RUST

Juan David Valencia
Andrés Felipe Guerrero

1 ¿Que 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.

¿Qué es rust?

Rust es un lenguaje de programación compilado, de propósito general y multiparadigma que está siendo desarrollado por Mozilla.

El diseñador original de rust es Graydon Hoare, actualmente trabaja en Swift

moz://a

Objetivos de Rust

Velocidad

- Lenguaje compilado
- Sin Garbage Collector
- ‘Zero-cost abstractions’

Seguridad

- Mutabilidad
- Comprobaciones durante la compilación
- Ownership / Borrowing

Concurrencia

- Wrappers con Sync y Send
- Seguridad en los threads

2

¿ Cómo obtener Rust ?

Instalar Rust

- rustup permite el manejo de versiones de rust, permitiendo el cambio entre builds estables, beta y nightly
- El instalador de Windows y el comando para la instalación en Unix se encuentran en la página oficial de Rust
- https://www.rust-lang.org/en-US/install.html

3 Generalidades del lenguaje

Comentarios e impresión

- Comentarios de una línea: //
- Comentarios multilinea: /* ... */
- Comentarios de doc: /// o //!

- Impresión a consola: print!
- Impresión a consola con salto de línea: println!
- Interpolación de Strings:

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

Tipos primitivos

- bool -> true , false
- char -> ‘x’ , ‘1’ , ‘♫’
- Tipos numéricos:
  - i8, i16, i32, i64
  - u8, u16, u32, u64
  - isize, usize
  - f32, f64
- Binario: 0b1100, 0b_1100_1010
- Octal: 0o14, 0o_14_12
- Hexadecimal: 0xc, 0o_a4_c3

Operadores

- Operadores aritméticos:
  - +, -, *, /, %
- Operadores relacionales:
  - ==, !=, >, >=, <, <=
- Operadores lógicos
  - &&, ||, !
- Otras funciones matemáticas:
  - abs(), pow(), sqrt(), log(), log10(), exp()

Variables

- Variable Bindings
  - `let x = 5;`
- Patterns
  - `let (x, y) = (1, 2);`
- Type annotation
  - `let x: i32 = 5;`
- Rust tiene inferencia de tipos

Variables

- Mutability
  - `let x = 42;`

    `x = 5;`

    ```
    error: re-assignment of immutable
    variable `x`
        x = 5;
        ^~~~~~
    ```

  - `let mut x = 42;`

    `x = 5;`

Variables

- Alcance de las variables y shadowing

```
let x: i32 = 8;
{
    println!("{}", x); // Imprime "8".
    let x = 12;
    println!("{}", x); // Imprime "12".
}
println!("{}", x); // Imprime "8".
let mut x =  “Hola mundo”;
println!("{}", x); // Imprime "Hola mundo".
```

Arreglos

Lista de elementos del mismo tipo de tamaño fijo

```
let a = [1, 2, 3];
let mut b = [4, 5, 6];
```

Los arreglos tienen tipo [T; N]

- ```
  let a = [1, 2, 3];
  println!("a tiene {} elementos", a.len());
  // Imprime "a tiene 3 elementos".
  ```
- ```
  let b = [4, 5, 6];
  println!("El 2 elemento es: {}", b[1]);
  ```
- // Imprime "El segundo elemento es 5".

Tuplas

- Una tupla es una lista ordenada de tamaño fijo
  - `let x = (1, "hello");`
  - `let x: (i32, &str) = (1, "hello");`
- Podemos asignar una tupla a otra si son del mismo tamaño y tipos
  - ```
    let mut x = (1, 2); // x: (i32, i32)
    let y = (2, 3); // y: (i32, i32)
    x = y;
    ```
- Destructuring let:
  - `let (x, y, z) = (1, 2, 3);`
    ```

Strings

En rust existen 2 tipos de String:

- &str:
  - ‘String slice’
  - Tamaño fijo e inmutables
  - `let greeting = "Hello there.";`
- String:
  - String guardada en un heap
  - El string puede cambiar de tamaño
  - `let mut s = "Hello".to_string();`
  - `let char = s.chars().nth(1);`
  - `let s3 = s1 + s2; // s1:String s2:&str`
  - `let s3 = s1 + &s2; // s1:String s2:String`

Funciones

- Todo programa tiene por lo menos la función main
  - `fn main() { }`
- Si la función tiene argumentos es obligatorio definir el tipo de dato
  - ```
    fn print_number(x: i32) {
        println!("x is: {}", x);
    }
    ```
- Podemos declarar el tipo de dato retornado
  - ```
    fn add_one(x: i32) -> i32 {
        x + 1
    }
    ```

If

- ```
  let x = 5;

  if x == 5 {
      println!("x is five!");
  } else if x == 6 {
      println!("x is six!");
  } else {
      println!("x is not five or six :(");
  }
  ```
- ```
  let x = 5;

  let y = if x == 5 { 10 } else { 15 };
  ```

Loops

- En rust hay 3 tipos de loops:
- Loop: loop infinito

```
loop {
    println!("Loop forever!");
}
```

- Funcionamiento similar a

```
while true{}
```

Sin embargo se recomienda usar loop en ese caso para mejorar la seguridad del código y el rendimiento

While

- While: Cuando estamos seguros de cuántas veces vamos a iterar

```
let mut x = 5;

while x > 1 {
    x -= 1;

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

For

- For: número exacto de iteraciones

```
for var in expression {
    code
}
```

  - Rangos:

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

  - Iteradores:

```
for item in array.iter() {
    println!("{}", item);
}
```

Loops

- Los loops nos ofrecen 2 palabras clave para modificar las iteraciones
  - break: termina todo el loop
  - continue: termina la iteración actual
- Rust tambien nos permite nombrar los loops:

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; }
        if y % 2 == 0 { continue 'inner; }
        println!("x: {}, y: {}", x, y);
    }
}
```

Match

- Reemplaza la estructura Switch o Case
- Toma una expresión y se ramifica según su valor
- ‘Exhaustiveness Checking’ -> _

```
let x = 1;
match x {
    1 => println!("one"),
    2 => println!("two"),
    _ => println!("something else"),
}
```

Vectores

- Arreglos de tamaño dinámico
- Tienen el tipo Vec<T> , es decir, pueden crearse vectores de cualquier tipo
- Se crean con el macro vec!

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
let v = vec![0; 10]; // Vector of 10 zeroes.

let i: usize = 0;
let j: i32 = 0;

v[i]; // Funciona!
v[j]; // Error
```

Iterar en un vector

- 3 maneras de iterar:
  - Usando una referencia:

```
for i in &v {
    println!("A reference to {}", i);
}
```

  - Usando una referencia mutable:

```
for i in &mut v {
    println!("A mutable reference to {}", i);
}
```

  - Tomando propiedad del vector:

```
for i in v {
    println!("Take ownership of the vector {}", i);
}
```

Structs

- Las estructuras son una manera de crear tipos de datos complejos.

```
struct Point {
    x: i32,
    y: i32,
}
let origin = Point { x: 0, y: 0 };
println!("The x coordinate is {}", origin.x);
```

- Tuple Structs:

```
struct Point(i32, i32, i32);
let origin = Point(0, 0, 0);
println!("The z coordinate is {}", origin.2);
```

Traits

Un trait le dice al compilador acerca de una funcionalidad que debe proveer un tipo de dato.

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Closures

Los closures son funciones que pueden capturar el ambiente que los rodea, permitiendo usar variables del scope en el que se declaran.

```
let closure = |arguments| {
    //code
};
```

```
let mut num = 5;
let plus_num = |x: i32| x + num;
```

Macros

- Los macros son similares a las funciones, aunque terminan con ! .
- Otorgan mayor flexibilidad que una función. Permiten aceptar cualquier numero de argumentos, entre otras ventajas.
- Rust provee varios macros muy útiles:
  - println!()
  - assert!()
- Se crean usando el macro macro_rules!
  -
```
macro_rules! foo {
    () => (fn x() { });
}
```

3

Ownership/Borrowing

Ownership

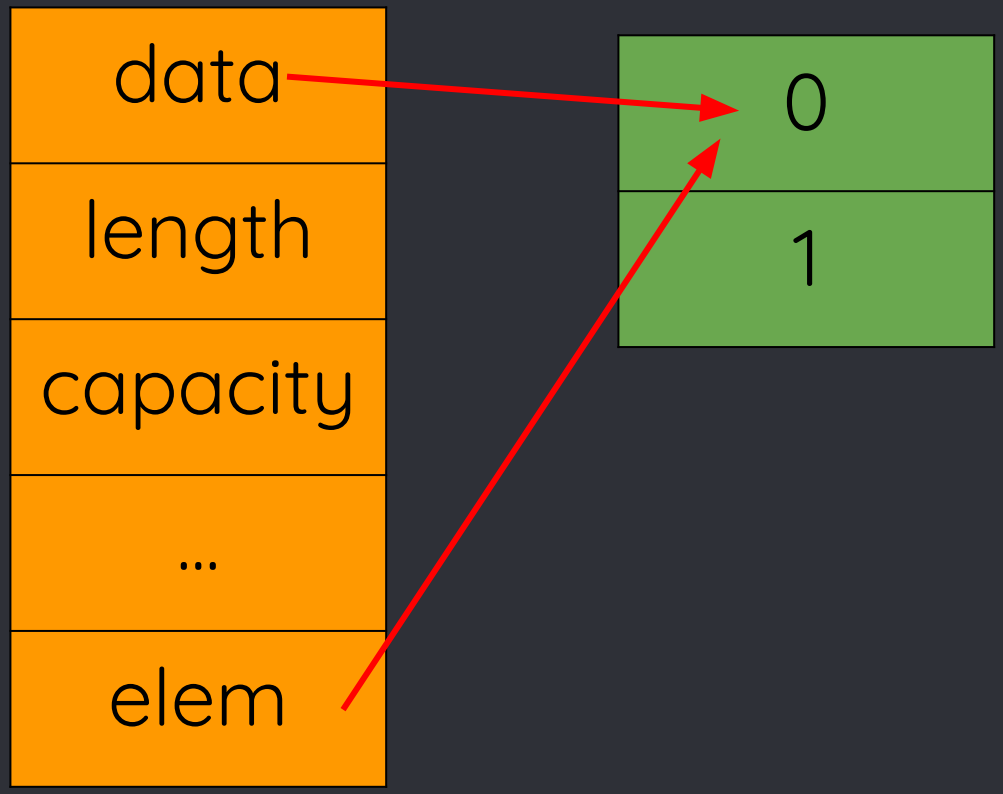
Es el sistema utilizado por rust para lograr su mayor objetivo: La seguridad de la memoria

¿Qué es seguridad de la memoria?

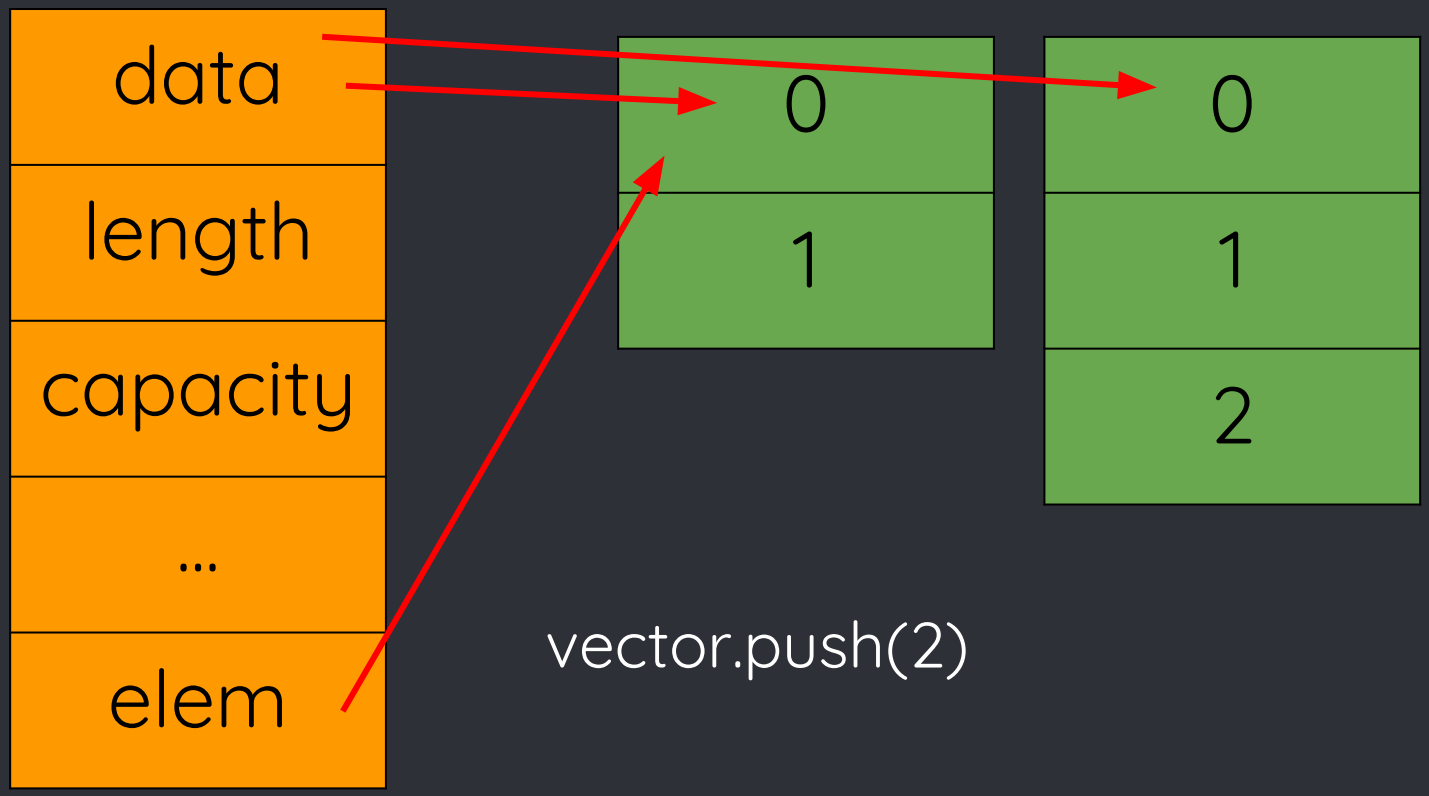
```cpp
int main() {
  vector<string> v(5);
  v[0] = "Hola mundo";
  ... Llenamos el vector
  auto& elem = v[0];
  cout << elem;
  v.push_back("Hello world");
  cout << elem;
}
```

```
Hola mundo
□□□□□□□□□□□□□□□6□□0□□6□□0□□6□□0□□6□□0□□6□□0□□6□□0□□6□□0□□6□□0
 ?6?□0 ?6?□0 ?6?□0 ?6?□0 ?6?□0 ?6?□0 ?6?□0 ?6?□0 □7?□0 □7?□
0 □7□□0 □7□□0 (7□□0 (7□□0 87□□0 87□□0 H7□□0 H7□□0 X7□□0 X7□□0 h7□□0
 h7?□0 x7?□0 x7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0
 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 ?7?□0 □8?
□0□□8□□0□□8□□0□□8□□0□(8□□0□(8□□0□88□□0□88□□0□H8□□0□H8□□0□X8□□0
 X8?□0 h8?□0 h8?□0 x8?□0 x8?□0 ?8?□0 ?8?□0 ?8?□0 ?8?□0 ?8?□0
?8?□0□?8?□0□?8?□0□?8?□0□?8?□0□?8?□0□?8?□0□?8?□0□?8?□0□?
8?□0□?8?□0□□9?□0□□9?□0□□9?□0□□9?□0□(9?□0□(9?□0□89?□0□89?□0
```

Aliasing



Mutation



Ownership

Las variables en Rust tienen “propiedad” sobre el valor al que están ligadas

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

¡ El vector **[1,2,3]** le **pertenece** a **v** !

Cuando el programa sale del scope de la variable, se limpia su espacio en memoria

Ownership

Rust se asegura de que solo exista **una** variable que tenga propiedad sobre un recurso dado

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

let v2 = v;

println!("v[0] is: {}", v[0]);

// error: use of moved value: `v`
```

Ownership

Los tipos primitivos implementan el ‘trait’ **Copy**, permitiendo reasignar variables sin perder la referencia inicial

```
let a = 5;

let b = number;

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

// a is 5
```

¿ Y si queremos regresar la propiedad ?

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) ->
(Vec<i32>, Vec<i32>, i32) {
    // Utilizar v1 y v2.

    // Retornar la propiedad de las variables y el
resultado de la función.
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Borrowing

El sistema de ‘préstamo’ nos permite crear referencias a un recurso existente, sin que este recurso cambie de propietario.

Las referencias no tienen control sobre el tiempo de vida del recurso.

Hay 2 tipos de referencias:

- Referencias compartidas
- Referencias mutables

Referencias compartidas

Las referencias compartidas nos permiten ‘prestar’ el recurso a distintas funciones, threads o bloques de código. Sin embargo, estas referencias son inmutables. Si se trata de modificar el recurso el compilador mostrará un error. Se usan agregando un **&** a la variable

```
fn foo(v: &Vec<i32>) {
    println!("{}", v[0]);
}

let v = vec![1,2];

foo(&v);
```

Referencias mutables

A diferencia de las compartidas, las referencias mutables permiten modificar el recurso, sin embargo sólo puede existir **una** referencia mutable que apunte a un objeto. Se usan agregando **&mut** a la variable.

```
fn foo(v: &mut Vec<i32>) {
    v.push(5);
}

let mut v = vec![];

foo(&mut v);
```

Reglas de borrowing

El sistema de borrowing tiene 2 reglas:

- Todo ‘préstamo’ debe tener un ‘scope’ no más grande que el del propietario.
- Puedes utilizar cualquiera de los 2 tipos de ‘préstamo’ **pero** no los 2 a la vez.

Reglas de borrowing

```rust
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

```rust
let mut x = 5;
let y = &mut x;

*y += 1;

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

Reglas de borrowing

La segunda regla ayuda a prevenir problemas como **Iterator Invalidation**

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

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

Reglas de borrowing

Otro problema que se previene es el **uso de memoria liberada**. Las referencias no deberían durar más que los recursos a los que apuntan.

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

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

4 Concurrencia

Send y Sync

Rust provee 2 traits que nos ayudan en la concurrencia:

- **Send:** Indica que el tipo que lo implementa es capaz de transferir ‘propiedad’ de forma segura entre threads

- **Sync:** Indica que el tipo que lo implementa no tiene la posibilidad de crear inseguridad de la memoria al ser usado en múltiples threads

Threads

La librería estándar de Rust provee la librería de threads, la cual nos permite correr código Rust en paralelo

```
use std::thread;

fn main() {
    thread::spawn(|| {
        println!("Hello from a thread!")
    });
}
```

Threads

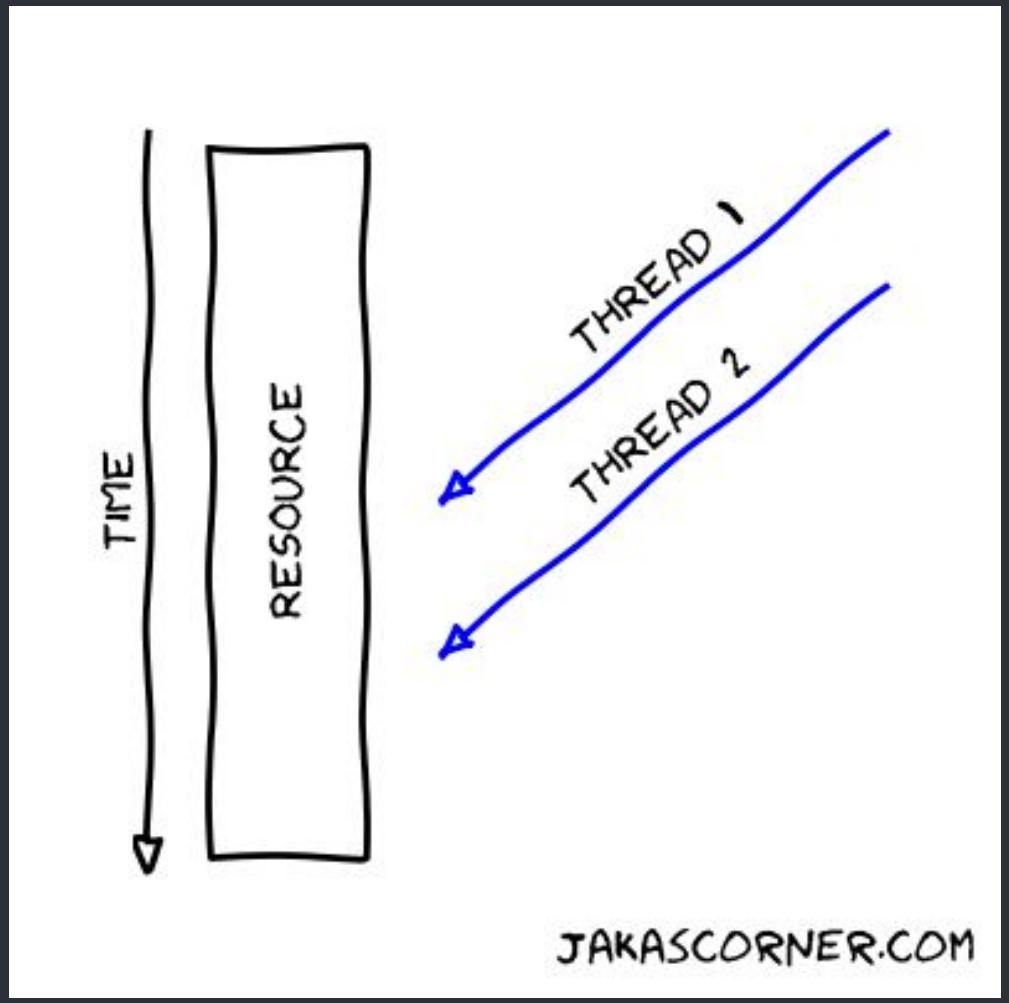
```
fn main() {
    let x = 1;
    thread::spawn(|| {
        println!("x is {}", x);
    });
}
```

```
fn main() {
    let x = 1;
    thread::spawn(move || {
        println!("x is {}", x);
    });
}
```

“

Shared mutable state is the root of all evil. Most languages attempt to deal with this problem through the 'mutable' part, but Rust deals with it by solving the 'shared' part.

Data Race



Data Race

```
use std::thread;

fn main() {
    let mut data = vec![1, 2, 3];

    for i in 0..3 {
        thread::spawn(move || {
            data[0] += i;
        });
    }
    thread::sleep(Duration::from_millis(50));
}
// error: capture of moved value ‘data’
```

Rc<T>

‘Reference Counted pointer’. Nos permite tener varios ‘apuntadores de propiedad’ a el mismo recurso y el recurso será destruido **solo** cuando todos los apuntadores sean destruidos.

Este apuntador no es ‘thread safe’, es decir, no podemos usarlo para transmitir mensajes entre varios threads.

Esta estructura requiere 2 ‘espacios’ (usize) más que Box<T> para los contadores de referencias. Además tiene un costo computacional extra cuando debe cambiar el contador

Cell<T>

Permite realizar mutabilidad interna sin costo moviendo los datos dentro y fuera de la ‘cell’.

```
let x = Cell::new(1);
let y = &x;
let z = &x;
x.set(2);
y.set(3);
z.set(4);
println!("{}", x.get());
```

```
let mut x = 1;
let y = &mut x;
let z = &mut x;
x = 2;
*y = 3;
*z = 4;
println!("{}", x);
```

RefCell<T>

Permite realizar mutabilidad interna obligando a seguir el patrón ‘read-write lock’ en runtime. De esta manera se asegura en tiempo de ejecución que no exista ningún otro ‘borrow’ activo mientras haya un ‘mutable borrow’.

```
let x = RefCell::new(vec![1,2,3,4]);
{
    println!("{:?}", *x.borrow())
}
{
    let mut my_ref = x.borrow_mut();
    my_ref.push(1);
}
```

Arc<T>

Arc<T> es una versión de Rc<T> que usa un ‘Atomic Reference Count’. El contador de referencias es únicamente modificado mediante operaciones atómicas, evitando ‘data races’

Similar a shared_ptr en C++, sin embargo el contenido de shared_ptr es mutable, el de Arc<T> no lo es.

Provee las mismas garantías que Rc pero con el costo añadido de la realización de las operaciones atómicas cuando el contador debe ser modificado

Arc<T>

```
let five = Arc::new(5);

for _ in 0..10 {
    let five = five.clone();

    thread::spawn(move || {
        println!("{:?}", five);
    });
}
```

Creamos un nuevo Arc con el dato **5**

Clonamos el Arc para acceder al recurso

Cada thread tiene acceso a una copia del recurso

Mutex<T>

Provee exclusión mutua mediante ‘RAII guards’, es decir, mantienen un estado hasta que su destructor sea llamado.

Cuando llamamos el método lock() en un Mutex<T> el thread al que pertenece se bloquea hasta que logre acceder al recurso bloqueado, retornando un ‘guard’.

El recurso es desbloqueado cuando el guard sale de scope.

Nos permite realizar mutabilidad segura entre threads.

Mutex<T>

```
fn push(mutex: &Mutex<Vector<i32>>) {

    let mut guard = mutex.lock();

    guard.push(5);

}
```

Adquiere el guard y bloquea el recurso

Muta los datos de manera segura

Levanta el bloqueo sobre los datos al salir de scope

Mutex<T>

```
fn main() {
    let data = Arc::new(Mutex::new(vec![1, 2, 3]));

    for i in 0..3 {
        let data = data.clone();
        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            data[0] += i;
        });
    }

    thread::sleep(Duration::from_millis(50));

}
```

Channels

Podemos crear canales usando mpsc::channel() para ‘sincronizar’ nuestros threads. Podemos enviar cualquier dato con el trait Send.

```
let (tx, rx) = mpsc::channel();

for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move || {
        let answer = i * i;
        tx.send(answer).unwrap();
    });
}

for _ in 0..10 {
    println!("{}", rx.recv().unwrap());
}
```

Panic!

La llamada al macro **panic!** detendrá la ejecución del thread actual y generará un error

```
let handle = thread::spawn(move || {
    println!("Antes de panic");
    panic!("oops!");
    println!("Después de panic");
});
```

```
Antes de panic
thread '<unnamed>' panicked at 'oops!', main.rs:7
```

Unsafe

El poder de Rust está en las distintas comprobaciones y restricciones en tiempo de compilación y ejecución que realiza, sin embargo en ocasiones es necesario saltarse estas reglas para la construcción de ciertos trozos de código (como Foreign Function Interfaces). Para esto usamos **unsafe**

```
unsafe fn danger_will_robinson() {
    // Scary stuff...
}
unsafe {
    // Scary stuff...
}
```

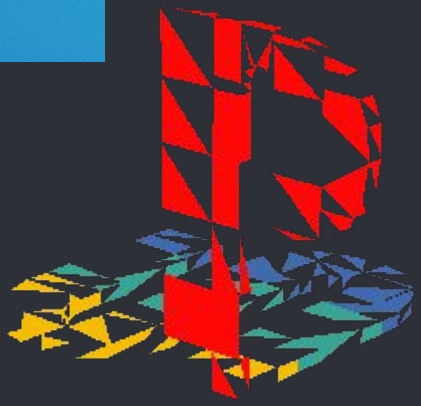
Rust en la actualidad

- Cargo, es el gestor de paquetes de Rust.

  Actualmente existen mas de 9000 ‘crates’ en **crates.io**

- Rust en la actualidad

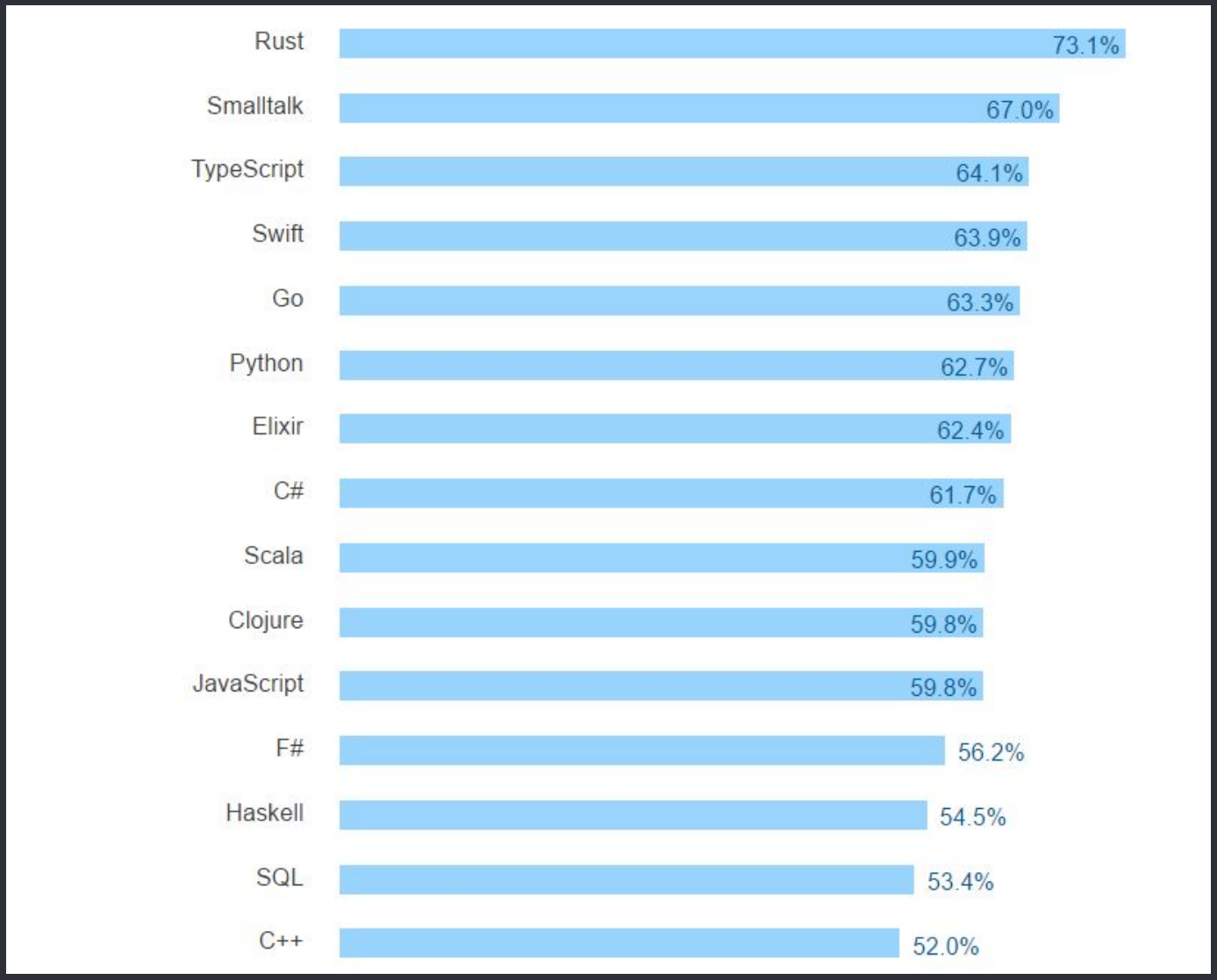
piston



Redox

MaidSafe

Rust en la actualidad



% de personas que usan el lenguaje y desean seguir trabajando con él

Bibliografía

- Why Rust? - Jim Blandy - O'Reilly
- https://doc.rust-lang.org/
- https://doc.rust-lang.org/stable/book/
- http://rustbyexample.com/
- http://henrikeichenhardt.blogspot.com.co/2013/06/why-shared-mutable-state-is-root-of-all.html
- https://www.youtube.com/watch?v=d1uraoHM8Gg&t=2934s
- https://www.youtube.com/watch?v=U1EFgCNLDB8&t=1475s
- https://github.com/kud1ing/awesome-rust
- https://insights.stackoverflow.com/survey/2017