Ownership

. para eliminar categorías completas de bugs

. restringir los programas que son considerados válidos

. que no se pague un costo en tiempo de ejecución

Reglas

1. Cada valor tiene una variable que es su dueña
2. Sólo puede existir una dueña a la vez
3. Cuando la dueña sale de alcance, el valor será liberado

* Cada valor tiene un único dueño que determina su lifetime. Cuando el dueño es liberado (dropped) el valor adueñado también.
* Una variable es dueña de su valor. Cuando el control sale del bloque en el cual la variable está declarada, la variable se destruye, por lo que su valor también se destruye.

| Tipo | Ownership | Alias | Mutable |
| --- | --- | --- | --- |
| T | adueñado |  | X |
| &T | referencia compartida | X |  |
| &mut T | referencia mutable |  | X |

Bases

* Un Box<T> es un puntero a un valor de tipo T almacenado en heap.
* Invocando Box::new(v) asigna espacio en heap, mueve el valor v a ese espacio y retorna un Box que apunta a ese espacio.
* Un Box es dueño del espacio al que apunta, de tal manera que cuando el Box es destruido, se libera su espacio también.

Movimientos

* Se pueden mover valores de un dueño a otro.
* La librería estándar provee tipos de punteros que llevan conteo de referencias como Rc y Arc. Esto permite tener múltiples dueños bajo ciertas restricciones.
* Se puede tomar prestado una referencia a un valor. Las referencias son punteros que no adueñan los valores a los que apuntan y tienen lifetimes limitados.

Operaciones que mueven

* Asignar un valor a una variable, pasarla a una función o retornarla de una función no copian el valor sino que lo mueven.
* El origen cede su pertenencia del valor al destinatario y pasa a ser no inicializada, el destinatario ahora controla el lifetime del valor.

Tipos que se copian

* Asignar un valor de un tipo que es Copy copia el valor en lugar de moverlo.
* El origen de la asignación permanece inicializado y usable con el mismo valor que tenía antes.
* Pasar tipos Copy a funciones y constructores opera similarmente.
* Cada movimiento es una copia no profunda byte a byte que deja al origen no inicializado.
* Las copias son similares solo que el origen permanece inicializado.
* Los tipos estándar Copy incluyen a todos los numéricos de enteros de máquina y punto flotante, char y bool y algunos más.
* Una tupla o arreglo de tamaño fijo cuyos componentes son Copy es a su vez Copy.
* Sólo tipos para los cuales una copia bit a bit alcanza pueden ser Copy.
* Cualquier tipo que necesita hacer algo especial cuando un valor es destruido puede no ser Copy.
* Struct y Enum no son Copy.
* Si todos los campos de Struct con Copy, se puede hacer que el tipo lo sea también usando un atributo.

Referencias

* Son otro tipo de puntero que no se adueñan del valor al que apunta sino que toma prestado (borrow).
* No tienen efecto sobre los lifetimes de sus referentes.
* No deben sobrevivir a sus referentes.
* Se le dice tomar prestado (borrowing) a crear una referencia a un valor.
* Son direcciones en memoria.
* Permite acceder a un valor sin afectar su pertenencia.
* Hay dos tipos:
  + **Compartida**: permite leer pero no modificar su referente. Se pueden tener a la vez tantas a un valor particular como se desee.
  + **Mutable**: se puede tanto leer como modificar. No se puede tener ninguna otra activa de ningún tipo a la misma vez.
* Nunca son nulas:
  + No existe un valor por defecto inicial.
  + No convierte enteros en referencias.
  + Para indicar la ausencia de un valor se usa Option<T>.

Lifetimes

Tiempo de vida en el código.

Seguridad de referencias

* Se asigna a cada tipo de referencia en un programa un lifetime que satisface las restricciones impuestas por cómo se usa.
* Un lifetime es un tramo del programa para el cual esa referencia se puede usar de manera segura.
* El lifetime de una variable debe contener o abarcar el lifetime de la referencia que toma prestada su valor.
* Más allá del punto donde una variable sale de alcance, la referencia sería un puntero colgante o inválido.
* Las lifetimes no se declaran. Son función únicamente del uso que se le da a valores y a sus referencias y las necesidades y restricciones que surgen de esto.

Genericidad y Traits

* Soporta polimorfismo a través de dos características relacionadas:
  + Traits: Versión de Rust e interfaces o clases base abstractas. Define un conjunto de métodos que pueden ser implementados para un tipo dado. Agregar un trait a un tipo no cuesta memoria e invocar los métodos de un trait para un tipo no tiene el sobrecosto de llamadas a métodos virtuales.
  + Generics: Otro tipo de polimorfismo en Ruts. Permiten parametrizar código con tipos, ya sea funciones o definiciones de otros tipos como structs o enums.

Implementar un trait sobre un tipo

* Podemos implementar un trait para un tipo <=> el trait o el tipo es local a nuestro crate.
* Es local al crate.
* No se puede implementar traits externos sobre tipos externos.
* El código de otros no puede romper el código del crate local, y viceversa. Sin esta regla dos crates podrían implementar el mismo trait para el mismo tipo, y el compilador no sabría cuál implementación usar.

Traits de utilidad

* Drop
* Sized
* Clone
* Copy
* Deref / DerefMut
* AsRef / AsRefMut
* From / Into
* ToOwned

Genericidad

La genericidad y traits están relacionados porque frecuentemente, al definir una función con un parámetro por tipo, se necesita restringir los tipos posibles a aquellos que implementen ciertas traits.