Tutorial de Rust

¿Qué es rust?

Rust es un lenguaje de programación de sistemas que corre increíblemente rápido, previene fallos de segmentación (segfaults) y garantiza seguridad en los threads.

Instalación

Para instalar Rust en sistemas Unix-like como linux y macOS solo abre la terminal y escribe el siguiente comando:

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

Si deseas instalar en windows puedes seguir el siguiente enlace https://win.rustup.rs/ que descargara rustup-init.exe

Escribiendo el “Hello world”

Para empezar solo tiene que crear un archivo que se puede llamar como usted desea, sólo tiene que terminar con la extensión *.rs,* para nuestro ejemplo lo llamaremos *main.rs.* Ahora ingresa el siguiente código:

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

Ingresamos los siguientes comando:

$ rustc main.rs

Ahora ya tenemos compilado nuestro código y hemos creado nuestro primer programa en rust. Puede correrse de la siguiente manera:

$ ./main
Hello, world!

Comentarios

Los comentarios en Rust son el clásico // para comentarios de una sola línea y /* */ para múltiples líneas de comentarios.

Variables Bindings

En Rust los tipos de variables pueden ser anotados cuando se declaran. Aunque en la mayoría de los casos el compilador será capaz de inferir el tipo de la variable a partir del contexto.

Valores pueden ser ligados a una variable utilizando la palabra reservada *let*

```
fn main() {
    let an_integer = 1u32;
    let a_boolean = true;
    let unit = ();
    let copied_integer = an_integer;

    println!("An integer: {:?}", copied_integer);
```

```
    println!("A boolean: {:?}", a_boolean);
    println!("Meet the unit value: {:?}", unit);
     // El compilador advierte sobre variables sin utilizar,
    //se puede utilizar una raya al piso al comienzo para silenciarlo
    let _unused_variable = 3u32;
    let noisy_unused_variable = 2u32;
}
```

Mutabilidad

Las variables son inmutables de manera predeterminada, pero esto puede ser sobreescrito usando el modificador *mut.*

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Before mutation: {}", mutable_binding);

    // Ok
    mutable_binding += 1;

    println!("After mutation: {}", mutable_binding);

    // Error!
    _immutable_binding += 1;

}
```

Funciones

Las funciones son declaradas utilizando la palabra reservada *fn*. Sus argumentos deben tener definido un tipo, y si la función retorna un valor su tipo debe ser especificado después de una flecha ->.

La expresión final de la función va a ser utilizada como el valor de retorno. Alternativamente se puede utilizar la palabra *return* para retornar el valor en cualquier lugar de la función.

```
fn main() {
    fizzbuzz_to(100);
}

// Función que retorna un booleano
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }

    // Esta es una expresión, the `return` no es necesario aquí
    lhs % rhs == 0
}

fn fizzbuzz(n: u32) -> () {
    if is_divisible_by(n, 15) {
        println!("fizzbuzz");
    } else if is_divisible_by(n, 3) {
        println!("fizz");
    } else if is_divisible_by(n, 5) {
        println!("buzz");
    } else {
        println!("{}", n);
    }
}
```

```
fn fizzbuzz_to(n: u32) {
    for n in 1..n + 1 {
        fizzbuzz(n);
    }
}
```

Ownership and moves

Las variables están a cargo de liberar sus propios recursos, así que los **recursos solo pueden tener un dueño.** Esto también previene que los recursos sean liberados más de una vez. Note que no todas las variables son dueños de un recurso.

Cuando estamos realizando las asignaciones o pasando argumentos por valor, la pertenencia de esos recursos es transferida. En Rust-speak, esto es conocido como *move.*

Después de mover los recursos, el previo dueño ya no puede ser utilizado. Esto previene crear apuntadores colgantes.

```
// This function takes ownership of the heap allocated memory
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // `c` is destroyed and the memory freed
}

fn main() {
    // _Stack_ allocated integer
    let x = 5u32;

    // *Copy* `x` into `y` - no resources are moved
    let y = x;
```

```
    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);

    // `a` is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);

    println!("a contains: {}", a);

    // *Move* `a` into `b`
    let b = a;
    // The pointer address of `a` is copied (not the data) into `b`.
    // Both are now pointers to the same heap allocated data, but
    // `b` now owns it.

    // Error! `a` can no longer access the data, because it no longer owns the
    // heap memory
    //println!("a contains: {}", a);
    // TODO ^ Try uncommenting this line

    // This function takes ownership of the heap allocated memory from `b`
    destroy_box(b);

    // Since the heap memory has been freed at this point, this action would
    // result in dereferencing freed memory, but it's forbidden by the compiler
    // Error! Same reason as the previous Error
    //println!("b contains: {}", b);
    // TODO ^ Try uncommenting this line
}
```

Borrowing

La mayoría del tiempo, nos gustaría acceder a los datos sin tomar la pertenencia sobre esos datos. Para lograr esto, Rust utiliza un mecanismo de préstamos. En vez de pasar los objetos por valor (T), los objetos pueden ser pasados por referencia (&T).

El compilador garantiza que las referencias siempre apuntan a objetos válidos. Eso quiere decir, que mientras las referencias a un objeto exista, el objeto no puede ser destruido.

```
// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}

fn main() {
    // Create a boxed i32, and a stacked i32
    let boxed_i32 = Box::new(5_i32);
    let stacked_i32 = 6_i32;

    // Borrow the contents of the box. Ownership is not taken,
    // so the contents can be borrowed again.
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);

    {
        // Take a reference to the data contained inside the box
        let _ref_to_i32: &i32 = &boxed_i32;

        // Error!
        // Can't destroy `boxed_i32` while the inner value is borrowed.
        eat_box_i32(boxed_i32);
        // FIXME ^ Comment out this line

        // `_ref_to_i32` goes out of scope and is no longer borrowed.
    }
    // `boxed_i32` can now give up ownership to `eat_box` and be destroyed
    eat_box_i32(boxed_i32);
}
```