido
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a> // expanded
```

% Mutabilidad

Mutabilidad, es la habilidad que una cosa posee para ser cambiada, funciona un poco diferente en Rust que en otros lenguajes. El primer aspecto de la mutabilidad es que no esta habilitada por defecto:

```
let x = 5;
x = 6; // error!
```

Podemos introducir mutabilidad con la palabra reservada `mut`:

```
let mut x = 5;

x = 6; // no hay problema!
```

Esto es un enlace a variable mutable. Cuando un enlace a variable es mutable, significa que tienes permitido cambiar a lo que el enlace apunta. Entonces, en el ejemplo anterior, no esta cambiando el valor en `x`, en cambio, el enlace cambio de un `i32` a otro.

Si deseas cambiar a que apunta el enlace a variable, necesitaras una referencia mutable:

```
let mut x = 5;
let y = &mut x;
```

`y` es un enlace a variable inmutable a una referencia mutable, lo que significa que no puedes asociar `y` a otra cosa (`y = &mut z`), pero puedes mutar lo que sea a lo que `y` esta asociado (`*y = 5`). Una diferencia muy sutil.

Por supuesto, si necesitas ambas cosas:

```
let mut x = 5;
let mut y = &mut x;
```

Ahora `y` puede ser asociado a otro valor, y el valor que esta referenciando puede ser cambiado.

Es importante notar que `mut` es parte de un patron, de manera tal que puedas hacer cosas como:

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
# }
```

Mutabilidad Interior vs. Mutabilidad Exterior

Sin embargo, cuando decimos que algo es 'immutable' en Rust, esto no significa que no pueda ser cambiado: lo que decimos es que algo tiene 'mutabilidad exterior'. Considera, por ejemplo, `Arc<T>`:

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

Cuando llamamos a `clone()`, el `Arc<T>` necesita actualizar el contador de referencias. A pesar de que no hemos usado ningún `mut` aquí, `x` es un enlace inmutable, tampoco tomamos `&mut 5` o alguna mas. Entonces, que esta pasando?

Para entender esto, debemos volver al núcleo de la filosofía que guía a Rust, seguridad en el manejo de memoria, y el mecanismo a través del cual Rust la garantiza, el sistema de pertenencia, y mas específicamente, el préstamo:

> Puedes tener uno u otro de estos dos tipos de prestamo, pero no los dos al mismo tiempo:
>
> - una o mas referencias (`&T`) a un recurso,
> - exactamente una referencia mutable (`&mut T`).

Entonces, esa es la definición real de 'inmutabilidad': es seguro tener dos apuntadores? En el caso de `Arc<T>`'s, si: la mutación esta completamente contenida dentro de la estructura en si misma. No esta disponible al usuario. Por esta razón, retorna `clone()` `&T` s. Si proporcionase `&mut T` s, seria un problema.

Otros tipos como los del modulo `std::cell`, poseen lo opuesto: mutabilidad interior. Por ejemplo:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

RefCel proporciona referencias `&mut` a lo que contienen a traves del metodo `borrow_mut()`. No es esto peligroso? Que tal si hacemos:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
# (y, z);
```

Esto, en efecto, hará pánico en tiempo de ejecución. Esto es lo que `RefCell` hace: aplica las reglas de préstamo de Rust en tiempo de ejecución, y hace `panic!` os si dichas reglas son violadas. Lo anterior nos permite acercarnos a otro aspecto de las reglas de mutabilidad de Rust. Hablemos acerca de ello primero.

Mutabilidad a nivel de campos

La mutabilidad es una propiedad de un préstamo (`&mut`) o un enlace a variable (`let mut`). Esto se traduce en que, por ejemplo, no puedes tener un `struct` con algunos campos mutables y otros inmutables:

```
struct Punto {
    x: i32,
    mut y: i32, // nope
}
```

La mutabilidad de un struct esta en su enlace a variable:

```
struct Punto {
    x: i32,
    y: i32,
}

let mut a = Punto { x: 5, y: 6 };

a.x = 10;

let b = Punto { x: 5, y: 6};

b.x = 10; // error: cannot assign to immutable field `b.x`
```

Sin embargo, usando `Cell<T>` , puedes emular mutabilidad a nivel de campos:

```
use std::cell::Cell;

struct Punto {
    x: i32,
    y: Cell<i32>,
}

let punto = Punto { x: 5, y: Cell::new(6) };

punto.y.set(7);

println!("y: {:?}", punto.y);
```

Esto imprimirá `y: Cell { value: 7 }`. Hemos actualizado `y` de manera satisfactoria.

% Estructuras (Structs)

Las estructuras (`struct` s) son una forma de crear tipos de datos mas complejos. Por ejemplo, si estuviéramos haciendo cálculos que involucraran coordenadas en un espacio 2D, necesitaríamos ambos, un valor `x` y un valor `y` :

```
let origen_x = 0;
let origen_y = 0;
```

una `struct` nos permite combinar ambos en un único tipo de datos unificado:

```
struct Punto {
    x: i32,
    y: i32,
}

fn main() {
    let origen = Punto { x: 0, y: 0 }; // origin: Punto

    println!("El origen esta en ({}, {})", origen.x, origen.y);
}
```

Hay muchas cosas pasando acá, así que analicémoslo por partes. Declaramos una estructura con la palabra reservada `struct` , seguida de un nombre. Por convención, los `struct` s comienzan con una letra mayúscula y son camel cased: `PuntoEnElEspacio` , no `Punto_En_El_Espacio` .

Podemos crear una instancia de nuestra `struct` via `let` , como es usual, pero usamos una sintaxis estilo `clave: valor` para asignar cada campo. El order no necesita ser el mismo que en la declaración original.

Finalmente, debido a que tenemos nombres de campos, podemos acceder a ellos a través de la notación de puntos: `origen.x` .

Los valores en `struct` s son inmutables por defecto, justo como los demás enlaces a variables en Rust. Usa `mut` para hacerlos mutables:

```
struct Punto {
    x: i32,
    y: i32,
}

fn main() {
    let mut punto = Punto { x: 0, y: 0 };

    punto.x = 5;

    println!("El origen esta en ({}, {})", punto.x, punto.y);
}
```

Esto imprimirá `El origen esta en (5, 0)`.

Rust no soporta mutabilidad de campos a nivel de lenguaje, es por ello que no puedes escribir algo como esto:

```
struct Punto {
    mut x: i32,
    y: i32,
}
```

La mutabilidad es una propiedad del enlace a variable, no de la estructura en si misma. Si estas acostumbrado a la mutabilidad a nivel de campos, esto puede parecer un poco extraño al principio, pero simplifica las cosas de manera significativa. Incluso te permite hacer a las cosas mutables solo por un periodo corto de tiempo:

```
struct Punto {
    x: i32,
    y: i32,
}

fn main() {
    let mut punto = Punto { x: 0, y: 0 };

    punto.x = 5;

    let punto = punto; // ahora, este nuevo enlace a variable no puede ser cambiado

    punto.y = 6; // esto causa un error
}
```

Sintaxis de actualización

Una `struct` puede incluir `..` para indicar que deseas usar una copia de algún otro `struct` para algunos de los valores. Por ejemplo:

```
struct Punto3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut punto = Punto3d { x: 0, y: 0, z: 0 };
punto = Punto3d { y: 1, .. punto };
```

Lo anterior, asigna a `punto` un nuevo `y`, pero mantiene los antiguos valores de `x` y `z`. Tampoco tiene que ser la misma estructura, puedes hacer uso de esta sintaxis cuando creas nuevas, y esta copiara los valores que no especifiques:

```
# struct Punto3d {
#     x: i32,
#     y: i32,
#     z: i32,
# }
let origen = Punto3d { x: 0, y: 0, z: 0 };
let punto = Punto3d { z: 1, x: 2, .. origen };
```

Tupla estructuras (Tuple structs)

Rust posee otro tipo de datos que es como un híbrido entre una tupla y una `struct`, llamado `tulla estructura` (`tuple struct`). Las tupla estructuras poseen un nombre, pero sus campos no:

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

Las dos anteriores no serán iguales, incluso si poseen los mismos valores:

```
# struct Color(i32, i32, i32);
# struct Punto(i32, i32, i32);
let negro = Color(0, 0, 0);
let origen = Punto(0, 0, 0);
```

Casi siempre es mejor usar una `struct` que una `tuple struct`. En su lugar, podríamos escribir `Color` and `Punto` así:

```
struct Color {
    rojo: i32,
    azul: i32,
    verde: i32,
}

struct Punto {
    x: i32,
    y: i32,
    z: i32,
}
```

Ahora, tenemos nombres, en lugar de posiciones. Los buenos nombres son importantes, y con una `struct`, tenemos nombres.

Hay un caso en el cual una tupla estructura es muy útil, y es una tupla estructura con un solo elemento. Se denomina patrón `nuevotipo` (`newtype`), puesto a que crear un nuevo tipo, distinto a el valor que contiene, expresando su propia semántica:

```
struct Pulgadas(i32);

let longitud = Pulgadas(10);

let Pulgadas(longitud_enteros) = longitud;
println!("la longitud es {} pulgadas", longitud_enteros);
```

Como notaras, puedes extraer el entero contenido con un `let` de destructuración, justo como en las tuplas regulares. En este caso, el `let Pulgadas(longitud_enteros)` asigna `10` a `longitud_enteros`.

Estructuras tipo-unitario (unit-like structs)

Puedes definir una `struct` sin ningún miembro:

```
struct Electron;
```

Dicha `struct` es denominada `tipo-unitario` (`unit-like`) por su semejanza a la tupla vacía, `()`, algunas veces llamada `unidad` (`unit`). Como una tupla estructura, define un nuevo tipo.

Lo anterior es raramente util en si mismo (aunque algunas veces puede servir como un tipo marcador), pero en combinación con otras características, puede tornarse útil. Por ejemplo, una librería puede requerir crear una estructura que implementa cierto trait para manejar

eventos. De no poseer ninguna data para guardar en la estructura, simplemente puedes crear una `struct` tipo-unitario.

% Enumeraciones

Una enumeración (`enum`) en Rust es un tipo que representa data que puede ser una de un conjunto de variantes posible:

```
enum Mensaje {
    Salir,
    CambiarColor(i32, i32, i32),
    Mover { x: i32, y: i32 },
    Escribir(String),
}
```

Cada variante puede opcionalmente tener data asociada. La sintaxis para la definición de variantes es semejante a la sintaxis usada para definir estructuras (`struct` s): puedes tener variantes sin datos (como las estructuras tipo-unitario), variantes con data nombrada, y variantes con data sin nombres (como tupla estructuras). Sin embargo, a diferencia de las definiciones de estructuras, un `enum` es un único tipo. Un valor de un enum puede coincidir con cualquiera de las variantes. Por esta razón, un enum es denominado algunas veces un 'tipo suma' ('sum type'): el conjunto de valores posibles del enum es la suma de los conjuntos de valores posibles para cada variante.

Utilizamos la sintaxis `::` para hacer uso de cada variante: las variantes están dentro del ámbito del `enum` . Lo que hace que lo siguiente sea valido:

```
# enum Mensaje {
#     Mover { x: i32, y: i32 },
# }
let x: Mensaje = Mensaje::Mover { x: 3, y: 4 };

enum TurnoJuegoMesa {
    Mover { celdas: i32 },
    Pasar,
}

let y: TurnoJuegoMesa = TurnoJuegoMesa::Mover { celdas: 1 };
```

Ambas variantes poseen el nombre `Mover` , pero debido a que su ámbito es dentro del nombre del enum, ambas pueden ser usadas sin conflicto.

Un valor de un tipo enum contiene información acerca de cual variante es, en adición a cualquier data asociada con dicha variante. Esto es algunas veces denominado 'union etiquetada' ('tagged union'), puesto a que la data incluye una 'etiqueta' ('tag') que indica que tipo es. El compilador usa esta información para asegurarse que estemos accediendo a la data en el enum de manera segura. Por ejemplo, no puedes simplemente tratar de destructurar un valor como si este fuere una de las posibles variantes:

```
fn procesar_cambio_color(msj: Mensaje) {
    let Mensaje::CambiarColor(r, v, a) = msj; // compile-time error
}
```

No soportar este tipo de operaciones puede lucir un poco limitante, pero es una limitación que podemos superar. Existen dos maneras, implementando la igualdad por nuestra cuenta, o a través de el pattern matching con expresiones `match`, que aprenderás en la siguiente sección. Todavía no sabemos lo suficiente de Rust para implementar la igualdad por nuestra cuenta, pero lo veremos en la sección `traits`.

Constructores como funciones

Un constructor de un enum puede ser también usado como una función. Por ejemplo:

```
# enum Mensaje {
# Escribir(String),
# }
let m = Mensaje::Escribir("Hola, mundo".to_string());
```

Es lo mismo que

```
# enum Mensaje {
# Escribir(String),
# }
fn foo(x: String) -> Mensaje {
    Mensaje::Escribir(x)
}

let x = foo("Hola, mundo".to_string());
```

Esto no es inmediatamente util para nosotros, pero cuando lleguemos a los `closures`, hablaremos acerca de pasar funciones como argumentos a otras funciones. Por ejemplo, con los iteradores, podemos convertir un vector de `String`s en un vector de `Message::Escribir`s:

```
# enum Mensaje {
# Escribir(String),
# }

let v = vec!["Hola".to_string(), "Mundo".to_string()];

let v1: Vec<Mensaje> = v.into_iter().map(Mensaje::Escribir).collect();
```

% Match

A menudo, un simple `if` / `else` no es suficiente, debido a que tienes mas de dos opciones posibles. También, las condiciones pueden ser complejas. Rust posee una palabra reservada, `match`, que te permite reemplazar construcciones `if` / `else` complicadas con algo mas poderoso. Echa un vistazo:

```
let x = 5;

match x {
    1 => println!("uno"),
    2 => println!("dos"),
    3 => println!("trees"),
    4 => println!("cuatro"),
    5 => println!("cinco"),
    _ => println!("otra cosa"),
}
```

`match` toma una expresión y luego bifurca basado en su valor. Cada `brazo` de la rama tiene la forma `valor => expresión`. Cuando el valor coincide, la expresión del brazo es evaluada. Es llamado `match` por el termino 'pattern matching', del cual `match` es una implementación. Hay una sección entera acerca de los patrones que cubre todos los patrones posibles.

Entonces, cual es la gran ventaja? Bueno, hay unas pocas. Primero que todo, `match` impone `chequeo de agotamiento` ('exhaustiveness checking'). Ves el ultimo brazo, el que posee el guión bajo (`_`)? Si removemos ese brazo, Rust nos proporcionara un error:

```
error: non-exhaustive patterns: `_` not covered
```

En otras palabras, Rust esta tratando de decirnos que olvidamos un valor. Debido a que `x` es un entero, Rust sabe que `x` puede tener un numero de valores diferentes - por ejemplo, `6`. Sin el `_`, sin embargo, no hay brazo que pueda coincidir, y en consecuencia Rust se rehusa a compilar el código. `_` actúa como un 'brazo que atrapa todo'. Si ninguno de los otros brazos coincide, el brazo con el `_` lo hará, y puesto que tenemos dicho 'brazo que atrapa todo', tenemos ahora un brazo para cada valor posible de `x`, y como resultado, nuestro programa compilara satisfactoriamente.

`match` es también una expresión, lo que significa que lo podemos usar en el lado derecho de un `let` o directamente en donde una expresión sea usada:

```
let x = 5;

let number = match x {
    1 => "uno",
    2 => "dos",
    3 => "tres",
    4 => "cuatro",
    5 => "cinco",
    _ => "otra cosa",
};
```

Algunas veces es una buena forma de convertir algo de un tipo a otro.

Haciendo `match` en enums

Otro uso importante de la palabra reservada `match` es para procesar las posibles variantes de una enum:

```
enum Mensaje {
    Salir,
    CambiarColor(i32, i32, i32),
    Mover { x: i32, y: i32 },
    Escribir(String),
}

fn salir() { /* ... */ }
fn cambiar_color(r: i32, g: i32, b: i32) { /* ... */ }
fn mover_cursor(x: i32, y: i32) { /* ... */ }

fn procesar_mensaje(msj: Mensaje) {
    match msj {
        Mensaje::Salir => salir(),
        Mensaje::CambiarColor(r, g, b) => cambiar_color(r, g, b),
        Mensaje::Mover { x: x, y: y } => mover_cursor(x, y),
        Mensaje::Escribir(s) => println!("{}", s),
    };
}
```

Otra vez, el compilador de Rust chequea agotamiento, demandando que tengas un brazo para cada variante de la enumeración. Si dejas uno por fuera, Rust generara un error en tiempo de compilación, a menos que uses `_`.

A diferencia de los usos previos de `match`, no puedes usar de la sentencia `if` para hacer esto. Puedes hacer uso de una sentencia `if let` que puede ser vista como una forma abreviada de `match`.

% Patrones

Los patrones son bastante comunes en Rust. Los usamos en enlaces a variable, sentencias match, y otros casos. Embarquemonos en un tour torbellino por todas las cosas que los patrones son capaces de hacer!

Un repaso rápido: puedes probar patrones contra literales directamente, y `_` actúa como un caso `cualquiera`:

```
let x = 1;

match x {
    1 => println!("uno"),
    2 => println!("dos"),
    3 => println!("tres"),
    _ => println!("cualquiera"),
}
```

Imprime `uno`.

Multiples patrones

Puedes probar multiples patrones con `|`:

```
let x = 1;

match x {
    1 | 2 => println!("uno o dos"),
    3 => println!("tres"),
    _ => println!("cualquiera"),
}
```

Lo anterior imprime `uno o dos`.

Destructuracion

Si posees un tipo de datos compuesto, como un `struct`, puedes destructurarlo dentro de un patron:

```
struct Punto {
    x: i32,
    y: i32,
}

let origen = Punto { x: 0, y: 0 };

match origen {
    Punto { x, y } => println!("({},{})", x, y),
}
```

Puedes usar `:` para darle un nombre diferente a un valor.

```
struct Punto {
    x: i32,
    y: i32,
}

let origen = Punto { x: 0, y: 0 };

match origen {
    Punto { x: x1, y: y1 } => println!("({},{})", x1, y1),
}
```

Si solo nos importan algunos valores, no tenemos que darle nombres a todos:

```
struct Punto {
    x: i32,
    y: i32,
}

let origen = Punto { x: 0, y: 0 };

match origen {
    Punto { x, .. } => println!("x es {}", x),
}
```

Esto imprime `x es 0`.

Puedes hacer este tipo de pruebas en cualquier miembro, no solo el primero:

```
struct Punto {
    x: i32,
    y: i32,
}

let origen = Punto { x: 0, y: 0 };

match origen {
    Punto { y, .. } => println!("y es {}", y),
}
```

Lo anterior imprime `y es 0`.

Este comportamiento de 'destructuracion' funciona en cualquier tipo de datos compuesto, como tuplas o enums.

Ignorando enlaces a variables

Puedes usar `_` en un patron para ignorar tanto el tipo como el valor.

Por ejemplo, he aquí un `match` contra un `Result<T, E>`:

```
# let algun_valor: Result<i32, &'static str> = Err("Hubo un error");
match algun_valor {
    Ok(valor) => println!("valor obtenido: {}", valor),
    Err(_) => println!("ha ocurrido un error"),
}
```

En el primer brazo, enlazamos el valor dentro de la variante `Ok` a la variable `valor`. Pero en el brazo `Err` usamos `_` para ignorar el error especifico, y solo imprimir un mensaje de error general.

`_` es valido en cualquier patron que cree un enlace a variable. También puede ser util para ignorar porciones de una estructura mas grande:

```
fn coordenada() -> (i32, i32, i32) {
    // generar y retornar algún tipo de tupla de tres elementos
# (1, 2, 3)
}

let (x, _, z) = coordenada();
```

Aquí, asociamos ambos el primer y ultimo elemento de la tupla a `x` y `z` respectivamente, ignorando el elemento de la mitad.

Similarmente, puedes usar `..` en un patrón para ignorar multiples valores.

```
enum TuplaOpcional {
    Valor(i32, i32, i32),
    Faltante,
}

let x = TuplaOpcional::Valor(5, -2, 3);

match x {
    TuplaOpcional::Valor(..) => println!("Tupla obtenida!"),
    TuplaOpcional::Faltante => println!("Sin suerte."),
}
```

Esto imprime `Tupla obtenida!`.

ref y ref mut

Si deseas obtener una referencia, debes usar la palabra reservada `ref`:

```
let x = 5;

match x {
    ref r => println!("Referencia a {} obtenida", r),
}
```

Imprime `Referencia a 5 obtenida`.

Acá, la `r` dentro del `match` posee el tipo `&i32`. En otras palabras la palabra reservada `ref` *crea* una referencia, para ser usada dentro del patrón. Si necesitas una referencia mutable `ref mut` funcionara de la misma manera:

```
let mut x = 5;

match x {
    ref mut rm => println!("Referencia mutable a {} obtenida", rm),
}
```

Rangos

Puedes probar un rango de valors con `...`:

```
let x = 1;

match x {
    1 ... 5 => println!("uno al cinco"),
    _ => println!("cualquier cosa"),
}
```

Esto imprime `uno al cinco`.

Los rangos son usados mayormente con enteros y `chars` s:

```
let x = '';

match x {
    'a' ... 'j' => println!("letra temprana"),
    'k' ... 'z' => println!("letra tardia"),
    _ => println!("algo mas"),
}
```

This prints `algo mas`.

Enlaces a variable

Puedes asociar valores a nombres con `@`:

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("valor de rango {} obtenido", e),
    _ => println!("lo que sea"),
}
```

This prints `valor de rango 1 obtenido`. Lo anterior es util cuando desees hacer un match complicado a una parte de una estructura de datos:

```
#[derive(Debug)]
struct Persona {
    nombre: Option<String>,
}

let nombre = "Steve".to_string();
let mut x: Option<Persona> = Some(Persona { nombre: Some(nombre) });
match x {
    Some(Persona { nombre: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

Dicho código imprime `Some("Steve")` : hemos asociado el `nombre` interno a `a` .

Si usas `@` con `|` , necesitas asegurarte de que el nombre sea asociado en cada parte del patron:

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("valor de rango {} obtenido", e),
    _ => println!("lo que sea"),
}
```

Guardias

Puedes introducir `guardias match` (‘match guards’) con `if` :

```
enum EnteroOpcional {
    Valor(i32),
    Faltante,
}

let x = EnteroOpcional::Value(5);

match x {
    EnteroOpcional::Valor(i) if i > 5 => println!("Entero mayor a cinco obtenido!"),
    EnteroOpcional::Valor(..) => println!("Entero obtenido!"),
    EnteroOpcional::Faltante => println!("Sin suerte."),
}
```

Esto imprime `Entero obtenido!"` .

Si estas usando `if` con multiples patrones, el `if` aplica a ambos lados:

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("si"),
    _ => println!("no"),
}
```

Lo anterior imprime `no`, debido a que el `if` aplica a el `4 | 5` completo, y no solo al `5`. En otras palabras, la precedencia del `if` se comporta de la siguiente manera:

```
(4 | 5) if y => ...
```

y no así:

```
4 | (5 if y) => ...
```

Mezcla y Match

Uff! Eso fue un montón de formas diferentes para probar cosas, y todas pueden ser mezcladas y probadas, dependiendo de los que estés haciendo:

```
match x {
    Foo { x: Some(ref nombre), y: None } => ...
}
```

Los patrones son muy poderosos. Haz buen uso de ellos.

% Sintaxis de Métodos

Las funciones son geniales, pero si deseas llamar bastantes en alguna data, puede tornarse incomodo. Considera este código:

```
baz(bar(foo));
```

Pudiéramos leer esto de izquierda a derecha, veríamos entonces ‘baz bar foo’. Pero ese no es el orden en el cual las funciones son llamadas, el orden de hecho, es de adentro hacia afuera: ‘foo bar baz’. No seria excelente si pudiéramos hacerlo:

```
foo.bar().baz();
```

Por suerte, como habrás podido deducir de la pregunta anterior, por supuesto que podemos! Rust provee la habilidad de usar la denominada ‘sintaxis de llamada a métodos’ (‘method call syntax’) a través de la palabra reservada `impl`.

Llamadas a métodos

Así funcionan:

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl Circulo {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radio * self.radio)
    }
}

fn main() {
    let c = Circulo { x: 0.0, y: 0.0, radio: 2.0 };
    println!("{}", c.area());
}
```

Lo anterior imprimira `12.566371`.

Hemos construido una `struct` representando a un circulo. Después escribimos un bloque `impl` y dentro de el, definimos un método, `area`.

Los métodos reciben un primer parámetro especial, del cual existen tres variantes: `self`, `&self`, y `&mut self`. Puedes pensar acerca de este primer parámetro como el `foo` en `foo.bar()`. Las tres variantes corresponden a los tres tipos de cosas que `foo` podría ser: `self` si es solo un valor en la pila, `&self` si es una referencia y `&mut self` si es una referencia mutable. Debido a que proporcionamos el parámetro `&self` a `area`, podemos usarlo justo como cualquier otro. Como conocemos que es un `Circulo`, podemos acceder a `radio` como lo haríamos con cualquier otra `struct`.

Deberíamos usar por defecto `&self`, así como preferir también el préstamo por encima de la toma de pertenencia y recibir referencias inmutables en vez de mutables, en lo posible. He aquí un ejemplo de las tres variantes:

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl Circle {
    fn referencia(&self) {
       println!("recibiendo a self como una referencia!");
    }

    fn referencia_mutable(&mut self) {
       println!("trecibiendo a self como una referencia mutable!");
    }

    fn toma_pertenecia(self) {
       println!("tomando pertenencia de self!");
    }
}
```

Llamadas a métodos en cadena

Entonces, ahora sabemos como llamar a un método, como `foo.bar()`. Pero que hay acerca de nuestro ejemplo original, `foo.bar().baz()`? Se denomina ‘encadenamiento de métodos’ (‘method chaining’). Veamos un ejemplo:

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl Circulo {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radio * self.radio)
    }

    fn agrandar(&self, incremento: f64) -> Circulo {
        Circulo { x: self.x, y: self.y, radio: self.radio + incremento }
    }
}

fn main() {
    let c = Circulo { x: 0.0, y: 0.0, radio: 2.0 };
    println!("{}", c.area());

    let d = c.agrandar(2.0).area();
    println!("{}", d);
}
```

Observa el valor de retorno:

```
# struct Circulo;
# impl Circulo {
fn agrandar(&self, incremento: f64) -> Circulo {
# Circulo } }
```

Solo decimos que retornamos un `Circulo`. Con este método, podemos agrandar un nuevo `Circulo` a un tamaño arbitrario.

Funciones asociadas

Puedes también definir funciones asociadas que no tomen el parámetro `self`. He aquí un patron muy común en código Rust:

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl Circulo {
    fn new(x: f64, y: f64, radio: f64) -> Circulo {
        Circulo {
            x: x,
            y: y,
            radio: radio,
        }
    }
}

fn main() {
    let c = Circulo::new(0.0, 0.0, 2.0);
}
```

Esta 'función asociada' construye un nuevo `Circulo`. Nota que las funciones asociadas son llamadas con la sintaxis `Struct::funcion()`, en lugar de `ref.metodo()`. Otros lenguajes llaman a las funciones asociadas 'métodos estáticos'

El patrón Constructor (Builder)

Digamos que queremos que nuestros usuarios puedan crear `Circulo`s, pero solo les permitiremos dar valores a las propiedades que sean relevantes para ellos. De lo contrario, los atributos `x` y `y` serán `0.0`, y el radio sera `1.0`. Rust no posee sobrecarga de métodos, argumentos con nombre o argumentos variables. Se emplea el patron builder en su lugar. Dicho patron luce así:

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl Circulo {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radio * self.radio)
    }
}

struct ConstructorCirculo {
    x: f64,
```

```
    y: f64,
    radio: f64,
}

impl ConstructorCirculo {
    fn new() -> ConstructorCirculo {
        ConstructorCirculo { x: 0.0, y: 0.0, radio: 1.0, }
    }

    fn x(&mut self, coordenada: f64) -> &mut ConstructorCirculo {
        self.x = coordenada;
        self
    }

    fn y(&mut self, coordenada: f64) -> &mut ConstructorCirculo {
        self.y = coordenada;
        self
    }

    fn radio(&mut self, radio: f64) -> &mut ConstructorCirculo {
        self.radio = radio;
        self
    }

    fn finalizar(&self) -> Circulo {
        Circulo { x: self.x, y: self.y, radio: self.radio }
    }
}

fn main() {
    let c = ConstructorCirculo::new()
                .x(1.0)
                .y(2.0)
                .radio(2.0)
                .finalizar();

    println!("area: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

Lo que hemos hecho es crear otra `struct` , `ConstructorCirculo` . Hemos definido nuestros métodos constructores en ella. También hemos definido nuestro método `area()` en `Circulo` . Hemos agregado otro método en `ConstructorCirculo` : `finalizar()` . Este método crea nuestro `Circulo` desde el constructor. Ahora, hemos usado el sistema de tipos para hacer valer nuestros intereses: podemos usar los métodos en `ConstructorCirculo` para crear `Circulo` s en la forma que decidamos.

% Vectores

Un 'vector' es un arreglo dinámico, implementado como el tipo de la biblioteca estándar `Vec<T>`. La `T` significa que podemos tener vectores de cualquier tipo (echa un vistazo al capitulo de [genéricos][generics] para mas información). Los vectores siempre alojan sus datos en el montículo. Puedes crear vectores con la macro `vec!`:

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

(Nota que a diferencia de la macro `println!` que hemos usado en el pasado, usamos corchetes `[]` con la macro `vec!`. Rust te permite usar cualquiera en cualquier situación, en esta oportunidad es por pura convención)

Existe una forma alternativa de `vec!` para repetir un valor inicial:

```
let v = vec![0; 10]; // diez ceros
```

Accediendo a elementos

Para obtener el valor en un indice en particular del vector, usamos `[]` s:

```
let v = vec![1, 2, 3, 4, 5];

println!("El tercer elemento de v es {}", v[2]);
```

Los indices comienzan desde `0`, entonces, el tercer elemento es `v[2]`.

Iterando

Una vez poseas un vector, puedes iterar a través de sus elementos con `for`. Hay tres versiones:

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("Una referencia a {}", i);
}

for i in &mut v {
    println!("Una referencia mutable a {}", i);
}

for i in v {
    println!("Tomando pertenencia del vector y su elemento {}", i);
}
```

Los vectores poseen muchos otros métodos útiles, acerca de los cuales puedes leer en su documentation

% Cadenas de Caracteres

Las cadenas de caracteres son un concepto importante a dominar para cualquier programador. El sistema de manejo de cadenas de caracteres en Rust es un poco distinto al de otros lenguajes, debido a su foco en programación de sistemas. Siempre que poseas una estructura de datos de tamaño variable, las cosas pueden ponerse un poco difíciles, y las cadenas de caracteres son una estructura de datos que puede variar en tamaño. Dicho esto, las cadenas de caracteres de Rust también funcionan de manera diferente que en algunos otros lenguajes de programación de sistemas, como C.

Entremos en los detalles. Una ‘cadena de caracteres’ (‘string’) es una secuencia de valores escalares Unicode codificada como un flujo de bytes UTF-8. Todas las cadenas de caracteres están garantizadas a ser una codificación valida de secuencias UTF-8. Adicionalmente, y a diferencia de otros lenguajes de sistemas, las cadenas de caracteres no son terminadas en null y pueden contener bytes null.

Rust posee dos tipos principales de cadenas de caracteres: `&str` y `String`. Hablemos primero acerca de `&str`. Estos son denominados ‘pedazos de cadenas de caracteres’ (‘string slices’). Los literales String son del tipo &'static str`:

```
let saludo = "Hola."; // saludo: &'static str
```

Esta cadena de caracteres es asignada estáticamente, significando que es almacenada dentro de nuestro programa compilado, y existe por la duración completa de su ejecución. El enlace `saludo` es una referencia a una cadena asignada estéticamente. Los pedazos de cadenas de caracteres poseen un tamaño fijo, y no pueden ser mutados.

Por otro lado, un `String`, es una cadena de caracteres asignada desde el montículo. Dicha cadena puede crecer, y también esta garantizada ser UTF-8. Los `String` son creados comúnmente a través de la conversión de un pedazo de cadena de carácter usando el método `to_string`.

```
let mut s = "Hola".to_string(); // mut s: String
println!("{}", s);

s.push_str(", mundo.");
println!("{}", s);
```

Los `String`s haran coercion a un `&str` con un `&`:

```
fn recibe_pedazo(pedazo: &str) {
    println!("Recibí: {}", pedazo);
}

fn main() {
    let s = "Hola".to_string();
    recibe_pedazo(&s);
}
```

Esta coerción no ocurre para las funciones que aceptan uno de los traits `&str` 's en lugar de `&str` . Por ejemplo, `TcpStream::connect` posee un parámetro de tipo `ToSocketAddrs` . Un `&str` esta bien pero un `String` debe ser explícitamente convertido usando `&*` .

```
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // parametro &str

let cadena_direccion = "192.168.0.1:3000".to_string();
TcpStream::connect(&*cadena_direccion); // convirtiendo cadena_direccion a &str
```

Ver un `String` como un `&str` es barato, pero convertir el `&str` a un `String` involucra asignación de memoria. No hay razón para hacer eso a menos que sea necesario!

Indexado

Debido a que las cadenas de caracteres son UTF-8 validos, no soportan indexado:

```
let s = "hola";

println!("La primera letra de s es {}", s[0]); // ERROR!!!
```

Usualmente, el acceso a un vector con `[]` es muy rápido. Pero, puesto a que cada carácter codificado en una cadena UTF-8 puede tener multiples bytes, debes recorrer toda la cadena para encontrar la n^{th} letra de una cadena. Esta es una operación significativamente mas cara, y no queremos generar confusiones. Incluso, 'letra' no es exactamente algo definido en Unicode. Podemos escoger ver a una cadena de caracteres como bytes individuales, o como codepoints:

```
let hachiko = "忠犬ハチ公";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

Lo anterior imprime:

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
忠, 犬, ハ, チ, 公,
```

Como puedes ver, hay mas bytes que caracteres (`char` s).

Puedes obtener algo similar a un indice de esta forma:

```
# let hachiko = "忠犬ハチ公";
let perro = hachiko.chars().nth(1); // algo como hachiko[1]
```

Esto enfatiza que tenemos que caminar desde el principio de la lista de `chars` .

Cortado (Slicing)

Puedes obtener un pedazo de una cadena de caracteres con la sintaxis de cortado:

```
let perro = "hachiko";
let hachi = &perro[0..5];
```

Pero nota que estos son desplazamientos de *byte*, no desplazamientos de *character*. Entonces, lo siguiente fallara en tiempo de ejecución:

```
let perro = "忠犬ハチ公";
let hachi = &perro[0..2];
```

con este error:

```
thread '
' panicked at 'index 0 and/or 2 in `忠犬ハチ公` do not lie on
character boundary'
```

Concatenación

Si posees un `String`, puedes concatenarle un `&str` al final:

```
let hola = "Hola ".to_string();
let mundo = "mundo!";

let hola_mundo = hola + mundo;
```

Pero si tienes dos `String`s, necesitas un `&`:

```
let hola = "Hola ".to_string();
let mundo = "mundo!".to_string();

let hola_mundo = hola + &world;
```

Esto es porque `&String` puede hacer coercion automática a un `&str`. Esta característica es denominada 'coerciones `Deref`'.

% Genéricos

Algunas veces, cuando se escribe una función o una estructura de datos, podríamos desear que esta pudiese funcionar con multiples tipos de argumentos. En Rust podemos lograr esto a través de los genéricos. Los genéricos son llamados 'polimorfismo parametrico' en la teoría de tipos, lo que significa que son tipos o funciones que poseen multiples formas ('poly' de multiple, 'morph' de forma) sobre un determinado parámetro ('parametrico').

De cualquier modo, suficiente acerca de teoría de tipos, veamos un poco de código genérico. La biblioteca estándar de Rust provee un tipo, `Option<T>`, el cual es genérico:

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

La parte `<T>`, la cual has visto unas cuantas veces anteriormente, indica que este es un tipo de datos genérico. Dentro de la declaración de nuestro enum, donde sea que veamos una `T` sustituimos ese tipo por el mismo tipo usado en el genérico. He aquí un ejemplo del uso de `Option<T>`, con algunos anotaciones de tipo extra:

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

En la declaración del tipo, decimos `Option<i32>`. Nota cuan similar esto luce esto a `Option<T>`. Entonces, en este `Option`, `T` posee el valor de `i32`. En el lado derecho del binding, hacemos un `Some(T)`, en donde `T` es `5`. Debido a que `5` es un `i32`, ambos lados coinciden, y Rust es feliz. De no haber coincidido, hubiésemos obtenido un error:

```
let x: Option = Some(5);
// error: mismatched types: expected `core::option::Option`,
// found `core::option::Option<_>` (expected f64 but found integral variable)
```

Eso no significa que no podamos crear `Option<T>` s que contengan un `f64`. Simplemente deben coincidir:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

Muy bueno. Una definición, multiples usos.

Los genéricos no deben necesariamente ser genéricos sobre un solo tipo. Considera otro tipo similar en la biblioteca estándar de Rust, `Result<T, E>`:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Este tipo es genérico sobre *dos* tipos: `T` y `E`. Por cierto, las letras mayusculas pueden ser cualquiera. Pudimos haber definido `Result<T, E>` como:

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

de haber querido. La convención dice que el primer parametro generico debe ser `T`, de 'tipo', y que usemos `E` para 'error'. A Rust, sin embargo no le importa.

El tipo `Result<T, E>` se usa para retornar el resultado de una computación, con la posibilidad de retornar un error en caso de que dicha computación no haya sido exitosa.

Funciones genéricas

Podemos escribir funciones que tomen tipos genéricos con una sintaxis similar:

```
fn recibe_cualquier_cosa<T>(x: T) {
    // hacer algo con x
}
```

La sintaxis posee dos partes: el `<T>` dice "esta función es generica sobre un tipo, `T`", y la parte `x: T` dice "x posee el tipo `T`."

Multiples argumentos pueden tener el mismo tipo:

```
fn recibe_dos_cosas_del_mismo_tipo<T>(x: T, y: T) {
    // ...
}
```

Pudimos haber escrito una version que recibe multiples tipos:

```
fn recibe_dos_cosas_de_distintos_tipos<T, U>(x: T, y: U) {
    // ...
}
```

Structs genericos

También puedes almacenar un tipo genérico en una estructura:

```
struct Punto<T> {
    x: T,
    y: T,
}

let origen_entero = Punto { x: 0, y: 0 };
let origen_flotante = Punto { x: 0.0, y: 0.0 };
```

Similarmente a las funciones, la sección `<T>` es en donde declaramos los parámetros genéricos, después de ello hacemos uso de `x: T` en la declaración del tipo, también.

Cuando deseamos agregar una implementación para la estructura genérica, simplemente declaras el parámetro de tipo después de `impl` :

```
# struct Punto<T> {
#     x: T,
#     y: T,
# }
#
impl<T> Punto<T> {
    fn intercambiar(&mut self) {
        std::mem::swap(&mut self.x, &mut self.y);
    }
}
```

Hasta ahora solo has visto genéricos que aceptan absolutamente cualquier tipo. Estas son útiles en muchos casos, ya has visto `Option<T>`, y mas tarde conoceras contenedores universales como `Vec<T>`. Por otro lado, a veces querrás intercambiar esa flexibilidad por mayor poder expresivo. Lee acerca de limites trait para ver porque y como.

% Traits

Un trait es una facilidad del lenguaje que le indica compilador de Rust acerca de la funcionalidad que un tipo debe proveer.

Recuerdas la palabra reservada `impl` ?, usada para llamar a una función con la sintaxis de métodos?

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl Circulo {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radio * self.radio)
    }
}
```

Los traits son similares, excepto que definimos un trait con solo la firma de método y luego implementamos el trait para la estructura. Así:

```
struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

trait TieneArea {
    fn area(&self) -> f64;
}

impl TieneArea for Circulo {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radio * self.radio)
    }
}
```

Como puedes ver, el bloque `trait` luce muy similar a el bloque `impl` , pero no definimos un bloque, solo la firma de tipos. Cuando implementamos un trait, usamos `impl Trait for Item` , en vez de solo `impl Item` .

Limites trait para funciones genéricas

Los traits son útiles porque permiten a un tipo hacer ciertas promesas acerca de su comportamiento. La funciones genéricas pueden explotar esto para restringir los tipos que aceptan. Considera esta función, la cual no compila:

```
fn imrimir_area(figura: T) {
    println!("Esta figura tiene un area de {}", figura.area());
}
```

Rust se queja:

```
error: no method named `area` found for type `T` in the current scope
```

Debido a que `T` puede ser de cualquier tipo, no podemos estar seguros que implementa el método `area`. Pero podemos agregar una 'restricción de trait' a nuestro `T` genérico, asegurándonos de que lo implemente:

```
# trait TieneArea {
#     fn area(&self) -> f64;
# }
fn imrimir_area<T: TieneArea>(shape: T) {
    println!("Esta figura tiene un area de {}", figura.area());
}
```

La sintaxis `<T: TieneArea>` se traduce en "cualquier tipo que implemente el trait `TieneArea`.". A consecuencia de que los traits definen firmas de tipos de función, podemos estar seguros que cualquier tipo que implemente `TieneArea` tendrá un método `.area()`.

He aquí un ejemplo extendido de como esto funciona:

```
trait TieneArea {
    fn area(&self) -> f64;
}

struct Circulo {
    x: f64,
    y: f64,
    radio: f64,
}

impl TieneArea for Circulo {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radio * self.radio)
    }
}

struct Cuadrado {
    x: f64,
    y: f64,
    lado: f64,
}

impl TieneArea for Cuadrado {
    fn area(&self) -> f64 {
        self.lado * self.lado
    }
}

fn imrimir_area<T: TieneArea>(figura: T) {
    println!("Esta figura tiene un area de {}", figura.area());
}

fn main() {
    let c = Circulo {
        x: 0.0f64,
        y: 0.0f64,
        radio: 1.0f64,
    };

    let s = Cuadrado {
        x: 0.0f64,
        y: 0.0f64,
        lado: 1.0f64,
    };

    imrimir_area(c);
    imrimir_area(s);
}
```

Este programa produce la salida:

```
Esta figura tiene un area de 3.141593
Esta figura tiene un area de 1
```

Como puedes ver, `imrimir_area` ahora es genérica, pero también asegura que hallamos proporcionado los tipos correctos. Si pasamos un tipo incorrecto:

```
imrimir_area(5);
```

Obtenemos un error en tiempo de compilación:

```
error: the trait `TieneArea` is not implemented for the type `_` [E0277]
```

Limites de trait para estructuras genericas

Tus estructuras genéricas pueden beneficiarse también de las restricciones de trait. Todo lo que necesitas es agregar la restricción cuando declaras los parámetros de tipos. A continuación un nuevo tipo `Rectangulo<T>` y su operación `es_cuadrado`:

```
struct Rectangulo<T> {
    x: T,
    y: T,
    ancho: T,
    altura: T,
}

impl<T: PartialEq> Rectangulo<T> {
    fn es_cuadrado(&self) -> bool {
        self.ancho == self.altura
    }
}

fn main() {
    let mut r = Rectangulo {
        x: 0,
        y: 0,
        ancho: 47,
        altura: 47,
    };

    assert!(r.es_cuadrado());

    r.altura = 42;
    assert!(!r.es_cuadrado());
}
```

`es_cuadrado()` necesita chequear que los lados son iguales, y para ello los tipos deben ser de un tipo que implemente el trait `core::cmp::PartialEq`:

```
impl Rectangulo { ... }
```

Ahora, un rectángulo puede ser definido en función de cualquier tipo que pueda ser comparado por igualdad.

Hemos definido una nueva estructura `Rectangulo` que acepta números de cualquier precision, objetos de cualquier tipo siempre y cuando puedan ser comparados por igualdad. Podríamos hacer lo mismo para nuestras estructuras `TieneArea`, `Cuadrado` y `Circulo`? Si, pero estas necesitan multiplicación, y para trabajar con eso necesitamos saber mas de los traits de operadores.

Reglas para la implementación de traits

Hasta ahora, solo hemos agregado implementaciones de traits a estructuras, pero puedes implementar cualquier trait para cualquier tipo. Técnicamente, *podriamos* implementar `TieneArea` para `i32`:

```
trait TieneArea {
    fn area(&self) -> f64;
}

impl TieneArea for i32 {
    fn area(&self) -> f64 {
        println!("esto es tonto");

        *self as f64
    }
}

5.area();
```

Se considera pobre estilo implementar métodos en esos tipos primitivos, aun cuando es posible.

Esto puede lucir como el viejo oeste, pero hay dos restricciones acerca de la implementación de traits que previenen que las cosas se salgan de control. La primera es que si el trait no esta definido en tu ámbito, no aplica. He aquí un ejemplo: la biblioteca estándar provee un trait `Write` que agrega funcionalidad extra a los `File`s, posibilitando la E/S de archivos. Por defecto, un `File` no tendría sus métodos:

```
let mut f = std::fs::File::open("foo.txt").ok().expect("No se pudo abrir foo.txt");
let buf = b"cualquier cosa"; // literal de cadena de bytes. buf: &[u8; 8]
let resultado = f.write(buf);
# resultado.unwrap(); // ignorar el error
```

He aqui el error:

```
error: type `std::fs::File` does not implement any method in scope named `write`
let resultado = f.write(buf);
                ^~~~~~~~~~
```

Necesitamos primero hacer `use` del trait `Write`:

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

let mut f = std::fs::File::open("foo.txt").ok().expect("No se pudo abrir foo.txt");
let buf = b"cualquier cosa";
let resultado = f.write(buf);
# resultado.unwrap(); // ignorar el error
```

Lo anterior compilara sin errores.

Esto significa que incluso si alguien hace algo malo como agregar métodos a `i32`, no te afectara, a menos que hagas `use` de ese trait.

Hay una restricción mas acerca de la implementación de traits: uno de los dos bien sea el trait o el tipo para el cual estas escribiendo la `impl`, debe ser definido por ti. Entonces, podríamos implementar el trait `TieneArea` para el tipo `i32`, puesto que `TieneArea` esta en nuestro código. Pero si intentáramos implementar `ToString`, un trait proporcionado por Rust para `i32`, no podríamos, debido a que ni el trait o el tipo están en nuestro código.

Una ultima cosa acerca de los traits: las funciones genéricas con un limite de trait usan ‘monomorfizacion’ (‘monomorphization’) (mono: uno, morfos: forma), y por ello son despachadas estáticamente. Que significa esto? Echa un vistazo a el capitulo acerca de objetos trait para mas detalles.

Multiples limites de trait

Has visto que puedes limitar un parámetro de tipo genérico con un trait:

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

Si necesitas mas de un limite, puedes hacer uso de `+`:

```
use std::fmt::Debug;

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!("{:?}", x);
}
```

`T` necesita ahora ser ambos `Clone` y `Debug`.

La clausula where

Escribir funciones con solo unos pocos tipos genéricos y un pequeño numero de limites de trait no es tan feo, pero a medida que el numero se incrementa, la sintaxis se vuelve un poco extraña:

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

El nombre de la función esta lejos a la izquierda, y la lista de parámetros esta lejos a la derecha. Los limites de trait se interponen en la mitad.

Rust tiene una solución, y se llama ‘clausula `where`’:

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn main() {
    foo("Hola", "mundo");
    bar("Hola", "mundo");
}
```

`foo()` usa la sintaxis demostrada previamente, y `bar()` usa una clausula `where` . Todo lo que necesitas es dejar los limites por fuera cuando definas tus parámetros de tipo y luego agregar un `where` después de la lista de parámetros. Para listas mas largas, espacios en blanco pueden ser agregados:

```
use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

Dicha flexibilidad puede agregar claridad en situaciones complejas.

La clausula `where` es también mas poderosa que la sintaxis mas simple. Por ejemplo:

```
trait ConvertirA<Salida> {
    fn convertir(&self) -> Salida;
}

impl ConvertirA<i64> for i32 {
    fn convertir(&self) -> i64 { *self as i64 }
}

// puede ser llamada con T == i32
fn normal<T: ConvertirA<i64>>(x: &T) -> i64 {
    x.convertir()
}

// puede ser llamada con T == i64
fn inversa<T>() -> T
        // pesto es user ConvertirA como si fuera "ConvertirA<i64>"
        where i32: ConvertirA<T> {
    42.convertir()
}
```

Lo anterior demuestra una característica adicional de `where`: permite limites en los que el lado izquierdo es un tipo arbitrario (`i32` en este caso), no solo un simple parámetro de tipo (como `T`).

Metodos por defecto

Si ya sabes como un implementador típico definirá un método, puedes permitir a tu trait proporcionar uno método por defecto:

```
trait Foo {
    fn es_valido(&self) -> bool;

    fn es_invalido(&self) -> bool { !self.es_valido() }
}
```

Los implementadores del trait `Foo` necesitan implementar `es_valido()`, pero no necesitan implementar `es_invalido()`. Lo obtendrán por defecto. También pueden sobreescribir la implementación por defecto si lo desean:

```
# trait Foo {
#     fn es_valido(&self) -> bool;
#
#     fn es_invalido(&self) -> bool { !self.es_valido() }
# }
struct UsaDefault;

impl Foo for UsaDefault {
    fn es_valido(&self) -> bool {
        println!("UsaDefault.es_valid llamada.");
        true
    }
}

struct SobreescribeDefault;

impl Foo for SobreescribeDefault {
    fn es_valido(&self) -> bool {
        println!("SobreescribeDefault.es_valido llamada.");
        true
    }

    fn es_invalido(&self) -> bool {
        println!("SobreescribeDefault.es_invalido llamada!");
        true // esta implementacion es una auto-contradiccion!
    }
}

let default = UsaDefault;
assert!(!default.es_valido()); // imprime "UsaDefault.es_valid llamada."

let sobre = SobreescribeDefault;
assert!(sobre.is_invalid()); // prints "SobreescribeDefault.es_invalido llamada!"
```

Herencia

Algunas veces, implementar un trait requiere implementar otro:

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

Los implementadores de `FooBar` deben también implementar `Foo`, de esta manera:

```
# trait Foo {
#     fn foo(&self);
# }
# trait FooBar : Foo {
#     fn foobar(&self);
# }
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

Si olvidamos implementar `Foo`, Rust nos lo dira:

```
error: the trait `main::Foo` is not implemented for the type `main::Baz` [E0277]
```

% Drop

Ahora que hemos discutido los traits, hablemos de un trait particular proporcionado por la biblioteca estándar de Rust, `Drop`. El trait `Drop` provee una forma de ejecutar código cuando un valor sale de ámbito. Por ejemplo:

```
struct TieneDrop;

impl Drop for TieneDrop {
    fn drop(&mut self) {
        println!("Dropeando!");
    }
}

fn main() {
    let x = TieneDrop;

    // hacemos algo

} // x sale de ambito aqui
```

Cuando `x` sale de ámbito al final de `main()`, el código de `Drop` es ejecutado. `Drop` posee un método, también denominado `drop()`. Dicho método toma una referencia mutable a `self`.

Eso es todo! La mecánica de `Drop` es muy simple, sin embargo, hay algunos detalles. Por ejemplo, los valores son liberados (dropped) en orden opuesto a como fueron declarados. He aquí otro ejemplo:

```
struct Explosivo {
    potencia: i32,
}

impl Drop for Explosivo {
    fn drop(&mut self) {
        println!("BOOM multiplicado por {}!!!", self.potencia);
    }
}

fn main() {
    let petardo = Explosivo { potencia: 1 };
    let tnt = Explosivo { potencia: 100 };
}
```

Lo anterior imprimira:

```
BOOM multiplicado por 100!!!
BOOM multiplicado por 1!!!
```

El TNT se va primero que el petardo, debido que fue creado después. Ultimo en entrar, primero en salir.

Entonces para que es bueno `Drop` ? Generalmente, es usado para limpiar cualquier recurso asociado a un `struct` . Por ejemplo, el tipo `Arc<T>` es un tipo con conteo de referencias. Cuando `Drop` es llamado, este decrementará el contador de referencias, y si el numero total de referencias es cero, limpiará el valor subyacente.

% if let

`if let` te permite combinar `if` y `let` para reducir el costo de algunos tipos de coincidencia de patrones.

Por ejemplo, digamos que tenemos algún tipo de `Option<T>`. Queremos llamar a una función sobre ello en caso de ser un `Some<T>`, pero no hacer nada si es `None`. Seria algo así:

```
# let option = Some(5);
# fn foo(x: i32) { }
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

No tenemos que usar `match` aquí, por ejemplo, podríamos usar `if`:

```
# let option = Some(5);
# fn foo(x: i32) { }
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

Ninguna de esas dos opciones es particularmente atractiva. Podemos usar `if let` para hacer lo mismo, pero de mejor forma:

```
# let option = Some(5);
# fn foo(x: i32) { }
if let Some(x) = option {
    foo(x);
}
```

Si un patrón concuerda satisfactoriamente, asocia cualquier parte apropiada del valor a los identificadores en el patrón, para luego evaluar la expresión. Si el patrón no concuerda, no pasa nada.

Si quisieras hacer algo en el caso de que el patrón no concuerde, puedes usar `else`:

```
# let option = Some(5);
# fn foo(x: i32) { }
# fn bar() { }
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

while let

De manera similar, `while let` puede ser usado cuando desees iterar condicionalmente mientras el valor concuerde con cierto patrón. Convierte código como este:

```
# let option: Option<i32> = None;
loop {
    match option {
        Some(x) => println!("{}", x),
        _ => break,
    }
}
```

En código como este:

```
# let option: Option<i32> = None;
while let Some(x) = option {
    println!("{}", x);
}
```

% Objetos Trait

Cuando el código involucra polimorfismo, es necesario un mecanismo para determinar que versión especifica debe ser ejecutada. Dicho mecanismo es denominado `despacho`. Hay dos formas mayores de despacho: despacho estático y despacho dinámico. Si bien es cierto que Rust prefiere el despacho estático, también soporta despacho dinámico a través de un mecanismo llamado 'objetos trait'.

Bases

Por el resto de este capitulo, necesitaremos un trait y algunas implementaciones. Creemos uno simple, `Foo`. `Foo` posee un solo método que retorna un `String`.

```
trait Foo {
    fn metodo(&self) -> String;
}
```

También implementaremos este trait para `u8` y `String`:

```
# trait Foo { fn metodo(&self) -> String; }
impl Foo for u8 {
    fn metodo(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn metodo(&self) -> String { format!("string: {}", *self) }
}
```

Despacho estático

Podemos usar el trait para efectuar despacho estático mediante el uso de limites de trait:

```
# trait Foo { fn metodo(&self) -> String; }
# impl Foo for u8 { fn metodo(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn metodo(&self) -> String { format!("string: {}", *self) } }
fn hacer_algo<T: Foo>(x: T) {
    x.metodo();
}

fn main() {
    let x = 5u8;
    let y = "Hola".to_string();

    hacer_algo(x);
    hacer_algo(y);
}
```

Rust utiliza ‘monomorfizacion’ para el despacho estático en este código. Lo que significa que creara una versión especial de `hacer_algo()` para ambos `u8` y `String`, reemplazando luego los lugares de llamada con invocaciones a estas funciones especializadas. En otras palabras, Rust genera algo así:

```
# trait Foo { fn metodo(&self) -> String; }
# impl Foo for u8 { fn metodo(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn metodo(&self) -> String { format!("string: {}", *self) } }
fn hacer_algo_u8(x: u8) {
    x.metodo();
}

fn hacer_algo_string(x: String) {
    x.metodo();
}

fn main() {
    let x = 5u8;
    let y = "Hola".to_string();

    hacer_algo_u8(x);
    hacer_algo_string(y);
}
```

Lo anterior posee una gran ventaja: el despacho estático permite que las llamadas a función puedan ser insertadas en linea debido a que el receptor es conocido en tiempo de compilación, y la inserción en linea es clave para una buena optimizacion. El despacho estático es rápido, pero viene con una desventaja: ‘código inflado’ (‘code bloat’), a consecuencia de las repetidas copias de la misma función que son insertadas en el binario, una para cada tipo.

Ademas, los compiladores no son perfectos y pueden “optimizar” código haciendolo mas lento. Por ejemplo, funciones insertadas en linea de manera ansiosa inflaran la cache de instrucciones (y el cache gobierna todo a nuestro alrededor). Es por ello que `#[inline]` y `#[inline(always)]` deben ser usadas con cautela, y una razón por la cual usar despacho dinámico es algunas veces mas eficiente.

Sin embargo, el caso común es que el despacho estático sea mas eficiente. Uno puede tener una delgada función envoltorio despachada de manera estática efectuando despacho dinámico, pero no vice versa, es decir; las llamadas estáticas son mas flexibles. Es por esto que la biblioteca estándar intenta ser despachada dinámicamente siempre y cuando sea posible.

Despacho dinámico

Rust proporciona despacho dinámico a través de una facilidad denominada ‘objetos trait’. Los objetos trait, como `&Foo` o `Box<Foo>`, son valores normales que almacenan un valor de *cualquier* tipo que implementa el trait determinado, en donde el tipo preciso solo puede ser determinado en tiempo de ejecución.

Un objeto trait puede ser obtenido de un apuntador a un tipo concreto que implemente el trait *convirtiéndolo* (e.j. `&x as &Foo`) o aplicándole *coercion* (e.j. usando `&x` como un argumento a una función que recibe `&Foo`).

Esas coerciones y conversiones también funcionan para apuntadores como `&mut T` a `&mut Foo` y `Box<T>` a `Box<Foo>`, pero eso es todo hasta el momento. Las coerciones y conversiones son idénticas.

Esta operación puede ser vista como el ‘borrado’ del conocimiento del compilador acerca del tipo especifico del apuntador, y es por ello que los objetos traits son a veces referidos como `borrado de tipos`.

Volviendo a el ejemplo anterior, podemos usar el mismo trait para llevar a cabo despacho dinámico con conversión de objetos trait:

```
# trait Foo { fn metodo(&self) -> String; }
# impl Foo for u8 { fn metodo(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn metodo(&self) -> String { format!("string: {}", *self) } }

fn hacer_algo(x: &Foo) {
    x.metodo();
}

fn main() {
    let x = 5u8;
    hacer_algo(&x as &Foo);
}
```

o a traves de la coercion:

```
# trait Foo { fn metodo(&self) -> String; }
# impl Foo for u8 { fn metodo(&self) -> String { format!("u8: {}", *self) } }
# impl Foo for String { fn metodo(&self) -> String { format!("string: {}", *self) } }

fn hacer_algo(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hola".to_string();
    hacer_algo(&x);
}
```

Una función que recibe un objeto trait no es especializada para cada uno de los tipos que implementan `Foo`: solo una copia es generada, resultando algunas veces (pero no siempre) en menos inflación de código. Sin embargo, el despacho dinámico viene con el costo de requerir las llamadas mas lentas a funciones virtuales, efectivamente inhibiendo cualquier posibilidad de inserción en linea y las optimizaciones relacionadas.

Porque apuntadores?

Rust, a diferencia de muchos lenguajes administrados, no coloca cosas detrás de apuntadores por defecto, lo que se traduce en que los tipos tengan diferentes tamaños. Conocer el tamaño de un valor en tiempo de compilación es importante para cosas como: pasarlo como argumento a una función, moverlo en la pila y asignarle (y deasignarle) espacio en el montículo para su almacenamiento.

Para `Foo`, necesitaríamos tener un valor que podría ser mas pequeño que un `String` (24 bytes) o un `u8` (1 byte), así como cualquier otro tipo que pueda implementar `Foo` en crates dependientes (cualquier numero de bytes). No hay forma de garantizar que este

ultimo caso pueda funcionar si los valores no son almacenados en un apuntador, puesto que esos otros tipos pueden ser de tamaño arbitrario.

Colocar el valor detrás de un apuntador significa que el tamaño del valor no es relevante cuando estemos lanzando un objeto trait por los alrededores, solo el tamaño del apuntador en si mismo.

Representación

Los métodos del trait pueden ser llamados en un objeto trait a través de un registro de apuntadores a función tradicionalmente llamado 'vtable' (creado y administrado por el compilador).

Los objetos trait son simples y complejos al mismo tiempo: su representation y distribución es bastante directa, pero existen algunos mensajes de error un poco raros y algunos comportamientos sorpresivos por descubrir.

Comencemos por lo mas simple, la representación de un objeto trait en tiempo de ejecución. El modulo `std::raw` contiene structs con distribuciones que son igual de complicadas que las de los tipos integrados, incluyendo los objetos trait:

```
# mod foo {
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
# }
```

Eso es, un trait object como `&Foo` consiste de un apuntador 'data' y un apuntador 'vtable'.

El apuntador data apunta hacia los datos (de un tipo desconocido `T`) que guarda el objeto trait, y el vtable pointer apunta a la vtable (table de metodos virtuales') ('virtual method table') correspondiente a la implementación de `Foo` para `T`.

Una vtable es esencialmente una struct de apuntadores a función, apuntando al segmento concreto de código maquina para cada implementación de metodo. Una llamada a metodo como `objeto_trait.metodo()` retornara el apuntador correcto desde la vtable y luego hará una llamada dinámica de este. Por ejemplo:

```
struct FooVtable {
    destructor: fn(*mut ()),
    tamano: usize,
    alineacion: usize,
    metodo: fn(*const ()) -> String,
}

// u8:

fn llamar_metodo_en_u8(x: *const ()) -> String {
    // el compilador garantiza que esta función solo sea llamada
    // con `x` apuntando a un u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.metodo()
}

static Foo_vtable_para_u8: FooVtable = FooVtable {
    destructor: /* magia del compilador */,
    tamano: 1,
    alineacion: 1,

    // conversion a a un apuntador a función
    metodo: llamar_metodo_en_u8 as fn(*const ()) -> String,
};


// String:

fn llamar_metodo_en_String(x: *const ()) -> String {
    // el compilador garantiza que esta función solo sea llamada
    // con `x` apuntando a un String
    let string: &String = unsafe { &*(x as *const String) };

    string.metodo()
}

static Foo_vtable_para_String: FooVtable = FooVtable {
    destructor: /* magia del compilador */,
    // valores para una computadora de 64 bits, dividelos por la mitad para una de 32
    tamano: 24,
    alineacion: 8,

    metodo: llamar_metodo_en_String as fn(*const ()) -> String,
};
```

El campo `destructor` en cada vtable apunta a una función que limpiara todos los recursos del tipo de la vtable: para `u8` es trivial, pero para `String` liberara memoria. Esto es necesario para adueñarse de objetos trait como `Box<Foo>`, que necesitan limpiar ambos la asignación `Box` así como el tipo interno cuando salgan de ámbito. Los campos `tamano` y

`alineacion` almacenan el tamaño del tipo borrado, y sus requerimientos de alineación; estos son esencialmente no usados por el momento puesto que la información es embebida en el destructor, pero sera usada en el futuro, a medida que los objetos trait sean progresivamente hechos mas flexibles.

Supongamos que tenemos algunos valores que implementen `Foo`. La forma explicita de construcción y uso de objetos trait `Foo` puede lucir un poco como (ignorando las incongruencias entre tipos: son apuntadores de cualquier manera):

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // almacena los datos
    data: &a,
    // almacena los metodos
    vtable: &Foo_vtable_para_String
};

// let y: &Foo = x;
let y = TraitObject {
    // almacena los datos
    data: &x,
    // almacena los metodos
    vtable: &Foo_vtable_para_u8
};

// b.metodo();
(b.vtable.metodo)(b.data);

// y.metodo();
(y.vtable.metodo)(y.data);
```

Seguridad de Objetos

No todo trait puede ser usado para crear un objeto trait. Por ejemplo, los vectores implementan `Clone`, pero si intentamos crear un objeto trait:

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

Obtenemos un error:

```
error: cannot convert to a trait object because trait `core::clone::Clone` is not object-
let o = &v as &Clone;
        ^~
note: the trait cannot require that `Self : Sized`
let o = &v as &Clone;
        ^~
```

El error dice que `Clone` no es ‘seguro para objetos’ (‘object-safe’). Solo los traits que son seguros para objetos pueden ser usados en la creación de objetos trait. Un trait es seguro para objetos si ambas condiciones son verdaderas:

- el trait no requiere que `Self: Sized`
- todos sus métodos son seguros para objetos

Entonces, que hace a un metodo seguro para objetos? Cada metodo debe requerir que `Self: Sized` o todas de las siguientes:

- no debe tener ninguna parámetro de tipo
- no debe usar `Self`

Uff! Como podemos ver, casi todas estas reglas hablan acerca de `Self`. Una buena intuición seria “exceptuando circunstancias especiales, si tu metodo de trait usa `Self`, no es seguro para objetos.”

% Closures

Algunas veces es util envolver una función y sus *variables libres* para mejor claridad y reusabilidad. Las variables libres que pueden ser usadas provienen del ámbito exterior y son 'cerradas' ('closed over') cuando son usadas en la función. De allí el nombre 'closure'. Rust provee una muy buena implementación, como veremos a continuación.

Sintaxis

Los closures lucen así:

```
let suma_uno = |x: i32| x + 1;

assert_eq!(2, suma_uno(1));
```

Creamos un enlace a variable, `suma_uno` y lo asignamos a un closure. Los argumentos del closure van entre pipes (`|`), y el cuerpo es una expresión, en este caso, `x + 1` . Recuerda que `{ }` es una expresión, de manera que podemos tener también closures multi-linea:

```
let suma_dos = |x| {
    let mut resultado: i32 = x;

    resultado += 1;
    resultado += 1;

    resultado
};

assert_eq!(4, suma_dos(2));
```

Notaras un par de cosas acerca de los closures que son un poco diferentes de las funciones regulares definidas con `fn` . Lo primero es que no necesitamos anotar los tipos de los argumentos los valores de retorno. Podemos:

```
let suma_uno = |x: i32| -> i32 { x + 1 };

assert_eq!(2, suma_uno(1));
```

Pero no necesitamos hacerlo. Porque? Básicamente, se implemento de esa manera por razones de ergonomia. Si bien especificar el tipo completo para funciones con nombre es de utilidad para cosas como documentación e inferencia de tipos, la firma completa en los

closures es raramente documentada puesto a que casi siempre son anónimos, y no causan los problemas de tipo de error-a-distancia que la inferencia en funciones con nombre pueden causar.

La segunda sintaxis es similar, pero un tanto diferente. He agregado espacios acá para facilitar la comparación:

```
fn  suma_uno_v1   (x: i32) -> i32 { x + 1 }
let suma_uno_v2 = |x: i32| -> i32 { x + 1 };
let suma_uno_v3 = |x: i32|          x + 1  ;
```

Pequenas diferencias, pero son similares.

Closures y su entorno

El entorno para un closure puede incluir enlaces a variable del ámbito que los envuelve en adición a los parámetros y variables locales. De esta manera:

```
let num = 5;
let suma_num = |x: i32| x + num;

assert_eq!(10, suma_num(5));
```

El closure `suma_num`, hace referencia a el enlace `let` en su ámbito: `num`. Mas específicamente, toma prestado el enlace. Si hacemos algo que resulte en un conflicto con dicho enlace, obtendríamos un error como este:

```
let mut num = 5;
let suma_num = |x: i32| x + num;

let y = &mut num;
```

Que falla con:

```
error: cannot borrow `num` as mutable because it is also borrowed as immutable
    let y = &mut num;
                ^~~
note: previous borrow of `num` occurs here due to use in closure; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` until the borrow
  ends
    let suma_num = |x| x + num;
                    ^~~~~~~~~~
note: previous borrow ends here
fn main() {
    let mut num = 5;
    let suma_num = |x| x + num;

    let y = &mut num;
}
^
```

Un error un tanto verbose pero igual de util! Como lo dice, no podemos tomar un préstamo mutable en `num` debido a que el closure ya esta tomándolo prestado. Si dejamos el closure fuera de ámbito, entonces es posible:

```
let mut num = 5;
{
    let suma_num = |x: i32| x + num;

} // suma_num sale de ambito, el prestamo termina aqui

let y = &mut num;
```

Sin embargo, si tu closure así lo requiere, Rust tomara pertenecia y moverá el entorno. Lo siguiente no funciona:

```
let nums = vec![1, 2, 3];

let toma_nums = || nums;

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

Obtenemos este error:

```
note: `nums` moved into closure environment here because it has type
  `[closure(()) -> collections::vec::Vec]`, which is non-copyable
let toma_nums = || nums;
                ^~~~~~
```

`Vec<T>` posee pertenecia de su contenido, y es por ello, que al hacer referencia a el dentro de nuestro closure, tenemos que tomar pertenencia de `nums`. Es lo mismo que si hubiéramos proporcionado `nums` como argumento a una función que tomara pertenencia sobre el.

Closures `move`

Podemos forzar nuestro closure a tomar pertenecia de su entorno con la palabra reservada `move`:

```
let num = 5;

let toma_pertenecia_num = move |x: i32| x + num;
```

Ahora, aun cuando la palabra reservada `move` esta presente, las variables siguen la semántica normal. En este caso, `5` implementa `Copy`, y en consecuencia, `toma_pertenecia_num` toma pertenecia de una copia de `num`. Entonces, cual es la diferencia?

```
let mut num = 5;

{
    let mut suma_num = |x: i32| num += x;

    suma_num(5);
}

assert_eq!(10, num);
```

En este caso, nuestro closure tomo una referencia mutable a `num`, y cuando llamamos a `suma_num`, este muto el valor subyacente, tal y como lo esperábamos. También necesitamos declarar `suma_num` como `mut`, puesto a que estamos mutando su entorno.

Si lo cambiamos a un closure `move`, es diferente:

```
let mut num = 5;

{
    let mut suma_num = move |x: i32| num += x;

    suma_num(5);
}

assert_eq!(5, num);
```

Obtenemos solo `5`. En lugar de tomar un prestamo mutable en nuestro `num`, tomamos pertenecia sobre una copia.

Otra forma de pensar acerca de los closures `move` es: estos proporcionan a el closure su propio registro de activación. Sin `move`, el closure puede ser asociado a el registro de activación que lo creo, mientras que un closure `move` es autocontenido. Lo que significa, por ejemplo, que generalmente no puedes retornar un closure no-`move` desde una función.

Pero antes de hablar de recibir closures como parámetros y usarlos como valores de retorno, debemos hablar un poco mas acerca de su implementación. Como un lenguaje de programación de sistemas Rust te proporciona una tonelada de control acerca de lo que tu código hace, y los closures no son diferentes.

Implementación de los Closures

La implementación de closures de Rust es un poco diferente a la de otros lenguajes. En Rust, los closures son efectivamente una sintaxis alterna para los traits. Antes de continuar, necesitaras haber leído el capitulo de traits asi como el capitulo acerca de objetos trait.

Ya los has leído? Excelente.

La clave para entender como funcionan los closures es algo un poco extraño: Usar `()` para llamar una función, como `foo()`, es un operador sobrecargable. Partiendo desde esta premisa, todo lo demás encaja. En Rust hacemos uso de el sistema de traits para sobrecargar operadores. Llamar funciones no es diferente. Existen tres traits que podemos sobrecargar:

```
# mod foo {
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
# }
```

Notaras unas pocas diferencias entre dichos traits, pero una grande es `self`: `Fn` recibe `&self`, `FnMut` toma `&mut self` y `FnOnce` recibe `self`. Lo anterior cubre los tres tipos de `self` a través de la sintaxis usual de llamadas a métodos. Pero han sido separados en tres traits, en lugar de uno solo. Esto nos proporciona gran control acerca del tipo de closures que podemos recibir.

La sintaxis `|| {}` es una sintaxis alterna para esos tres traits. Rust generara un `struct` para el entorno, `impl` el trait apropiado, y luego hará uso de este.

Recibiendo closures como argumentos

Ahora que sabemos que los closures son traits, entonces sabemos como aceptar y retornar closures: justo como cualquier otro trait!

Lo anterior también significa que podemos elegir entre despacho estático o dinámico. Primero, creemos una función que reciba algo llamable, ejecute una llamada sobre el y luego retorne el resultado:

```
fn llamar_con_uno<F>(algun_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    algun_closure(1)
}

let respuesta = llamar_con_uno(|x| x + 2);

assert_eq!(3, respuesta);
```

Pasamos nuestro closure, `|x| x + 2`, a `llamar_con_uno`. `llamar_con_uno` hace lo que sugiere: llama el closure, proporcionándole `1` como argumento.

Examinemos la firma de `llamar_con_uno` con mayor detalle:

```
fn llamar_con_uno<F>(algun_closure: F) -> i32
#    where F : Fn(i32) -> i32 {
#    algun_closure(1) }
```

Recibimos un parámetro, de tipo `F`. También retornamos un `i32`. Esta parte no es interesante. La siguiente lo es:

```
# fn llamar_con_uno<F>(algun_closure: F) -> i32
    where F : Fn(i32) -> i32 {
#   algun_closure(1) }
```

Debido a que `Fn` es un trait, podemos limitar nuestro genérico con el. En este caso, nuestro closure recibe un `i32` y retorna un `i32`, es por ello que el limite de genéricos que usamos es `Fn(i32) -> i32`.

Hay otro punto clave acá: debido a que estamos limitando un genérico con un trait, la llamada sera monomorfizada, y en consecuencia, estaremos haciendo despacho estático en el closure. Eso es super cool. En muchos lenguajes, los closures son inherentemente asignados desde el montículo, y casi siempre involucraran despacho dinámico. En Rust podemos asignar el entorno de nuestros closures desde la pila, así como despachar la llamada de manera estática. Esto ocurre con bastante frecuencia con los iteradores y sus adaptadores, quienes reciben closures como argumentos.

Por supuesto, si deseamos despacho dinámico, podemos tenerlo también. Un objeto trait, como es usual, maneja este caso:

```
fn llamar_con_uno(algun_closure: &Fn(i32) -> i32) -> i32 {
    algun_closure(1)
}

let answer = llamar_con_uno(&|x| x + 2);

assert_eq!(3, answer);
```

Ahora recibimos un objeto trait, un `&Fn`. Y tenemos que hacer una referencia a nuestro closure cuando lo pasemos a `llamar_con_uno`, es por ello que usamos `&||`.

Apuntadores a función y closures

Un apuntador a función es una especie de closure que no posee entorno. Como consecuencia, podemos pasar un apuntador a función a cualquier función que reciba un closure como argumento:

```
fn llamar_con_uno(algun_closure: &Fn(i32) -> i32) -> i32 {
    algun_closure(1)
}

fn suma_uno(i: i32) -> i32 {
    i + 1
}

let f = suma_uno;

let respuesta = llamar_con_uno(&f);

assert_eq!(2, respuesta);
```

En el ejemplo anterior no necesitamos la variable intermedia `f` de manera estricta, el nombre de la función también sirve:

```
let respuesta = llamar_con_uno(&suma_uno);
```

Retornando closures

Es muy común para código con estilo funcional el retornar closures en diversas situaciones. Si intentas retornar un closure, podrías incurrir en un error. Al principio puede parecer extraño, pero mas adelante lo entenderemos. Probablemente intentarías retornar un closure desde una función de esta manera:

```
fn factory() -> (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let respuesta = f(1);
assert_eq!(6, respuesta);
```

Lo anterior genera estos largos, pero relacionados errores:

```
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> i32` [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~
error: the trait `core::marker::Sized` is not implemented for the type `core::ops::Fn(i32
let f = factory();
    ^
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
let f = factory();
    ^
```

Para retornar algo desde una función, Rust necesita saber el tamaño del tipo de retorno. Pero debido q que `Fn` es un trait, puede ser varias cosas de diversos tamaños: muchos tipos pueden implementar `Fn`. Una manera fácil de darle tamaño a algo es tomando una referencia a este, debido a que las referencias tienen un tamaño conocido. En lugar de lo anterior podemos escribir:

```
fn factory() -> &(Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let respuesta = f(1);
assert_eq!(6, respuesta);
```

Pero obtenemos otro error:

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
```

Bien. Debido a que tenemos una referencia, necesitamos proporcionar un tiempo de vida. pero nuestra función `factory()` no recibe ningun argumento y por ello la elision de tiempos de vida no es posible en este caso. Entonces que opciones tenemos? Intentemos con `'static`:

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let respuesta = f(1);
assert_eq!(6, respuesta);
```

Pero obtenemos otro error:

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure@:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~
```

El error nos dice que no tenemos un `&'static Fn(i32) -> i32`, sino un `[closure@<anon>:7:9: 7:20]`. Un momento, que?

Debido a que nuestro closure genera su propio `struct` para el entorno así como una implementación para `Fn`, `FnMut` y `FnOnce`, dichos tipos son anónimos. Solo existen para este closure. Es por ello que Rust los muestra como `closure@<anon>` en vez de algún nombre autogenerado.

El error también habla de que se espera que el tipo de retorno sea una referencia, pero lo que estamos tratando de retornar no lo es. Mas aun, no podemos asignar directamente un tiempo de vida `'static` a un objeto. Entonces tomaremos un enfoque diferente y retornaremos un 'trait object' envolviendo el `Fn` en un `Box`. Lo siguiente *casi* funciona:

```
fn factory() -> Box i32> {
    let num = 5;

    Box::new(|x| x + num)
}
# fn main() {
let f = factory();

let respuesta = f(1);
assert_eq!(6, respuesta);
# }
```

Hay un ultimo problema:

```
error: closure may outlive the current function, but it borrows `num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~
```

Bueno, como discutimos anteriormente, los closures toman su entorno prestado. Y en este caso, nuestro entorno esta basado en un `5` asignado desde la pila, la variable `num`. Debido a esto el prestamo posee el tiempo de vida del registro de activación. De retornar este closure, la llamada a función podría terminar, el registro de activación desaparecería y nuestro closure estaría capturando un entorno de memoria basura! Con un ultimo arreglo, podemos hacer que funcione:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
# fn main() {
let f = factory();

let respuesta = f(1);
assert_eq!(6, respuesta);
# }
```

Al hacer el closure interno un `move Fn`, hemos creado un nuevo registro de activación para nuestro closure. Envolviéndolo con un `Box`, le hemos proporcionado un tamaño conocido, permitiéndole escapar nuestro registro de activación.

% Sintaxis Universal de Llamadas a Función

Algunas veces, varias funciones pueden tener el mismo nombre. Considera el siguiente código:

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!("impl de Foo en Baz"); }
}

impl Bar for Baz {
    fn f(&self) { println!("impl de Bar en Baz"); }
}

let b = Baz;
```

Si intentaramos llamar `b.f()`, obtendriamos un error:

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Foo"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Bar"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Necesitamos una forma de eliminar la ambigüedad. Esta característica es denominada 'sintaxis universal de llamadas a función', y luce así:

```
# trait Foo {
#     fn f(&self);
# }
# trait Bar {
#     fn f(&self);
# }
# struct Baz;
# impl Foo for Baz {
#     fn f(&self) { println!("impl de Foo en Baz"); }
# }
# impl Bar for Baz {
#     fn f(&self) { println!("impl de Bar en Baz"); }
# }
# let b = Baz;
Foo::f(&b);
Bar::f(&b);
```

Analicémoslo por partes.

```
Foo::
Bar::
```

Dichas mitades de invocacion son los tipos de los traits: `Foo` y `Bar`. Lo anterior es responsable de la eliminación de la ambigüedad entre las dos llamadas: Rust llama la función del trait que nombraste.

```
f(&b)
```

Cuando llamamos un método de la forma `b.f()` usando la sintaxis de métodos, Rust automáticamente tomara prestado a `b` si `f()` recibe a `&self`. En este caso, Rust no puede hacerlo. Es por ello que debemos pasar a `&b` de manera explicita.

Usando <>

La forma de SULF de la que acabamos de hablar:

```
Trait::metodo(args);
```

Es una version corta. Existe una version expandida que puede ser necesaria en algunas situaciones:

```
::metodo(args);
```

La sintaxis `<>::` es una forma de proporcionar una pista de tipos. El tipo va dentro de los `<>` s. En este caso es `Type as Trait` indicando que deseamos llamar a la version `Trait` de `metodo`. La sección `as Trait` es opcional de no ser ambigua. Lo mismo con los `<>` s, de allí proviene version mas corta.

Acá un ejemplo del uso de la forma mas larga.

```
trait Foo {
    fn clone(&self);
}

#[derive(Clone)]
struct Bar;

impl Foo for Bar {
    fn clone(&self) {
        println!("Creando un clon de Bar");

        <Bar as Clone>::clone(self);
    }
}
```

Lo anterior llamara el metodo `clone` en el trait `Clone`, en vez de la version `clone` del trait `Foo`.

% Crates y Modulos

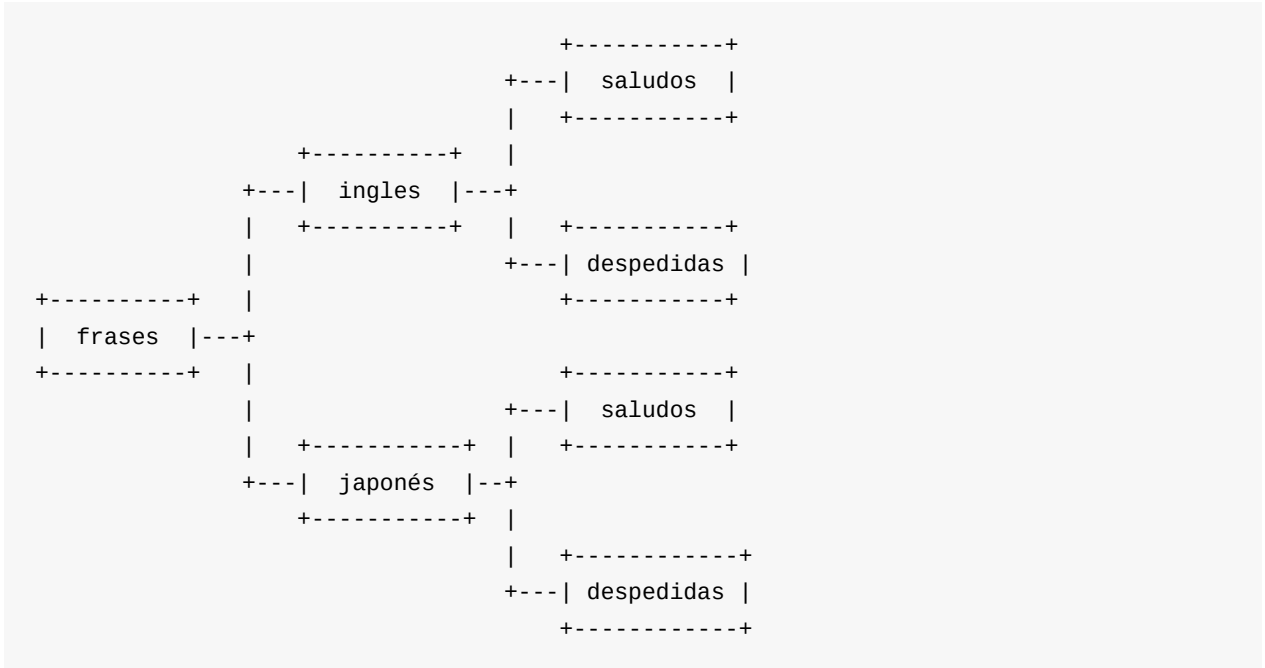
Cuando un proyecto comienza a crecer, se considera buena practica de ingeniería de software dividirlo en un grupo de componentes mas pequeños que posteriormente serán unidos. También es importante tener una interfaz bien definida, de manera que una parte de la funcionalidad sea privada, y otra publica. Para facilitar esto, Rust posee un sistema de módulos.

Terminologia basica: Crates y Modulos

Rust posee dos términos distintos relacionados con el sistema de módulos: ‘crate’ y ‘modulo’. Un crate es un sinónimo para ‘biblioteca’ or ‘paquete’ en otros lenguajes. De allí el nombre “Cargo” del manejador de paquetes de Rust: haces disponibles tus crates a los demás con Cargo. Los crates pueden producir un ejecutables o una biblioteca, dependiendo del proyecto.

Cada crate posee un *modulo raiz* implícito que contiene el código para dicho crate. Puedes definir un árbol de sub-modulos bajo el módulo raíz. Los módulos te permiten particionar tu codigo dentro del crate.

Como ejemplo, creemos un crate *frases*, que nos proveerá varias frases en diferentes idiomas. Para mantener las cosas simples, nos apegaremos solo a ‘saludos’ y ‘despedidas’ como los dos tipos de frases, así como Ingles y Japonés (日本語) para los dos idiomas en los que las frases estarán representadas. Tendremos la siguiente distribución de módulos:



En este ejemplo, `frases` es el nombre de nuestro crate. El resto son módulos. Puedes observar que los módulos forman un árbol, partiendo desde la *raíz*, que a su vez es la raíz del árbol `frases` en si.

Ahora que tenemos un plan, definamos estos módulos en código. Para comenzar, generemos un proyecto con Cargo:

```
$ cargo new frases
$ cd frases
```

Si recuerdas, lo anterior genera un proyecto simple:

```
$ tree .
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` es la raíz de nuestro crate, correspondiendo con `frases` en nuestro diagrama anterior. above.

Definiendo Módulos

Para definir cada uno de nuestros módulos, usamos la palabra reservada `mod`. Hagamos que nuestro `src/lib.rs` se vea así:

```
mod ingles {
    mod saludos {
    }

    mod despedidas {
    }
}

mod japones {
    mod saludos {
    }

    mod despedidas {
    }
}
```

Después de la palabra reservada `mod` debes proporcionar el nombre del modulo. Los nombres de de modulo siguen las mismas convenciones que los demás identificadores en Rust: `snake_case_en_minusculas`. El contenido de cada modulo esta delimitado por llaves (`{}`).

Dentro de un determinado `mod`, puedes declarar sub-`mod`s. Podemos referirnos a los sub-modulos con una notación dos puntos dobles (`::`): nuestros cuatro módulos anidados son `ingles::saludos`, `ingles::despedidas`, `japones::saludos`, y `japones::despedidas`. Debido a que estos sub-modulos están dentro del espacio de nombres del modulo padre, los nombres no entran en conflicto: `ingles::saludos` y `japones::saludos` son distintos incluso cuando sus nombres son ambos `saludos`.

Debido a que este crate no posee una función `main()`, y se llama `lib.rs`, Cargo construirá este crate como una biblioteca:

```
$ cargo build
   Compiling frases v0.0.1 (file:///home/tu/proyectos/frases)
$ ls target/debug
build  deps  examples  libfrases-a7448e02a0468eaa.rlib  native
```

`libfrases-hash.rlib` es el crate compilado. Antes de ver como usar este crate desde otro, dividámoslo en multiples archivos.

Crates con multiple archivos

Si cada crate fuera un solo archivo, dichos archivos serian bastante grandes. Es mas fácil dividir los crates en multiples archivos. Rust soporta esto de dos maneras.

En lugar de declarar un modulo así:

```
mod ingles {
    // el contenido del modulo va aquí
}
```

Podemos declararlo de esta forma:

```
mod ingles;
```

Si hacemos eso, Rust esperara encontrar bien un archivo `ingles.rs` o un archivo `ingles/mod.rs` con el contenido de nuestro modulo.

Nota que todos en estos archivos, no necesitas re-declarar el modulo: con la declaración inicial `mod` es suficiente.

Usando estas dos técnicas, podemos partir nuestro crate en dos directorios y siete archivos:

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── ingles
│   │   ├── despedidas.rs
│   │   ├── saludos.rs
│   │   └── mod.rs
│   ├── japones
│   │   ├── despedidas.rs
│   │   ├── saludos.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libfrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` es la raíz de nuestro crate, y luce así:

```
mod ingles;
mod japones;
```

Estas dos declaraciones le dicen a Rust que busque bien sea `src/ingles.rs` y `src/japones.rs`, o `src/ingles/mod.rs` y `src/japones/mod.rs`, dependiendo de nuestra preferencia. En este caso, debido a que nuestros módulos poseen sub-modulos hemos seleccionado la segunda. Ambos `src/ingles/mod.rs` y `src/japones/mod.rs` se ven lucen de esta manera:

```
mod saludos;
mod despedidas;
```

De nuevo, estas declaraciones hacen que Rust busque bien sea por `src/ingles/saludos.rs` y `src/japones/saludos.rs` o `src/ingles/despedidas/mod.rs` and `src/japones/despedidas/mod.rs`. Debido a que los sub-modulos no poseen sus propios sub-

modulos, hemos optado por usar el enfoque `src/ingles/saludos.rs` and `src/japones/despedidas.rs`. Uff!

Ambos `src/ingles/saludos.rs` y `src/japones/despedidas.rs` están vacíos por el momento. Agreguemos algunas funciones.

Coloca esto en `src/ingles/saludos.rs`:

```
fn hola() -> String {
    "Hello!".to_string()
}
```

Put this in `src/ingles/despedidas.rs`:

```
fn adios() -> String {
    "Goodbye.".to_string()
}
```

Put this in `src/japones/saludos.rs`:

```
fn hello() -> String {
    "こんにちは".to_string()
}
```

Por supuesto, puedes copiar y pegar el Japones desde esta pagina web, o simplemente escribe alguna otra cosa. No es importante que en efecto coloques ‘konnichiwa’ para aprender acerca del sistema de módulos de Rust.

Coloca lo siguiente en `src/japones/despedidas.rs`:

```
fn adios() -> String {
    "さようなら".to_string()
}
```

(‘Sayōnara’, por si eres curioso.)

Ahora que tenemos algo de funcionalidad en nuestro crate, intentemos usarlos desde otro.

Importing External Crates

Ya tenemos un crate biblioteca. Creemos un crate ejecutable que importe y use nuestra biblioteca.

Crea un archivo `src/main.rs` y coloca esto en el (no compilara todavía):

```
extern crate frases;

fn main() {
    println!("Hola en Ingles: {}", frases::ingles::saludos::hola());
    println!("Adios en Ingles: {}", frases::ingles::despedidas::adios());

    println!("Hola en Japones: {}", frases::japones::saludos::hola());
    println!("Adios en Japones: {}", frases::japones::despedidas::adios());
}
```

La declaración `extern crate` le informa a Rust que necesitamos compilar y enlazar al crate `frases`. Podemos entonces usar el modulo `frases` aqui. Como mencionamos anteriormente, puedes hacer uso de dos puntos dobles para hacer referencia a los sub-modulos y las funciones dentro de ellos.

Nota: cuando se importa un crate que tiene guiones en su nombre "como-este", lo cual no es un identificador valido en Rust, sera convertido cambiándole los guiones a guiones bajos, así que escribirías algo como `extern crate como_este;`

También, Cargo assume que `src/main.rs` es la raíz de un crate binario, en lugar de un crate biblioteca. Nuestro paquete ahora tiene dos crate: `src/lib.rs` y `src/main.rs`. Este patron es muy común en crates ejecutables: la mayoría de la funcionalidad esta en un crate biblioteca, y el crate ejecutable usa dicha biblioteca. De esta forma, otros programas pueden también hacer uso de la biblioteca, aunado a que ofrece un buena separación de responsabilidades.

Lo anterior todavía no funciona. Obtenemos cuatro errores que lucen similares a estos:

```
$ cargo build
   Compiling frases v0.0.1 (file:///home/tu/proyectos/frases)
src/main.rs:4:38: 4:72 error: function `hola` is private
src/main.rs:4     println!("Hola en Ingles: {}", frases::ingles::saludos::hola());
                                                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
frases/src/main.rs:4:5: 4:76 note: expansion site
```

Por defecto todo es privado en Rust. Hablemos de esto con mayor detalle.

Exportando una Interfaz Publica

Rust te permite controlar de manera precisa cuales aspectos de tu interfaz son públicos, y private es el defacto. Para hacer a algo publico, debes hacer uso de la palabra reservada `pub`. Enfoquemonos primero en el modulo `ingles`, para ello, reduzcamos nuestro `src/main.rs` a:

```
extern crate frases;

fn main() {
    println!("Hola en Ingles: {}", frases::ingles::saludos::hola());
    println!("Adios en Ingles: {}", frases::ingles::despedidas::adios());
}
```

En nuestro `src/lib.rs`, agreguemos `pub` a la declaración del modulo `ingles`:

```
pub mod ingles;
mod japones;
```

Y en nuestro `src/ingles/mod.rs`, hagamos a ambos `pub`:

```
pub mod saludos;
pub mod despedidas;
```

En nuestro `src/ingles/saludos.rs`, agreguemos `pub` a nuestra declaración `fn`:

```
pub fn hola() -> String {
    "Hola!".to_string()
}
```

Y también en `src/ingles/despedidas.rs`:

```
pub fn adios() -> String {
    "Adios.".to_string()
}
```

Ahora, nuestro crate compila, con unas pocas advertencias acerca de no haber usado las funciones en `japones`:

```
$ cargo run
   Compiling frases v0.0.1 (file:///home/tu/proyectos/frases)
src/japones/saludos.rs:1:1: 3:2 warning: function is never used: `hola`, #[warn(dead_code
src/japones/saludos.rs:1 fn hola() -> String {
src/japones/saludos.rs:2     "こんにちは".to_string()
src/japones/saludos.rs:3 }
src/japones/despedidas.rs:1:1: 3:2 warning: function is never used: `adios`, #[warn(dead_
src/japones/despedidas.rs:1 fn adios() -> String {
src/japones/despedidas.rs:2     "さようなら".to_string()
src/japones/despedidas.rs:3 }
     Running `target/debug/frases`
Hola en Ingles: Hello!
Adios en Ingles: Goodbye.
```

`pub` también aplica a las `struct` s y sus campos miembro. Y Rust teniendo siempre su tendencia hacia la seguridad, el hacer una `struct` public no hará a sus miembros public automáticamente: debes marcarlos individualmente como `pub` .

Ahora que nuestras funciones son public, podemos hacer uso de ellas. Grandioso! Sin embargo, escribir `frases::ingles::saludos::hola()` es largo y repetitivo. Rust posee otra palabra reservada para importar nombres en el ámbito actual, de manera que puedas hacer referencia a ellos con nombres mas cortos, Hablemos acerca de `use` .

Importando Modulos con `use`

Rust posee una palabra reservada, `use` , que te permite importar nombres en tu ámbito local. Cambiemos nuestro `src/main.rs` para que luzca de la siguiente manera:

```
extern crate frases;

use frases::ingles::saludos;
use frases::ingles::despedidas;

fn main() {
    println!("Hola en Ingles: {}", saludos::hola());
    println!("Adios en Ingles: {}", despedidas::adios());
}
```

Las dos lineas `use` importan cada modulo en el ámbito local, de manera tal que podamos referirnos a las funciones con nombres mucho mas cortos. Por convención, cuando se importan funciones, se considera una buena practica importar el modulo, en lugar de la función directamente. En otras palabras, puedes *hacer* esto:

```
extern crate frases;

use frases::ingles::saludos::hola;
use frases::ingles::despedidas::adios;

fn main() {
  println!("Hola en Ingles: {}", hola());
  println!("Adios en Ingles: {}", adios());
}
```

Pero no es idiomático. Hacerlo de esta forma tiene altas probabilidades de introducir un conflicto de nombres. En nuestro pequeño programa, no es gran cosa, pero a medida que crece, se va convirtiendo en un problema. Si tenemos nombres conflictivos, Rust generara un error de compilación. Por ejemplo, de haber hecho publicas las funciones de `japones` y haber intentado:

```
extern crate frases;

use frases::ingles::saludos::hola;
use frases::japones::saludos::hola;

fn main() {
    println!("Hola en Ingles: {}", hola());
    println!("Hola en Japones: {}", hola());
}
```

Rust proporcionaría un error en tiempo de compilación:

```
   Compiling frases v0.0.1 (file:///home/tu/frases)
src/main.rs:4:5: 4:40 error: a value named `hola` has already been imported in this modul
src/main.rs:4 use frases::japones::saludos::hola;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `frases`.
```

Si estuviéramos importando multiples nombres del mismo modulo, no necesitamos escribirlo dos veces. En lugar de:

```
use phrases::ingles::saludos;
use phrases::ingles::despedidas;
```

Podemos usar esta version mas corta:

```
use phrases::ingles::{saludos, despedidas};
```

Re-exportando con `pub use`

No solo usamos `use` para acortar identificadores. Puedes también usarlos dentro de tu crate para re-exportar una función dentro de otro modulo. Esto te permite presentar una interfaz externa que necesariamente no mapee de directamente a la organización interna de tu código.

Veamos un ejemplo. modifica tu `src/main.rs` para que se lea así:

```
extern crate frases;

use frases::ingles::{saludos, despedidas};
use frases::japones;

fn main() {
    println!("Hola en Ingles: {}", saludos::hola());
    println!("Adios en Ingles: {}", despedidas::adios());

    println!("Hola en Japones: {}", japones::hola());
    println!("Adios en Japones: {}", japones::adios());
}
```

Entonces, modifica tu `src/lib.rs` para hacer el modulo `japones` publico:

```
pub mod ingles;
pub mod japones;
```

A continuación, haz las dos funciones publicas, primero en `src/japones/saludos.rs`:

```
pub fn hola() -> String {
    "こんにちは".to_string()
}
```

Y luego en `src/japones/despedidas.rs`:

```
pub fn adios() -> String {
    "さようなら".to_string()
}
```

Finalmente, modifica tu `src/japones/mod.rs` de esta forma:

```
pub use self::saludos::hola;
pub use self::despedidas::adios;

mod saludos;
mod despedidas;
```

La declaración `pub use` trae la función a el ámbito en esta parte de nuestra jerarquía de modulos. Debido que hemos hecho `pub use` dentro de nuestro modulo `japones`, ahora tenemos una función `frases::japones::hola()` y una función `frases::japones::adios()`, aun cuando el código para ellas vive en `frases::japones::saludos::hola()` y frases::japones::despedidas::adios()`. Nuestra organización interna no define nuestra interfaz externa.

Acá tenemos un `pub use` para cada función que deseamos traer en el ámbito de `japones`. Alternativamente pudimos haber usado la sintaxis alternativa de comodín para incluir todo desde `saludos` en el ámbito actual: `pub use self::saludos::*`

Que hay acerca de el `self`? Bueno, por defecto, las declaraciones `use` son rutas absolutas, partiendo desde la raíz de tu crate. `self`, a diferencia, hace esa ruta relativa a tu lugar actual dentro de la jerarquía. Hay una ultima forma especial de usar `use`: puedes usar `use super::` para alcanzar un nivel superior en la jerarquía desde tu posición actual. Algunas personas gustan ver a `self` como `.` y `super` como `..` similarmente los usados por los shells para mostrar los directorios actual y padre respectivamente.

Fuera de `use`, las rutas son relativas: `foo::bar()` se refiere a una función dentro de `foo` en relación a en donde estamos. Si posee un prefijo `::`, como en `::foo::bar()`, entonces se refiere a un `foo` diferente, una ruta absoluta desde la raíz de tu crate.

El ultimo código que escribimos, compilara y se ejecutara sin problemas:

```
$ cargo run
   Compiling frases v0.0.1 (file:///home/tu/proyectos/frases)
     Running `target/debug/frases`
Hola in Ingles: Hello!
Adios in Ingles: Goodbye.
Hola in Japones: こんにちは
Adios in Japones: さようなら
```

Importes complejos

Rust ofrece un numero de opciones avanzadas que pueden hacer tus sentencias `extern crate` mas compactas y convenientes. He aquí un ejemplo:

```
extern crate frases as dichos;

use dichos::japones::saludos as saludos_ja;
use dichos::japones::despedidas::*;
use dichos::ingles::{self, saludos as saludos_en, despedidas as despedidas_en};

fn main() {
    println!("Hola en Ingles; {}", saludos_en::hola());
    println!("Y en Japones: {}", saludos_ja::hola());
    println!("Goodbye in Ingles: {}", ingles::despedidas::adios());
    println!("Otra vez: {}", despedidas_en::adios());
    println!("Y en Japones: {}", adios());
}
```

Que esta pasando?

Primero, ambos `extern crate` y `use` permiten renombrar lo que esta siendo importado. Entonces el crate todavía se llama "frases", pero aquí nos referiremos a el como "dichos". Similarmente, el primer `use` trae el modulo `japones::saludos` desde el crate, pero lo hace disponible a través del nombre `saludos_ja` en lugar de simplemente `saludos`. Lo anterior puede ayudar a evitar ambigüedad cuando se importan nombres similares desde distintos lugares.

El segundo `use` posee un asterisco para traer *todos* los símbolos desde el modulo `dichos::japones::despedidas`. Como podrás ver mas tarde podemos referirnos al `adios` Japones sin calificadores de modulo. Este tipo de glob debe ser usando con cautela.

El tercer `use` requiere un poco mas de explicación. Esta usando "expansion de llaves" para comprimir tres sentencias `use` en una (este tipo de sintaxis puede serte familiar si has escrito scripts del shell de Linux). La forma descomprimida de esta sentencia seria:

```
use dichos::ingles;
use dichos::ingles::saludos as saludos_en;
use dichos::ingles::despedidas as despedidas_en;
```

Como puedes ver, las llaves comprimen las sentencias `use` para varios items bajo la misma ruta, y en este contexto `self` hace referencia a dicha ruta. Nota: Las llaves no pueden ser anidadas o mezcladas con globbing de asteriscos.

% `const` y `static`

Rust posee una manera de definir constantes con la palabra reservada `const`:

```
const N: i32 = 5;
```

A diferencia de los enlaces a variable `let`, debes anotar el tipo de un `const`.

Las constantes viven por el tiempo de vida completo del programa. Para ser mas específicos, las constantes en Rust no tienen direcciones de memoria fijas. Esto es debido a que estas son insertadas en linea en cada lugar en el que son usadas. Por esta razón, referencias a la misma constante no están garantizadas a apuntar a la misma dirección de memoria.

`static`

Rust proporciona una facilidad de tipo 'variable global' en items estáticos. Son similares a las constantes, pero los items estáticos no son insertados en linea tras su uso. Esto significa que solo existe una instancia para cada valor, y esta ubicada en una dirección de memoria fija.

He aqui un ejemplo:

```
static N: i32 = 5;
```

A diferencia de los enlaces a variable `let`, debes anotar el tipo de un `static`.

Los items `static` viven por el tiempo de vida del programa completo, y en consecuencia cualquier referencia es almacenada en una constante posee `un tiempo de vida 'static`:

```
static NOMBRE: &'static str = "Steve";
```

Mutabilidad

Puedes introducir mutabilidad con la palabra reservada `mut`:

```
static mut N: i32 = 5;
```

Debido que es mutable, un hilo podría estar actualizando `N` mientras que otro este leyéndolo, causando inseguridad en el manejo de memoria. Es por ello que ambos el acceso y la mutación de un `static mut` son `inseguros`, y en consecuencia deben ser efectuados dentro de un bloque `unsafe`:

```
# static mut N: i32 = 5;

unsafe {
   N += 1;

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

Aunado a esto, cualquier tipo almacenado en un `static` debe ser `Sync`, y podría no tener una implementación de `Drop`.

Inicialización

Ambos `const` y `static` tienen requerimientos al proporcionarles un valor. Solo se les puede proporcionar un valor que sea una expresión constante. En otras palabras, no puedes usar el resultado de una llamada a función, algo similarmente complejo o algo determinado en tiempo de ejecución.

Cual construcción debería usar?

Casi siempre, si puedes escoger entre las dos, elige `const`. Es bastante raro que quieras tener una dirección de memoria asociada con tu constante, también, el usar const permite optimizaciones como propagación de constantes no solo en tu crate si no en los crates subyacentes.

% Atributos

Las declaraciones pueden ser anotadas con 'atributos' en Rust. Los atributos lucen así:

```
#[test]
# fn foo() {}
```

o de esta manera:

```
# mod foo {
#![test]
# }
```

La diferencia entre los dos es el `!`, que cambia a que cosa aplica el atributo:

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

El atributo `#[foo]` aplica a el siguiente item, que es la declaración del `struct`. El atributo `#![bar]` aplica a el item que lo encierra, la declaración `mod`. De resto, son lo mismo. Ambos cambian de alguna forma el significado de el item al cual están asociados.

Por ejemplo, considera una función como esta:

```
#[test]
fn comprobar() {
    assert_eq!(2, 1 + 1);
}
```

Esta marcado con `#[test]`. Esto significa que es especial: cuando ejecutes las pruebas, esta función sera ejecutada. Cuando compilas normalmente, no sera incluida ni siquiera. La función es ahora una función de prueba.

Los atributos pueden tener también data adicional:

```
#[inline(always)]
fn fn_super_rapida() {
# }
```

O incluso claves y valores:

```
#[cfg(target_os = "macos")]
mod solo_macos {
# }
```

Los atributos son usados para un numero de cosas diferentes en Rust. Hay una lista de atributos completa en la referencia. Actualmente, no tienes permitido crear tus propios atributos, el compilador de Rust los define.

% Alias `type`

La palabra reservada `type` te permite declarar un alias a otro tipo:

```
type Nombre = String;
```

Ahora, puedes usar este tipo como si fuera un tipo real:

```
type Name = String;

let x: Nombre = "Hola".to_string();
```

Sin embargo, nota, que es un *alias*, no un nuevo tipo. En otras palabras, debido a que Rust es fuertemente tipificado, podrías esperar que una comparación entre dos tipos diferentes falle:

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

lo anterior, produce:

```
error: mismatched types:
 expected `i32`,
    found `i64`
(expected i32,
    found i64) [E0308]
     if x == y {
              ^
```

Pero, si tuviéramos un alias:

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

Compilaría sin errores. Valores de un tipo `Num` son lo mismo que un valor de tipo `i32`, en todos los aspectos.

Puedes también hacer uso de alias de tipos en genéricos:

```
use std::result;

enum ErrorConcreto {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ErrorConcreto>;
```

El código anterior, crea una version especializada de el tipo `Result`, la cual posee siempre un `ErrorConcreto` para la parte `E` de `Result<T, E>`. Esta practica es usada comúnmente en la biblioteca estándar para la creación de errores personalizados para cada sub-seccion. Por ejemplo, io::Result.

% Conversión Entre Tipos

Rust, con su foco en seguridad, proporciona dos formas diferentes de conversión entre tipos. La primera, `as`, es para conversión segura. En contraste, `transmute` permite conversion arbitraria, y es una de las características mas peligrosas de Rust!

`as`

La palabra reservada `as` lleva a cabo conversion básica:

```
let x: i32 = 5;

let y = x as i64;
```

`as`, sin embargo, solo permite ciertos tipos de conversión:

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // cuatro ochos hacen 32
```

Lo anterior produce un error:

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // cuatro ochos hacen 32
        ^~~~~~~~
```

Es una 'conversion no escalar' ('non-scalar cast') porque tenemos multiples valores: los cuatro elementos en el arreglo. Este tipo de conversiones son muy peligrosas, debido a que asumen cosas acerca de como las multiples estructuras subyacentes están implementadas. Para esto, necesitamos algo mas peligroso.

`transmute`

La función `transmute` es proporcionada por una intrínseca del compilador, y lo que hace es muy simple, pero al mismo tiempo muy peligroso. Le dice a Rust que trate un valor de un tipo como si fuera un valor de otro tipo. Lleva acabo esto sin respetar el sistema de tipos, y confía completamente en ti.

En nuestro ejemplo anterior, sabemos que nuestro arreglo de cuatro `u8` s representa un `u32` de manera correcta, por lo tanto queremos hacer la conversión. Usando `transmute` en lugar de `as` Rust nos permite:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u32>(a);
}
```

Debemos envolver la operación en un bloque `unsafe` para que el código anterior compile satisfactoriamente. Técnicamente, solo la llamada a `mem::transmute` necesita estar dentro del bloque, pero en este caso esta bien tener todo lo relacionado a la conversion de manera que sepas en donde buscar. En este caso, los detalles acerca de `a` son también importantes, es por ello que están dentro del bloque. Veras código en cualquiera de los dos estilos, algunas veces el contexto esta tan lejos que envolver todo el código en un bloque `unsafe` no es una gran idea.

Mientras que `transmute` hace muy pocos chequeos, se asegura, al menos, que los tipos sean del mismo tamaño. Lo siguiente, falla:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4],="" u64="">(a);
}
```

con:

```
error: transmute called on types with different sizes: [u8; 4] (32 bits) to u64
(64 bits)
```

Dicho esto, estás por tu cuenta!

% Tipos Asociados

Los tipos asociados son una parte poderosa del sistema de tipos de Rust. Se relacionan con la idea de una 'familia de tipos', en otras palabras, la agrupación de multiples tipos. Esa descripción es una poco abstracta, es mejor que nos adentremos de una vez en un ejemplo. Si queremos escribir un trait `Grafo`, tenemos dos tipos por encima de los cuales debemos ser genéricos: el tipo de los nodos y el tipo de los vertices. Podríamos escribir un trait, `Grafo<N, V>` como este:

```
trait Grafo<N, V> {
    fn tiene_vertice(&self, &N, &N) -> bool;
    fn vertices(&self, &N) -> Vec<V>;
    // etc
}
```

Si bien esto de alguna manera funciona, termina siendo un poco raro. Por ejemplo, cualquier función que quiera recibir un `Grafo` como parámetro ahora también necesita ser genérica por sobre los tipos `N`odo y `V`ertice:

```
fn distancia>(grafo: &G, inicio: &N, fin: &N) -> u32 { ... }
```

Nuestro calculo de la distancia funciona sin tomar en cuenta nuestro tipo `Vértice`, en consecuencia la `V` en esta firma es simplemente una distracción.

Lo que realmente queremos expresar es que ciertos tipos `V`ertice y `N`odo vienen juntos para formar cada clase de `Grafo`. Podemos hacer esto con tipos asociados:

```
trait Grafo {
    type N;
    type V;

    fn tiene_vertice(&self, &Self::N, &Self::N) -> bool;
    fn vertices(&self, &Self::N) -> Vec<Self::V>;
    // etc
}
```

Ahora, nuestros clientes pueden abstraerse por encima de un determinado `Grafo`:

```
fn distancia(grafo: &G, inicio: &G::N, fin: &G::N) -> u32 { ... }
```

Sin necesidad de lidiar con el tipo `V`ertice!

Echemos un vistazo con mayor detalle a todo esto.

Definiendo tipos asociados

Construyamos ese trait `Grafo`. He aquí la definición:

```
trait Graph {
    type N;
    type V;

    fn tiene_vertice(&self, &Self::N, &Self::N) -> bool;
    fn vertices(&self, &Self::N) -> Vec<Self::V>;
}
```

Simple. Los tipos asociados usan la palabra reservada `type`, y van dentro del cuerpo del trait en conjunto con las funciones.

Dichas declaraciones `type` pueden tener lo mismo que las funciones. Por ejemplo si deseáramos que nuestro tipo `N` implementase `Display`, de manera que pudiésemos imprimir los nodos, podríamos hacer lo siguiente:

```
use std::fmt;

trait Grafo {
    type N: fmt::Display;
    type V;

    fn tiene_vertice(&self, &Self::N, &Self::N) -> bool;
    fn vertices(&self, &Self::N) -> Vec<Self::V>;
}
```

Implementando tipos asociados

Justo como cualquier otro trait, los traits que usan tipos asociados hacen uso de la palabra reservada `impl` para proporcionar implementaciones. A continuación una implementación simple de Grafo:

```
# trait Grafo {
#     type N;
#     type V;
#     fn tiene_vertice(&self, &Self::N, &Self::N) -> bool;
#     fn vertices(&self, &Self::N) -> Vec<Self::V>;
# }
struct Nodo;

struct Vertice;

struct MiGrafo;

impl Grafo for MiGrafo {
    type N = Nodo;
    type V = Vertice;

    fn tiene_vertice(&self, n1: &Nodo, n2: &Nodo) -> bool {
        true
    }

    fn vertices(&self, n: &Nodo) -> Vec<Vertice> {
        Vec::new()
    }
}
```

Esta tonta implementación siempre retorna `true` y un `Vec<Vertice>` vacío, pero te da una idea de como se implementa este tipo de traits con tipos asociados. Primero necesitamos tres `struct`s, una para el grafo, una para el nodo, y una para el vértice. De haber tenido mas sentido usar un tipo diferente, también hubiese funcionado, ahora usaremos `struct`s para los tres.

Lo siguiente es la linea `impl`, que es idéntica a la implementación de cualquier otro trait.

De aquí en adelante, usamos `=` para definir nuestros tipos asociados. El nombre que el trait usa va del lado izquierdo del `=`, y el tipo en concreto para el que estamos implementando este trait va del lado derecho. Finalmente, podemos usar tipos concretos en nuestras declaraciones de función.

Objetos trait con tipos asociados

Hay otra sintaxis de la cual debemos hablar: objetos trait. Si deseáramos crear un objeto trait a partir de un tipo asociado de esta manera:

```
# trait Grafo {
#     type N;
#     type V;
#     fn tiene_vertice(&self, &Self::N, &Self::N) -> bool;
#     fn vertices(&self, &Self::N) -> Vec;
# }
# struct Nodo;
# struct Vertice;
# struct MiGrafo;
# impl Grafo for MiGrafo {
#     type N = Node;
#     type V = Vertice;
#     fn tiene_vertice(&self, n1: &Nodo, n2: &Nodo) -> bool {
#         true
#     }
#     fn vertices(&self, n: &Nodo) -> Vec {
#         Vec::new()
#     }
# }
let grafo = MiGrafo;
let obj = Box::new(grafo) as Box;
```

Obtendríamos estos dos errores:

```
error: the value of the associated type `V` (from the trait `main::Grafo`) must
be specified [E0191]
let obj = Box::new(grafo) as Box;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trait
`main::Grafo`) must be specified [E0191]
let obj = Box::new(grafo) as Box;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

No podemos crear objetos trait de esta manera, puesto a que no conocemos los tipos asociados. En su lugar podríamos escribir:

```
# trait Grafo {
#     type N;
#     type V;
#     fn has_edge(&self, &Self::N, &Self::N) -> bool;
#     fn edges(&self, &Self::N) -> Vec<Self::V>;
# }
# struct Nodo;
# struct Vertice;
# struct MiGrafo;
# impl Grafo for MiGrafo {
#     type N = Nodo;
#     type V = Vertice;
#     fn tiene_vertice(&self, n1: &Nodo, n2: &Nodo) -> bool {
#         true
#     }
#     fn vertices(&self, n: &Nodo) -> Vec<Vertice> {
#         Vec::new()
#     }
# }
let grafo = MiGrafo;
let obj = Box::new(grafo) as Box<Grafo<N=Nodo, V=Vertice>>;
```

La sintaxis `N=Nodo` nos permite crear un tipo concreto, `Nodo`, para el parámetro de tipo `N`. Lo mismo con `V=Vertice`. De no haber proporcionado esta restricción, no hubiéramos podido determinar contra cual `impl` debe ser usado el objeto trait.

% Tipos sin Tamaño

La mayoría de los tipos posee un tamaño particular, en bytes, que es conocido en tiempo de compilación. Por ejemplo un `i32` tiene un tamaño de treinta y dos bits, o cuatro bytes. Sin embargo, hay algunos tipos que son útiles de expresar, pero no tienen un tamaño definido. Dichos tipos son denominados tipos 'sin tamaño' ('unsized') o 'de tamaño dinámico' ('dynamically sized'). Un ejemplo es `[T]`. Dicho tipo representa cierto numero de `T` en secuencia. Pero no sabemos cuantos de ellos hay, y en consecuencia el tamaño es desconocido.

Rust entiende algunos de estos tipos, pero tienen algunas restricciones. Son tres:

1. Solo podemos manipular una instancia de un tipo sin tamaño a través de un apuntador. Un `&[T]` funciona perfecto, pero un `[T]` no.
2. Las variables y argumentos no pueden tener tipos de tamaño dinámico.
3. Solo el ultimo campo en un `struct` puede tener un tipo de datos de tamaño dinámico; los otros campos no. Las variantes de enums no deben tener tipos de tamaño dinámico como data.

Entonces, porque molestarse? Bueno, debido a que `[T]` puede solo ser usado detrás de un apuntador, de no tener soporte a nivel de lenguaje para tipos sin tamaño, seria imposible escribir lo siguiente:

```
impl Foo for str {
```

o

```
impl Foo for [T] {
```

En su lugar, tendrías que escribir:

```
impl Foo for &str {
```

Significando que esta implementación solo funcionaria para referencias, y no otro tipos de apuntadores. Con el `impl for str`, todos los apuntadores, incluyendo (en algún punto, hay algunos bugs que arreglar primero) apuntadores inteligentes definidos por el usuario, pueden usar este `impl`.

?Sized

Si deseas escribir una función que acepte un tipo de datos de tamaño dinámico, puedes usar limite de trait especial, `?Sized`:

```
struct Foo<T: ?Sized> {
    f: T,
}
```

Este `?`, leído como “T puede ser `Sized`”, significa que este limite de trait es especial: nos permite hacer match con mas tipos, no menos. Es justo casi como que cualquier `T` implícitamente es `T: Sized`, y el `?` deshace este comportamiento por defecto.

% Operadores y Sobrecarga

Rust permite una forma limitada de sobrecarga de operadores. Existen ciertos operadores que pueden ser sobrecargados. Para soportar un operador particular entre tipos, hay un trait en especifico que puedes implementar, el cual sobrecarga el operador.

Por ejemplo, el operador `+` puede ser sobrecargado con el trait `Add`:

```
use std::ops::Add;

#[derive(Debug)]
struct Punto {
    x: i32,
    y: i32,
}

impl Add for Punto {
    type Output = Punto;

    fn add(self, otro: Punto) -> Punto {
        Punto { x: self.x + otro.x, y: self.y + otro.y }
    }
}

fn main() {
    let p1 = Punto { x: 1, y: 0 };
    let p2 = Punto { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

En `main`, podemos hacer uso de `+` en nuestros dos `Puntos`s, puesto que hemos implementado `Add<Output=Punto>` para `Punto`.

Existe un numero de operadores que pueden ser sobrecargados de esta manera, y todos sus traits asociados viven en el modulo `std::ops`. Echa un vistazo a la documentación para una lista completa.

Implementar dichos traits sigue un patron. Analicemos a `Add` con mas detalle:

```
# mod foo {
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
# }
```

Hay en total tres tipos involucrados aqui: El tipo para el cual estas implementado `Add`, `RHS` (de right hand side), que por defecto es `Self`, y `Output`. Para una expresion `let z = x + y`, `x` es el tipo `Self`, `y` es el `RHS`, y `z` es el tipo `Self::Output`.

```
# struct Punto;
# use std::ops::Add;
impl Add<i32> for Punto {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // sumar un i32 a un Punto obteniendo un f64
# 1.0
    }
}
```

te permitira hacer lo siguiente:

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

Usando los traits de operador en estructuras genericas

Ahora que sabemos como están definidos los traits de operador, podemos definir nuestro trait `TieneArea` y nuestra estructura `Cuadrado` del capitulo de traits de una forma mas genérica:

```
use std::ops::Mul;

trait TieneArea<T> {
    fn area(&self) -> T;
}

struct Cuadrado<T> {
    x: T,
    y: T,
    lado: T,
}

impl<T> TieneArea<T> for Cuadrado<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.lado * self.lado
    }
}

fn main() {
    let s = Cuadrado {
        x: 0.0f64,
        y: 0.0f64,
        lado: 12.0f64,
    };

    println!("Area de s: {}", s.area());
}
```

Para `TieneArea` y `Cuadrado`, solo declaramos un parametro de tipo `T` y reemplazamos el `f64`. El bloque `impl` necesita mas modificaciones:

```
impl TieneArea for Cuadrado
        where T: Mul + Copy { ... }
```

El método `area` requiere que podamos multiplicar los lados, es por ello que declaramos que `T` debe implementar `std::ops::Mul`. Como `Add`, mencionado anteriormente, `Mul` toma un parámetro `Output`: debido a que sabemos que los números no cambian de tipo cuando son multiplicados, también lo definimos como `T`. `T` también debe soportar copiado, de manera que Rust no trate de mover `self.lado` a el valor de retorno.

% Coerciones Deref

La biblioteca estándar proporciona un trait especial `Deref` . Es usado normalmente para sobrecargar `*` , el operador de dereferencia:

```
use std::ops::Deref;

struct EjemploDeref<T> {
    valor: T,
}

impl<T> Deref for EjemploDeref<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.valor
    }
}

fn main() {
    let x = EjemploDeref { valor: 'a' };
    assert_eq!('a', *x);
}
```

Lo anterior es útil para escribir tipos personalizados de apuntadores. Sin embargo, hay una facilidad del lenguaje relacionada a `Deref` : las ‘coerciones deref’. He aquí la regla: Si tienes un tipo `U` , y este implementa `Deref<Target=T>` , los valores de `&U` harán coercion Automatica a un `&T` . A continuación un ejemplo:

```
fn foo(s: &str) {
    // tomar la cadena prestada por un segundo
}

// String implementa Deref<Target=str>
let owned = "Hola".to_string();

// entonces, esto funciona:
foo(&owned);
```

Usando un ampersand en frente del valor tomamos una referencia a el. Entonces `owned` es un `String` , `&owned` es un `&String` , y debido a `impl Deref<Target=str>` `para` String `,` &String `hará deref a` &str `que es tomado por foo()` .

Eso es todo. Dicha regla es uno de los únicos lugares en los que Rust hace conversiones automáticas por nosotros, pero al mismo tiempo agrega mucha flexibilidad. Por ejemplo el tipo `Rc<T>` implementa `Deref<Target=T>` , de manera que esto funciona:

```
use std::rc::Rc;

fn foo(s: &str) {
    // tomar la cadena prestada por un segundo
}

// String implementa Deref<Target=str>
let owned = "Hello".to_string();
let contado = Rc::new(owned);

// entonces, esto funciona:
foo(&contado);
```

Todo lo que hemos hecho es envolver nuestro `String` en un `Rc<T>`. Pero ahora podemos pasar el `Rc<String>` a cualquier lugar en el cual tengamos un a `String`. La firma de `foo` no cambio, pero funciona de igual forma con cualquiera de los dos tipos. Este ejemplo tiene dos conversiones: de `Rc<String>` a `String` y luego de `String` a `&str`. Rust hará esto tantas veces como sea posible hasta que los tipos coincidan.

Otra implementación muy común proporcionada por la biblioteca estándar es:

```
fn foo(s: &[i32]) {
    // tomar el pedazo prestado por un segundo
}

// Vec<T> implementa Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

Los vectores pueden hacer `Deref` a un slice.

Deref y llamadas a metodo

`Deref` también entrara en efecto cuando se llame a un método. Considera el siguiente ejemplo:

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
```

Aunque `f` es un `&&Foo` y `Foo` recibe a `&self`, lo anterior funciona. Todo esto como consecuencia de que todas estas cosas son lo mismo:

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

Un valor de tipo `&&&&&&&&&&&&&&&&Foo` puede aun tener llamadas a métodos definidos en `Foo` debido a que el compilador insertara tantas operaciones *sean necesarias para hacerlo funcionar. Y debido a que inserta* `` `s, se hace uso de `` Deref`.

% Macros

Por ahora, has aprendido mucho sobre las herramientas que Rust ofrece para abstraer y reutilizar código. Estas unidades de reutilización poseen una rica estructura semántica. Por ejemplo, las funciones tienen una firma de tipos, los parámetros de tipo tienen límites de traits y las funciones sobrecargadas deben pertenecer a un trait particular.

Esta estructura significa que las principales abstracciones en Rust poseen un poderoso mecanismo de chequeo en tiempo de compilación. Pero, el precio es una flexibilidad reducida. Si se identifica visualmente un patrón de código repetido, podría ser difícil o tedioso expresar ese patrón como una función genérica, un trait, o cualquier otro elemento de la semántica de Rust.

Las macros nos permiten abstraer a un nivel *sintáctico*. Una invocación de macro es la abreviatura de una forma sintáctica "expandida". Dicha expansión ocurre durante la compilación, antes de comprobación estática. Como resultado, las macros pueden capturar muchos patrones de reutilización de código que las abstracciones fundamentales de Rust no pueden.

El inconveniente es que el código basado en macros puede ser más difícil de entender, porque menos de las normas internas de Rust aplican. Al igual que una función ordinaria, una macro bien hecha se puede utilizar sin entender detalles de implementación. Sin embargo, puede ser difícil diseñar una macro con un buen comportamiento! Además, los errores de compilación en código de macros son más difíciles de entender, porque describen problemas en el código expandido, no a nivel del código fuente que usan los desarrolladores.

Estos inconvenientes hacen de las macros una "herramienta de último recurso". Lo anterior no quiere decir que las macros son malas; forman parte de Rust porque a veces son necesarias para código conciso y abstracto. Simplemente mantén en cuenta este equilibrio.

Definiendo una macro

Puede que ya hayas visto la macro `vec!`, utilizada para inicializar un vector con un numero cualquiera de elementos.

```
let x: Vec<u32> = vec![1, 2, 3];
# assert_eq!(&[1,2,3], &x);
```

Esto no puede ser una función ordinaria, porque acepta cualquier número de argumentos. Pero podemos imaginarlo como una abreviación sintáctica para

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
# assert_eq!(&[1,2,3], &x);
```

Podemos implementar esta abreviatura, utilizando una macro: actual

> actual. La propia definición de `vec!` en libcollections difiere de la ↩

```
       presentada aquí, por razones de eficiencia y reutilización. Algunas
       de ellas son mencionadas en el [capitulo avanzado de macros][].
```

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
# fn main() {
#     assert_eq!(vec![1,2,3], [1, 2, 3]);
# }
```

¡Whoa!, un montón de sintaxis nueva. Examinémoslo parte por parte.

```
macro_rules! vec { ... }
```

Lo anterior dice que estamos definiendo una macro llamada `vec`, al igual que `fn vec` definiría una función llamada `vec`. En prosa, informalmente escribimos el nombre de una macro con un signo de exclamación, por ejemplo, `vec!`. Este signo de exclamación es parte de la sintaxis de invocación y sirve para distinguir una macro de una función ordinaria.

Coincidencia de patrones

Una macro se define a través de una serie de reglas, las cuales son casos de coincidencia de patrones. Anteriormente vimos

```
( $( $x:expr ),* ) => { ... };
```

Esto es similar a un brazo de una expresión `match`, pero las pruebas para coincidencia ocurren sobre los árboles de sintaxis Rust en tiempo de compilación. El punto y coma es opcional en el caso final (aquí, el único caso). El "patrón" en el lado izquierdo del `=>` es conocido como un 'matcher'. Los matchers tienen su propia pequeño gramática dentro del language.

El matcher `$x:expr` coincidirá con cualquier expresión Rust, asociando ese árbol sintáctico a la 'metavariable' `$x`. El identificador `expr` es un 'especificador fragmento'; todas las posibilidades se enumeran en el capitulo avanzado de macros. Al rodear el matcher con `$(...),*`, coincidirá con cero o más expresiones separadas por comas.

Ademas de la sintaxis especial de matchers, los tokens Rust que aparecen en un matcher deben coincidir de manera exacta. Por ejemplo:

```
macro_rules! foo {
    (x => $e:expr) => (println!("modo X: {}", $e));
    (y => $e:expr) => (println!("modo Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

imprimirá

```
modo Y: 3
```

Con

```
foo!(z => 3);
```

obtenemos el error del compilador

```
error: no rules expected the token `z`
```

Expansión

El lado derecho de una regla macro es sintaxis Rust ordinaria, en su mayor parte. Pero podemos insertar partes de sintaxis capturadas por el matcher. Del ejemplo original:

```
$(
    temp_vec.push($x);
)*
```

Cada expresión coincidente `$x` producirá una sola expresión `push` en la expansión de la macro. La repetición se desarrolla en "lockstep" con la repetición en el matcher (más sobre esto en un momento).

Debido a que `$x` ya fue marcado como una coincidencia con una expresión, no repetimos `:expr` en el lado derecho. Además, no incluimos una coma separando como parte del operador de repetición. En cambio, tenemos un punto y coma que termina dentro del bloque repetido.

Otro detalle: la macro `vec` tiene *dos* pares de llaves en el lado derecho. A menudo se combinan de este modo:

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

Las llaves exteriores son parte de la sintaxis de `macro_rules!`. De hecho, se puede utilizar `()` o `[]` en su lugar. Simplemente delimitan el lado derecho como un todo.

Las llaves interiores son parte de la sintaxis expandida. Recuerda que la macro `vec!` se utiliza en un contexto de expresión. Para escribir una expresión con varias sentencias, entre ellas enlaces a variable, utilizamos un bloque. Si la macro se expande a una sola expresión, no necesitas dicha capa extra de llaves.

Hay que tener también en cuenta que nunca *declaramos* que la macro produce una expresión. De hecho, esto no se determina hasta que usamos la macro como una expresión. Con cuidado, se puede escribir una macro cuya expansión funcione en varios contextos. Por ejemplo, la abreviatura de un tipo de datos podría ser válida como una expresión o un patrón.

Repetición

El operador de repetición sigue dos reglas principales:

1. `$(...)*` camina a través de una "capa" de repeticiones, para todos los `$nombre` s que contiene, al mismo paso, y
2. cada `$nombre` debe estar bajo al menos tantos `$(...)*` como los que fue comparado. Si esta bajo más, sera duplicado, según el caso.

Esta macro barroca ilustra la duplicación de las variables de los niveles de repetición exteriores.

```
macro_rules! o_O {
    (
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

Esa es la mayor parte de la sintaxis de los matcher. Todos los ejemplos utilizan `$(...)*`, que coincide con "cero o más" elementos sintácticos. Alternativamente, puedes escribir `$(...)+` que coincide con "uno o más". Ambas formas incluyen, opcionalmente, un separador, que puede ser cualquier token excepto `+` o `*`.

Este sistema se basa en "Macro-by-Example" (PDF).

Higiene

Algunos lenguajes implementan macros con sustitución de texto simple, lo que trae como consecuencia diversos problemas. Por ejemplo, este programa C imprime `13` en lugar de la esperada `25`.

```
#define CINCO_VECES(x) 5 * x

int main() {
    printf("%d\n", CINCO_VECES(2 + 3));
    return 0;
}
```

Después de la expansión tenemos `5 * 2 + 3`, en donde la multiplicación tiene mayor precedencia que la suma. Si has utilizado muchos macros en C, probablemente conoces los idiomas estándar para evitar este problema, así como cinco o seis otros. En Rust, no nos preocupamos por ello.

```
macro_rules! cinco_veces {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, cinco_veces!(2 + 3));
}
```

La metavariable `$x` se analiza como un nodo de expresión individual, y mantiene su lugar en el árbol de sintaxis, incluso después de la sustitución.

Otro problema común en los sistemas de macro es la "captura de variable". Aquí hay una macro C, utilizando una extensión de GNU C para emular bloques de expresión de Rust.

```
#define LOG(msj) ({ \
    int estado = obtener_estado_log(); \
    if (estado > 0) { \
        printf("log(%d): %s\n", estado, msj); \
    } \
})
```

He aquí un caso de uso que va terriblemente mal:

```
const char *estado = "estrías reticuladas";
LOG(estado)
```

Lo anterior se expande a

```
const char *estado = "estrías reticuladas";
int estado = obtener_estado_log();
if (estado > 0) {
    printf("log(%d): %s\n", estado, estado);
}
```

La segunda variable llamada `estado` sobreescribe a la primera. Esto es un problema porque la expresión de impresión (`printf`) debe hacer referencia a ambas.

La macro Rust equivalente tiene el comportamiento deseado.

```
# fn obtener_estado_log() -> i32 { 3 }
macro_rules! log {
    ($msj:expr) => {{
        let estado: i32 = obtener_estado_log();
        if estado > 0 {
            println!("log({}): {}", estado, $msj);
        }
    }};
}

fn main() {
    let estado: &str = "estrías reticuladas";
    log!(estado);
}
```

Esta versión funciona porque Rust tiene un sistema de macros higiénico. Cada expansión de macro ocurre en un "contexto de sintaxis" distinto, y cada variable está asociada con el contexto de sintaxis donde fue introducida. Es como si la variable `estado` dentro `main` está pintado de un "color" diferente de la variable `estado` dentro de la macro, y por lo tanto no entran en conflicto.

El sistema de macros de Rust también restringe la capacidad de las macros para introducir nuevos enlaces en el sitio de invocación. Código como el siguiente no funcionará:

```
macro_rules! foo {
    () => (let x = 3);
}

fn main() {
    foo!();
    println!("{}", x);
}
```

En lugar de eso necesitas pasar el nombre de la variable en la invocación, de manera que sea estiquetado con el contexto de sintaxis correcto.

```
macro_rules! foo {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!(x);
    println!("{}", x);
}
```

Lo anterior es válido para enlaces `let` y etiquetas de bucle, pero no para items. Así que el siguiente código compila:

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

Macros recursivas

La expansión de una macro puede incluir más invocaciones a macro, incluyendo invocaciones de la misma macro que esta siendo expandida. Estas macros recursivas son útiles para el procesamiento de entrada con estructura de árbol, como se ilustra en esta (simplista) taquigrafía HTML:

```
# #![allow(unused_must_use)]
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
#   // FIXME(#21826)
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

Depurando código de macro

Para ver los resultados de las macros en expansión, ejecuta `rustc --pretty expanded`. La salida representa todo un crate, por lo que también puede alimentar de nuevo a `rustc`, que a su vez producirá mejores mensajes de error que la compilación inicial. Es importante destacar que la salida de `--pretty expanded` puede tener un significado diferente si varias variables del mismo nombre (pero diferentes contextos sintácticos) están en juego en el mismo ámbito. En este caso `--pretty expanded,hygiene` te dirá acerca de los contextos de sintaxis.

`rustc` ofrece dos extensiones de sintaxis que ayudan con la depuración de macros. Por ahora, son inestables y requieren puertas de características (feature gates).

- `log_syntax!(...)` imprimirá sus argumentos en la salida estándar, en tiempo de compilación, y se "expandirá" a nada.

- `trace_macros!(true)` habilitara un mensaje compilador cada vez que una macro es expandida. Use `trace_macros!(false)` adelante en la expansión para apagarlo.

Mas información

El capitulo avanzado de macros entra en mas detalles acerca de la sintaxis de macros. También describe como compartir macros entre diferentes crates y módulos.

% Apuntadores Planos

% Unsafe

La principal atracción de Rust son sus poderosas garantías estáticas acerca de comportamiento. Pero los chequeos de seguridad son conservadores por naturaleza: existen programas que son en efecto seguros, pero el compilador no es capaz de de verificar que esto sea cierto. Para escribir ese tipo de programas, debemos decirle al compilador que relaje un poco sus restricciones. Para ello, Rust posee una palabra reservada, `unsafe`. El código que hace uso de `unsafe` posee menos restricciones que el código normal.

Repasemos la sintaxis, y luego hablaremos de la semántica. `unsafe` es usado en cuatro contextos. El primero es para marcar una función como insegura:

```
unsafe fn peligro_will_robinson() {
    // cosas peligrosas
}
```

Todas las funciones llamadas desde FFI deben ser marcadas como `unsafe`, por ejemplo. El segundo uso de `unsafe` es un bloque unsafe:

```
unsafe {
    // osas peligrosas
}
```

El tercero es para traits unsafe:

```
unsafe trait Peligroso { }
```

Y la cuarta es para la `impl` ementación de uno de dichos traits:

```
# unsafe trait Peligroso { }
unsafe impl Peligroso for i32 {}
```

Es importante poder tener la capacidad de delinear código que podría posiblemente contener bugs que originen problemas graves. Si un programa Rust termina de manera abrupta (un segfault), puedes tener por seguro que es en algún lugar de las secciones marcadas como `unsafe`.

Que significa ‘seguro’?

Seguro, en el contexto de Rust, significa 'no hacer nada inseguro'. Es importante saber que hay ciertos comportamientos que son probablemente indeseables en tu código, pero son expresamente *no* inseguros:

- Deadlocks.
- Perdida de memoria u otros recursos.
- Salida sin llamada a los destructores.
- Desbordamiento de enteros.

Rust no puede prevenir todos los tipos de problemas de software. Las cosas de la lista anterior no son buenas, pero tampoco califican como `unsafe` específicamente.

En adición, los siguientes son todos comportamiento indefinido en Rust, y deben ser evitadas, incluso cuando se escribe código `unsafe`:

- Condiciones de carrera.
- Deereferenciar un apuntador nulo/colgante.
- Lectura de memoria undef (memoria no inicializada)
- Violación de las reglas de aliasing de apuntadores a través de apuntadores planos.
- `&mut T` y `&T` siguen el modelo noalias de LLVM, excepto cuando el `&T` contiene un `UnsafeCell<U>`. El código unsafe no debe violar esas garantías de aliasing.
- Mutar un valor/referencia inmutable sin un `UnsafeCell<U>`.
- Invocar comportamiento indefinido a través de intrínsecos del compilador:
  - Indexar por fuera de los limites de un objeto con `std::ptr::offset` (intrínseco `offset`), con la excepción de un solo byte después del final lo cual es permitido.
  - Usar `std::ptr::copy_nonoverlapping_memory` (intrínsecos `memcpy32` / `memcpy64`) en buffers que se solapen.
- Valores inválidos en tipos primitivos, incluso en campos/variables locales privadas:
  - Referencias null, referencias colgantes o boxes.
  - Un valor distinto que `false` (0) o `true` (1) en un `bool`.
  - Un discriminante en un `enum` que no este incluido en su definición de tipo.
  - Un valor en un `char` el cual es un sustituto o por encima de `char::MAX`.
  - Una secuencia de bytes no-UTF en un `str`.
- Unwinding en Rust desde código foraneo o unwinding desde Rust a código foraneo.

Superpoderes Unsafe

En ambos funciones y bloques unsafe, Rust te permitirá hacer tres cosas que normalmente no podrías hacer. Solo tres. Y son:

1. Acceder o actualizar una variable mutable estética.

2. Dereferenciar un apuntador plano.
3. Llamar a funciones `unsafe`. Esta es la habilidad mas importante.

Eso es todo. Es importante destacar que `unsafe`, por ejemplo, no 'apaga el comprobador de prestamos'. Agregar de manera aleatoria `unsafe` a algún código no cambia su semántica, no comenzara a aceptar algo. Pero te permitirá escribir cosas que *si rompen* algunas de las reglas.

También encontraras la palabra reservada `unsafe` cuando escribas bindings a interfaces foráneas (no-Rust). Lo mas recomendable es escribir una segura interfaz nativa en Rust alrededor de los métodos proporcionados por la librería.

Echemos un vistazo a las tres habilidades listadas, en orden.

Acceder o actualizar una `static mut`

Rust posee una facilidad denominada '`static mut`' que te permite hacer estado global mutable. Hacerlo puede causar una condición de carrera, y en consecuencia es inherentemente inseguro. Para mayor detalle, dirígete a la sección static del libro.

Dereferenciar un apuntador plano

Los apuntadores planos te permiten llevar a cabo aritmética de punteros arbitraria, y pueden causar un numero de problemas de seguridad. En algunos sentidos, la habilidad de dereferenciar un apuntador arbitrario es una de las cosas mas peligrosas que puedes hacer, mas información en su sección en el libro.

Llamar funciones unsafe

Esta ultima habilidad funciona con ambos aspectos de `unsafe`: puedes solo llamar a funciones marcadas como `unsafe` desde dentro de un bloque unsafe.

Esta habilidad es poderosa y variada. Rust expone algunos intrínsecos del compilador como funciones unsafe, y algunas funciones unsafe hacen bypass de algunos chequeos de seguridad, intercambiando seguridad por velocidad.

Lo repetiré de nuevo: aun y cuando *puedes* hacer cosas arbitrarias en bloques unsafe y funciones no significa que debas hacerlo. El compilador actuara como si tu eres el responsable estuvieses de mantener arriba todas las invariantes, así que debes ser cuidadoso!

% Rust Nocturno

Rust proporciona tres canales de distribución: nocturno (nightly), beta y estable (stable). Las facilidades inestables están solo disponibles en Rust nocturno. Para mayor detalle acerca de este proceso, véase 'Estabilidad como un entregable'.

Para instalar Rust nocturno, puedes usar `rustup.sh`:

```
$ curl -s https://static.rust-lang.org/rustup.sh | sh -s -- --channel=nightly
```

Si tienes alguna duda acerca de la [inseguridad potencial] del uso de `curl | sh`, por favor, continua leyendo y encontraras nuestro disclaimer. Siéntete también en libertad de usar una version de dos pasos de la instalación examinando nuestro script:

```
$ curl -f -L https://static.rust-lang.org/rustup.sh -O
$ sh rustup.sh --channel=nightly
```

Si estas en Windows, por favor descarga bien sea el instalador de 32 bits o el instalador de 64 bits y ejecutalo.

Desinstalando

Si has decidido que ya no quieres mas a Rust, estaremos un poco triste, pero esta bien. No todos los lenguajes de programación son para todo el mundo. Simplemente ejecuta el script de desinstalación:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

Si usaste el instalador de Windows, simplemente ejecuta el `.msi` de nuevo y este te dará la opción de desinstalación.

Algunas personas, y de alguna forma con derecho, se sienten perturbados cuando les decimos que hagan `curl | sh`. Básicamente, cuando haces esto, estas confiando que la buena gente que mantiene Rust no van a hackear tu computadora y hacer cosas malas. Eso es =buen instinto! Si eres una de esas personas, por favor echa un vistazo a la documentación acerca de la instalación de Rust desde fuentes. O las descargas de los ejecutables oficiales.

Ah, debemos mencionar las plataformas oficialmente soportadas:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 o mayor, varias distribuciones), x86 and x86-64

- OSX 10.7 (Lion) o mayor, x86 and x86-64

Nosotros, el equipo de Rust probamos Rust de manera extensiva en esas plataformas, y otras pocas mas, como Android. Pero estas son las mas propensas en funcionar correctamente, debido a que son las que poseen mas pruebas.

Finalmente, un comentario acerca de Windows. Rust considera a Windows como una plataforma de primera clase en lo que se refiere al release, pero si somos honestos, la experiencia en Windows no esta tan integrada como las experiencias en Linux y OS X. Estamos trabajando en ello! Si algo no funciona, es un bug. Por favor haznos saber si eso pasa. Cada commmit es probado en Windows justo como cualquier otra plataforma.

Si has instalado Rust, puedes abrir una terminal, y escribir:

```
$ rustc --version
```

Deberías ver el numero de version, el hash de commit, fecha del commit y fecha de compilación:

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04) (built 2015-01-06)
```

Si lo ves, Rust ha sido instalado satisfactoriamente! Felicidades!

Este instalador también instala una copia local de la documentación, de manera que puedas leerla offline. En sistemas UNIX, la locación es `/usr/local/share/doc/rust`. En windows, se ubica en el directorio `share/doc`, en donde sea que hayas instalado Rust.

Si no, hay una variedad de lugares en los que puedes solicitar ayuda. El mas fácil es el canal #rust en irc.mozilla.org, el cual puedes acceder via Mibbit. Cliquea ese enlace, y estarás chateando con Rutaceos (un tonto apodo con el que nos referimos a nosotros), y podremos ayudarte. Otros muy buenos recursos incluyen el foro de usuarios, y Stack Overflow.

% Plugins del Compilador

Introducción

`rustc` puede cargar plugins de compilador, que son bibliotecas de usuario que extienden el comportamiento del compilador con nuevas extensiones de sintaxis, lints, etc.

Un plugin puede ser una biblioteca dinámica con una función *registradora* designada se encarga registrar la extension con `rustc`. Otros crates pueden cargar estas extensiones a través del uso del atributo `#![plugin(...)]`. Dirígete a la documentación de `rustc_plugin` para mayor detalle acerca de la mecánica de la definición y carga de un plugin.

De estar presentes, los argumentos pasados como `#![plugin(foo(... args ...))]` no son interpretados por rustc. Son proporcionados al plugin a través del método `args` de `Registry`.

En la gran mayoría de los casos, un plugin solo debería ser usado a través de `#![plugin]` y no por medio de un item `extern crate`. Enlazar un plugin traería a libsyntax y librustc como dependencias de tu crate. Esto es generalmente indeseable a menos que estés construyendo un plugin. El lint `plugin_as_library` chequea estos lineamientos.

La practica usual es colocar a los plugins del compilador en su propio crate, separados de cualquier macro `macro_rules!` o código Rust ordinario que vayan a ser usados por los consumidores de la biblioteca.

Extensiones de sintaxis

Los plugins pueden extender la sintaxis de Rust de varias maneras. Una forma de extension de sintaxis son las macros procedurales. Estas son invocadas de la misma forma que las macros ordinarias, pero la expansión es llevada a cabo por código Rust arbitrario que manipula arboles de sintaxis en tiempo de compilación.

Escribamos un plugin `roman_numerals.rs` que implementa literales de enteros con números romanos.

```
#![crate_type="dylib"]
#![feature(plugin_registrar, rustc_private)]

extern crate syntax;
extern crate rustc;
extern crate rustc_plugin;
```

```
use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::TokenTree;
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacEager};
use syntax::ext::build::AstBuilder;  // trait para expr_usize
use rustc_plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box {

    static NUMERALS: &'static [(&'static str, usize)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
        ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
        ("I",    1)];

    if args.len() != 1 {
        cx.span_err(
            sp,
            &format!("argument should be a single identifier, but got {} arguments", args
        return DummyResult::any(sp);
    }

    let text = match args[0] {
        TokenTree::Token(_, token::Ident(s, _)) => s.to_string(),
        _ => {
            cx.span_err(sp, "argument should be a single identifier");
            return DummyResult::any(sp);
        }
    };

    let mut text = &*text;
    let mut total = 0;
    while !text.is_empty() {
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = &text[rn.len()..];
            }
            None => {
                cx.span_err(sp, "invalid Roman numeral");
                return DummyResult::any(sp);
            }
        }
    }

    MacEager::expr(cx.expr_usize(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}
```

Entonces podemos hacer uso de `rn!()` como cualquier otra macro:

```
#![feature(plugin)]
#![plugin(roman_numerals)]

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

Las ventajas de esto por encima de una simple `fn(&str) -> u32` son:

- La conversión (arbitrariamente compleja) es efectuada en tiempo de compilación.
- La validación de entrada es también llevada a cabo en tiempo de compilación.
- Puede ser extendida para su uso en patrones, lo caul efectivamente proporciona una forma de definir una sintaxis literal nueva para cualquier tipo de dato.

En adicion a las macros procedurales, puedes definir attributos `derive` y otros tipos de expansiones. Dirigete a `Registry::register_syntax_extension` y a el enum `SyntaxExtension`. Para un ejemplo mas involucrado con macros, ve a `regex_macros`.

Tips y trucos

Algunos de los tips de depuración de macros son aplicables.

Puedes usar `syntax::parse` para transformar arboles de tokens en elementos de sintaxis de mas alto nivel, como expresiones:

```
fn expandir_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P = parser.parse_expr();
```

Mirar a el código del parser `libsyntax` te dara una idea de como funciona la infraestructura de parseo.

Mantén los `Span` s de todo lo que parsees, para un mejor reporte de errores. Puedes envolver `Spanned` alrededor de tus estructuras de datos,

Llamar a `ExtCtxt::span_fatal` abortara de manera inmediata la compilación. Es mejor llamar a `ExtCtxt::span_err` retornando un `DummyResult`, de manera que el compilador pueda continuar encontrando mas errores.

Para imprimir fragmentos de sintaxis para depuración, pudes usar `span_note` en conjunción con `syntax::print::pprust::*_to_string`.

El ejemplo anterior produjo un literal de entero usando `AstBuilder::expr_usize`. Como alternativa a el trait `AstBuilder`, `libsyntax` proporciona un conjunto de macros quasiquote. Estos están indocumentados y un poco rústicos en los bordes. Sin embargo, la implementación puede ser un buen punto de partida para un quasiquote mejorado como una biblioteca plugin ordinaria.

Plugins lint

Los plugins pueden extender la infraestructura de lints de Rust con chequeos adicionales para estilo de código, seguridad, etc. Escribamos ahora un plugin `lint_plugin_test.rs` que nos advierte acerca de cualquier item llamado `lintme`.

```
#![feature(plugin_registrar)]
#![feature(box_syntax, rustc_private)]

extern crate syntax;

// Cargando rustc como un plugin para obtener las macros
#[macro_use]
extern crate rustc;
extern crate rustc_plugin;

use rustc::lint::{EarlyContext, LintContext, LintPass, EarlyLintPass,
                  EarlyLintPassObject, LintArray};
use rustc_plugin::Registry;
use syntax::ast;

declare_lint!(TEST_LINT, Warn, "Warn about items named 'lintme'");

struct Pass;

impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }
}

impl EarlyLintPass for Pass {
    fn check_item(&mut self, cx: &EarlyContext, it: &ast::Item) {
        if it.ident.name.as_str() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_early_lint_pass(box Pass as EarlyLintPassObject);
}
```

Entonces código como

```
#![plugin(lint_plugin_test)]

fn lintme() { }
```

producirá una advertencia del compilador:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

Los componentes de un plugin lint son:

- una o mas invocaciones `declare_lint!` , las cuales defines structs `Lint` .
- un struct manteniendo el estado necesario para el pass lint (aquí, ninguno);
- una implementación `LintPass` definiendo como chequear cada elemento de sintaxis. Un único `LintPass` puede llamar a `span_lint` para diferentes `Lint` s, pero debe registrarlos a todos a través del método `get_lints` .

Los passes lint son recorridos de sintaxis, pero corren en una etapa de la compilación en la que la información de tipos esta disponible. Los lints integrados de Rust usan mayormente la infraestructura de los plugins lint, y proveen ejemplos de como acceder a la información de tipos.

Los lints definidos por plugins son controlados por los flags y atributos usuales del compilador, e.j `#[allow(test_lint)]` o `-A test-lint` . estos identificadores son derivados del primer argumentos a `declare_lint!` , con las conversiones de capitalización y puntuación apropiadas.

Puedes ejecutar `rustc -W help foo.rs` para ver una lista de los lints que `rustc` conoce, incluyendo aquellos proporcionados por plugins cargados por `foo.rs` .

% Ensamblador en linea

Para manipulaciones de muy bajo nivel y por razones de desempeño, uno podría desear controlar la CPU de manera directa. Para esto Rust soporta el uso de ensamblador en linea a través de la macro `asm!`. La sintaxis es parecida a la de GCC & Clang:

```
asm!(plantilla ensamblador
   : operandos de salida
   : operandos de entrada
   : clobbers
   : opciones
   );
```

Cualquier uso de `asm` esta protegido por puertas de feature (requiere `#![feature(asm)]` en el crate) y por supuesto requiere de un bloque `unsafe`.

> **Nota**: los ejemplos proporcionados aquí están escritos en ensamblador x86/x86-64, pero todas las plataformas están soportadas.

Plantilla ensamblador

La `plantilla ensamblador` es el único parámetro mandatorio y debe ser un literal de cadena de caracteres (e.j. `""`)

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// otras plataformas
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }

fn main() {
    // ...
    foo();
    // ...
}
```

(Los `feature(asm)` y `#[cfg]`s son omitidos de ahora en adelante.)

Los operandos de salida, operandos de entrada, clobbers y opciones son todos opcionales pero debes agregar el numero correcto de `:` en caso de que desees saltarlos:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax"
    :
    :
    : "{eax}"
   );
# } }
```

Los espacios en blanco no son significativos:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax" ::: "{eax}");
# } }
```

Operandos

Los operandos de entrada y salida siguen el mismo formato: `restriccion1"(expr1), "restriccion2"(expr2), ..."`. Las expresiones de operandos de salida deben ser valores lvalue mutables, o valores aun no asignados:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn add(a: i32, b: i32) -> i32 {
    let c: i32;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn add(a: i32, b: i32) -> i32 { a + b }

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

Sin embargo, Si desearas usar operandos reales en esta posición, seria obligatorio colocar llaves `{}` alrededor del registro que deseas, y también estarías obligado a colocar el tamaño especifico del operando. Lo anterior es muy util para programación de muy bajo nivel, para la que es importante cual registro es el que sea usa.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# unsafe fn read_byte_in(puerto: u16) -> u8 {
let resultado: u8;
asm!("in %dx, %al" : "={al}"(resultado) : "{dx}"(puerto));
resultado
# }
```

Clobbers

Algunas instrucciones modifican registros los cuales de otra forma hubieran mantenido valores diferentes es por ello que usamos las lista de clobbers para indicar al compilador a que no asuma que ningún valor cargado en dichos registros se mantendrá valido.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
// Colocando el valor 0x200 en eax
asm!("mov $$0x200, %eax" : /* sin salidas */ : /* sin entradas */ : "{eax}");
# } }
```

Los registros de entrada y salida no necesitan ser listados puesto a que esa información ya esta comunicada por las determinadas restricciones. De lo contrario, cualquier otro registro usado ya sea de manera explicita o implícita debe ser listado.

Si el ensamblador cambia el registro de código de condición `cc` debe ser especificado como uno de los clobbers. Similarmente, si el ensamblador modifica memoria, `memory` debe ser también especificado.

Opciones

La ultima sección, `opciones` es específica de Rust. El formato es una lista de cadenas de caracteres separadas por comma (e.j: `:"foo", "bar", "baz"`). Es usado para especificar información extra acerca del ensamblador:

Las opciones validas actualmente son:

1. *volatile* - especificar esto es analogo a `__asm__ __volatile__ (...)` en gcc/clang.
2. *alignstack* - ciertas instrucciones esperan que la pila este alineada de cierta manera (e.j. SSE), especificar esto le indica al compilador que inserte su código de alineamiento de pila usual.
3. *intel* - uso de la sintaxis de intel en lugar de la AT&T.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() {
let resultado: i32;
unsafe {
   asm!("mov eax, 2" : "={eax}"(resultado) : : : "intel")
}
println!("eax es actualmente {}", resultado);
# }
```

Mas Información

La implementación actual de la macro `asm!` es un binding directo a las expresiones ensamblador en linea de LLVM, así que asegurate de echarle un vistazo a su documentación para mayor información acerca de clobbers, restricciones, etc.

% No stdlib

% Intrínsecos

> **Nota**: los intrínsecos por siempre tendrán una interfaz inestable, se recomienda usar las interfaces estables de libcore en lugar de intrínsecos directamente.

Estas son importadas como si fuesen funciones FFI, con el ABI especial `rust-intrinsic`. Por ejemplo, si uno estuviese en un contexto libre, pero desease poder hacer `transmute` entre tipos, y llevar a cabo aritmética de apuntadores eficiente, uno importaría dichas funciones via una declaración como:

```
#![feature(intrinsics)]
# fn main() {}

extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: isize) -> *const T;
}
```

Al igual que cualquier otra función FFI, estas son siempre `unsafe` de llamar.

% Items de Lenguaje

% Enlace Avanzado

Los casos comunes de enlace con Rust han sido cubiertos anteriormente en este libro, pero soportar el rango de posibilidades disponibles en lenguajes es importante para Rust para lograr una buena interacción con bibliotecas nativas.

Argumentos de enlace

Hay otra forma de decirle a `rustc` como personalizar el enlazado, y es via el atributo `link_args`. Este atributo es aplicado a los bloques `extern` y especifica flags planos que necesitan ser pasados a el enlazador cuando se produce un artefacto. Un ejemplo podría ser:

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
# fn main() {}
```

Nota que actualmente esta facilidad esta escondida detrás la puerta `feature(link_args)` debido a que no es una forma sancionada de efectuar enlazado. Actualmente `rutsc` envuelve a el enlazador del sistema (`gcc` en la mayoría de los sistemas, `link.exe` en MSVC), es por ello que tiene sentido proveer argumentos de linea de comando extra, pero no siempre sera este el caso. En el futuro `rustc` pudiera usar LLVM directamente para enlazar con bibliotecas nativas, escenario en donde `link_args` no tendría sentido. Puedes lograr el mismo efecto del atributo `link_args` con el argumento `-C link-args` de `rustc`.

Es altamente recomendado *no* usar este atributo, y en su lugar usar el mas formal `#[link(...)]` en los bloques `extern`.

Enlazado estatico

El enlace estático se refiere a el proceso de crear una salida que contenga todas las bibliotecas requeridas y por lo tanto no requiera que las bibliotecas estén instaladas en cada sistema en el que desees usar tu proyecto compilado. La dependencias puras Rust son enlazadas estáticamente por defecto de manera que puedas usar las bibliotecas y ejecutables creados sin instalar Rust en todos lados. En contraste, las bibliotecas nativas (e.j `libc` y `libm`) son usualmente enlazadas de manera dinámica, pero esto se puede cambiar y enlazarlas también de manera estática.

El enlace es un tópico muy dependiente de plataforma, y el enlace estático puede no ser posible en todas las plataformas. Esta sección asume cierta familiaridad con el enlace en tu plataforma.

Linux

Por defecto, todos los programas en Linux se enlazaran con la biblioteca `libc` del sistema en conjunto con otro grupo de bibliotecas. Veamos un ejemplo en una maquina linux de 64 bits con GCC y `glibc` (por lejos la `libc` mas común en Linux):

```
$ cat example.rs
fn main() {}
$ rustc example.rs
$ ldd example
        linux-vdso.so.1 =>  (0x00007ffd565fd000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fa81889c000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fa81867e000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fa818475000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fa81825f000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa817e9a000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fa818cf9000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fa817b93000)
```

El enlace dinámico en linux puede ser indeseable si deseas usar las facilidades de la nueva biblioteca en sistemas viejos o sistemas destino que no poseen las dependencias requeridas para ejecutar el programa.

El enlace estático es soportado a través de una `libc` alternativa, `musl`. Puedes compilar tu version de Rust con `musl` habilitado e instalarlo en un directorio personalizado con las siguientes instrucciones:

```
$ mkdir musldist
$ PREFIX=$(pwd)/musldist
$
$ # Construir musl
$ curl -O http://www.musl-libc.org/releases/musl-1.1.10.tar.gz
$ tar xf musl-1.1.10.tar.gz
$ cd musl-1.1.10/
musl-1.1.10 $ ./configure --disable-shared --prefix=$PREFIX
musl-1.1.10 $ make
musl-1.1.10 $ make install
musl-1.1.10 $ cd ..
$ du -h musldist/lib/libc.a
2.2M    musldist/lib/libc.a
$
$ # Construir libunwind.a
$ curl -O http://llvm.org/releases/3.7.0/llvm-3.7.0.src.tar.xz
$ tar xf llvm-3.7.0.src.tar.xz
$ cd llvm-3.7.0.src/projects/
llvm-3.7.0.src/projects $ curl http://llvm.org/releases/3.7.0/libunwind-3.7.0.src.tar.xz
llvm-3.7.0.src/projects $ mv libunwind-3.7.0.src libunwind
llvm-3.7.0.src/projects $ mkdir libunwind/build
llvm-3.7.0.src/projects $ cd libunwind/build
llvm-3.7.0.src/projects/libunwind/build $ cmake -DLLVM_PATH=../../.. -DLIBUNWIND_ENABLE_S
llvm-3.7.0.src/projects/libunwind/build $ make
llvm-3.7.0.src/projects/libunwind/build $ cp lib/libunwind.a $PREFIX/lib/
llvm-3.7.0.src/projects/libunwind/build $ cd ../../../../
$ du -h musldist/lib/libunwind.a
164K    musldist/lib/libunwind.a
$
$ # Construir Rust con musl habilitado
$ git clone https://github.com/rust-lang/rust.git muslrust
$ cd muslrust
muslrust $ ./configure --target=x86_64-unknown-linux-musl --musl-root=$PREFIX --prefix=$P
muslrust $ make
muslrust $ make install
muslrust $ cd ..
$ du -h musldist/bin/rustc
12K     musldist/bin/rustc
```

Ahora posees un Rust con `musl` habilitado! Y debido a que lo hemos instalado en un prefijo personalizado necesitamos asegurarnos de que nuestro sistema pueda encontrar los ejecutables y bibliotecas apropiadas cuando tratemos de ejecutarlo:

```
$ export PATH=$PREFIX/bin:$PATH
$ export LD_LIBRARY_PATH=$PREFIX/lib:$LD_LIBRARY_PATH
```

Probémoslo!

```
$ echo 'fn main() { println!("Hola!"); panic!("falla"); }' > example.rs
$ rustc --target=x86_64-unknown-linux-musl example.rs
$ ldd example
        not a dynamic executable
$ ./example
Hola!
thread '
' panicked at 'failed', example.rs:1
```

Exito! Este binario puede ser copiado a casi cualquier maquina Linux con la misma arquitectura y ser ejecutado sin ningún problema.

`cargo build` también permite la opción `--target` por medio de la cual puedes construir tus crates normalmente. Sin embargo, podrías necesitar recompilar tus bibliotecas nativas con `musl` antes de poder enlazar con ellas.

% Pruebas de Puntos de Referencia

% Sintaxis de Cajas y Patrones

% Patrones de Slice

% Constantes Asociadas

Con el feature `associated_consts`, puedes definir constantes como:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, i32::ID);
}
```

Cualquier implementador de `Foo` tendrá que definir `ID`. Sin la definición:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32;
}

impl Foo for i32 {
}
```

resulta en

```
error: not all trait items implemented, missing: `ID` [E0046]
     impl Foo for i32 {
     }
```

Un valor por defecto puede también ser implementado:

```
#![feature(associated_consts)]

trait Foo {
    const ID: i32 = 1;
}

impl Foo for i32 {
}

impl Foo for i64 {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, i32::ID);
    assert_eq!(5, i64::ID);
}
```

Como puedes ver, al implementar `Foo`, puedes dejarlo sin implementación, como en el caso de `i32`. Entonces este usara el valor por defecto. Pero, también y al igual que como en `i64`, podemos agregar nuestra propia definición.

Las constantes asociadas no necesitan solo funcionan con traits. Un bloque `impl` para una `struct` o un `enum` es también valido:

```
#![feature(associated_consts)]

struct Foo;

impl Foo {
    const FOO: u32 = 3;
}
```

% Investigacion Academica