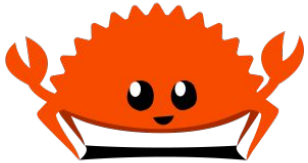
Introducción a Rust

NACION CRYPTO By DECRY

Qué es Rust?

Rust es un lenguaje de bajo nivel con una sintaxis moderna, Rust también es un lenguaje compilado al igual que C, C++, java, entre otros. Rust a su vez es un **lenguaje multiparadigma** tendiendo la posibilidad de programar con cualquiera de estos paradigmas. Programación funcional, por procedimientos, imperativa, concurrente y orientada a objetos.

RUST es una alternativa a lenguajes como C y C++ por sus características de rendimiento y alto nivel. No utiliza un recolector de basura como algunos de los lenguajes modernos, igualmente RUST garantiza la seguridad de la memoria y evita el uso incorrecto de la misma a diferencia de lo que se pudiera encontrar en C o C++.



Por qué Rust?

Al ser un lenguaje multipropósito, Rust presta mucha versatilidad a la hora de querer desarrollar, entre sus utilidades destacables están la creación de APIs, clientes HTTP, conectores a bases de datos y es usado para desarrollar aplicaciones con sistemas embebidos, que a su vez estos son usados para ejecutar tareas de control a un muy bajo costo de tamaño optimizando por ejemplo el uso de memoria RAM.

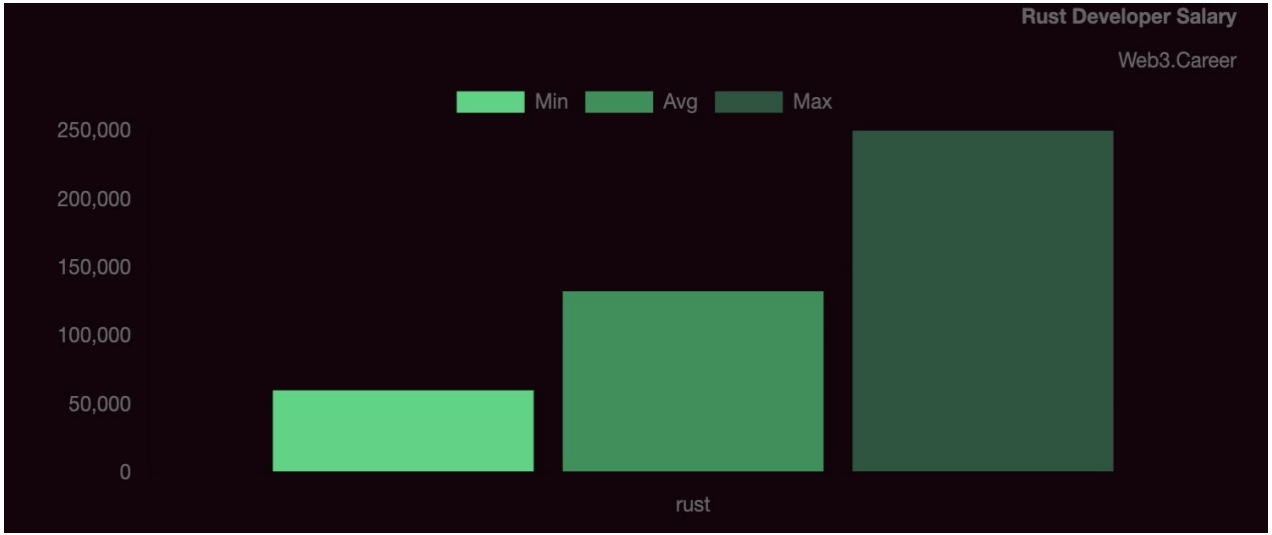
Otra característica interesante de Rust es que es robusto en el desarrollo web gracias al uso de **WebAssembly** el cual permite correr código escrito en diferentes lenguajes a una velocidad casi nativa

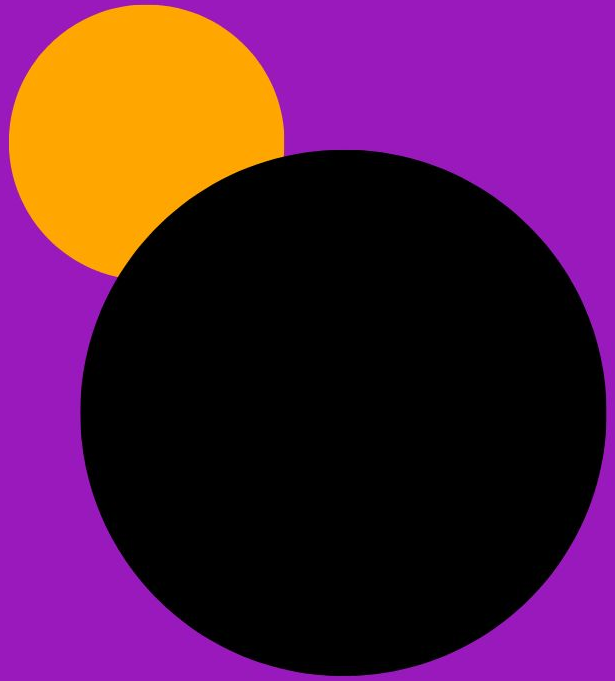
Quienes usan Rust?

Companies That Use Rust	Who Uses Rust at This Company?	What Does This Company Use Rust For?	Estimated Number of Employees
Amazon	Cloud engineers	Programming	798,000
Cloudflare	Computer programmers, Data scientists	Programming, data science backend	2,240
Coursera	Programmers, web developers	Programming, web development	779
Discord	Software engineers	Programming	1345
Dropbox	Software engineers, Cloud engineers	Programming, web development	2,760
Figma	Software engineers	Programming	350
Google	Software engineers, Android developers	Programming	135,301
Kraken	Software engineers	Programming	251
Microsoft	Software engineers	Programming	181,000
Mozilla	Software engineers	Programming	750

https://careerkarma.com/blog/who-uses-rust

Salarios!!


Rust Developer Salary
Web3.Career
Min
Avg
Max
250,000
200,000
150,000
100,000
50,000
0
rust



Instalación y complementos

Instalación

Rust

Si usamos un sistema macOS ó cualquier sistema basado en linux seguimos estos pasos:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

## Revisa si hay alguna actualización disponible
rustup check
## Actualiza a la última versión disponible
rustup update
```

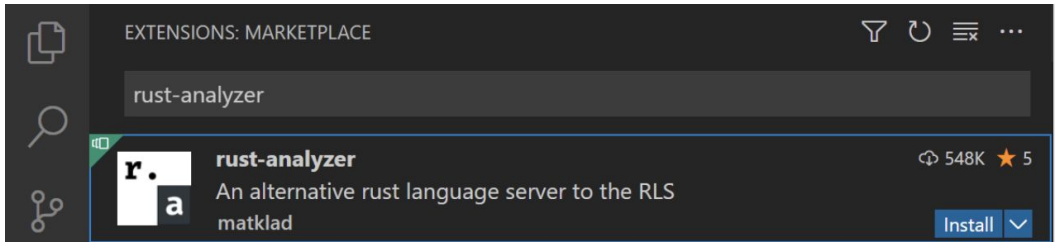
Instalación

VSCode

Instalar el ide Visual Studio Code:

```
https://code.visualstudio.com/Download
```

Para trabajar con Rust en VSCode, instalar el plugin “ rust-analyzer”



Manejador de paquetes: **CARGO**

Cargo es el manejador de paquetes de Rust, este tiene varias funciones muy interesantes que ayudan a que el desarrollo en rust sea más organizado y administrado a la hora de hacer programas grandes.

Algunas de las funciones que cargo permite hacer son:

- **crear un nuevo proyecto:** “cargo new <nombre_proyecto>”
- **cargo build:** el cual compila todos los archivos del proyecto
- **cargo run:** el cual ejecuta el programa, cabe resaltar que si la última versión del programa no está compilada con cargo build y se ejecuta con cargo run el proyecto igualmente se compilará, pero se recomienda primero compilar el proyecto y después ejecutarlo.

Manejador de paquetes: **CARGO**

Cuando se crea un nuevo proyecto con cargo, se genera un repositorio local con 3 ítems dentro:

- **src/:** Carpeta fuente que contiene un archivo main.rs
- **Cargo.toml:** Fichero de configuración donde se puede poner el nombre, la versión del proyecto y sus autores. También permite poner dependencias para usarlas en el código
- **.gitignore:** Archivo que le dice a git que archivos o carpetas debe ignorar dentro del proyecto

```
cargo new holamundo
cd holamundo
ls -a
src/ .gitignore Cargo.toml
```

Datos primitivos

Tipos primitivos:

Los tipos primitivos son los datos básicos que se pueden utilizar para definir variables y expresiones.

Los tipos primitivos son:

- **Enteros**
- **Booleanos**
- **Punto flotante (float)**
- **caracteres**

Enteros:

En Rust existen dos tipos de enteros, los enteros con signo (i) y los enteros sin signo (u), los enteros sin signo siempre serán positivos.

Así se define una variable con un valor entero en rust:

```
let x: i8 = -67;// variable definida con un entero con signo de 8 bits
println!("el número es {}",x);// muestra en pantall el número

let y: u16 = 1998;// Variable definida con un entero sin signo de 16 bits
println!("El número es {}",y);// muestra en pantalla el número
```

Lista de enteros:

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

Un numero entero con signo puede almacenar numeros desde (2n - 1) hasta 2n - 1 - 1, *n* es la cantidad de bit que el tipo usa.

Un numero entero sin signo puede almacenar 2n - 1, *n* es la cantidad de bit que el tipo usa.

Flotantes:

Un flotante es un número que contiene decimales, en Rust los flotantes siempre son flotantes con signo.

En Rust, hay dos tipos de flotantes “f32” y “f64”

- **f32**: flotante de 32 bits el cual representa una precisión simple.
- **f64:** flotante de 64 bits el cual representa una precisión doble.

```
let x: f32 = -67.8;// variable definida con un float de precisión simple
println!("el número es {}",x);// muestra en pantall el número

let y: f64 = 1998.41278589348;// variable definida con un float de precisión doble
println!("El número es {}",y);// muestra en pantalla el número
```

Caracteres:

Un carácter en Rust es un elemento único y contiene puntos de código unicode, y a diferencia de otros lenguajes donde char es tomado como un entero sin signo de 8 bits, en Rust es un único punto de código Unicode de 32 bits.

```
let letra: char = 'A';//Se asigna un char con un valor "A"
println!("el carácter es es {}",letra);// muestra en pantalla el caracter

let alien: char = '\u{1F47D}';// Se asigna un char unicode "cabeza de Alienpigena"
println!("el carácter es es {}",alien);// muestra en pantalla el caracter
```

Conversión entre Tipos de Datos

Rust no proporciona ninguna conversión de tipos implícita (coerción) entre tipos primitivos. Sin embargo, la conversión explícita de tipos (casting) se puede realizar utilizando la palabra clave as.

```
fn main() {
 let decimal = 65.4321_f32; // Explicit conversion
 let integer = decimal as u8;

}
```

Conversión entre Tipos de Datos

Las reglas de conversión se ejecutan en general de acuerdo a las convenciones de lenguaje C, de esta manera en una expresión en la intervienen datos de distintos tipos se debe realiza una conversión de todos ellos a la variable con el mayor tipo de datos.

```
fn main() {
 let integer = 20u8;
 let otro_integer = 10u16;

 let result = integer as u16 + otro_integer;

println!("{}",result)

}
```

Operadores y Expresiones

En Rust, los operadores son símbolos que realizan operaciones sobre los datos. Las expresiones combinan valores y operadores para producir un resultado.

Tipos de operadores:

- **Operadores Matemáticos.**
- **Operadores de comparación.**
- **Operadores lógicos.**
- **Operadores de asignación.**

Operadores Matemáticos:

Cómo su nombre lo dice los operadores matemáticos servirán para realizar las operaciones matemáticas que se necesiten, Rust cuenta con los siguientes operadores matemáticos.(**+, -, *, /, %**)

- **‘+’:** Suma dos valores.
- **‘-’:** Resta dos valores.
- **‘*’:** Multiplica dos valores.
- **‘/’:** Divide dos valores.
- **‘%’:** Obtiene el módulo.

Operadores Matemáticos:

En Rust, las expresiones matemáticas combinan los operadores matemáticos con los valores, esto con el fin de obtener un resultado a partir de dos valores.

```
let a = 10; // asignación de una variable inmutable = 5
let b = 5; // asignación de una variable inmutable = 5
let c = a + b; // suna
let d = c - a; // resta
let e = c * d; // multiplicación
let f = e / b; // división
let g = a % b; // módulo
```

```
let x: f32 = 10.34; // asignación de variable inmutable tipo float
let y: f32 = 7.43; // asignación de variable inmutable tipo float
let z: f32 = x + y; // expresión matemática que suma dos float
```

Operadores de comparación:

Los operadores comparativos sirven para comparar dos valores y como resultado de estas comparaciones se obtiene un valor booleano (true o false), estos operadores comparativos son muy usados en estructuras de tipo control de flujo, ya que permiten que el programa ejecute una acción dependiendo de un resultado determinado.

- ‘==’: Compara si dos valores son iguales.
- ‘!=’: Compara si dos valores son diferentes.
- ‘<:’ Compara si un valor es menor que otro.
- ‘>’: Compara si un valor es mayor que otro.
- ‘<=’: Compara si un valor es menor o igual que otro.
- ‘>=’: Compara si un valor es mayor o igual que otro

Operadores de comparación:

En Rust, las expresiones de comparación combinan los operadores matemáticos con los valores y estas devuelven un valor de tipo booleano.

```rust
let a: u8 = 3;
let b: u8 = 6;
let c: u8 = 6;


let w: bool = a == b; //false
let x: bool = a != c; //true
let y: bool = b < c; //false
let z: bool = a <= b; //true
let v: bool = a > b; //false
```

Operadores Lógicos:

Los operadores lógicos se usan para efectuar operaciones entre expresiones, esto da como resultado un valor booleano.

Los operadores lógicos en Rust son:

- **‘&&’:** (AND) si ambos son verdaderos el valor es **true**, en caso contrario es **false.**
- **‘||’:** (OR) Si alguna de las expresiones es verdadera el valor es **true**, en caso contrario es **false.**
- **‘!’:** (NOT) este operador devuelve el resultado contrario, es decir si el resultado es **false,** devuelve **true** y si es **true,** devuelve **false.**

Expresiones lógicas:

Las expresiones lógicas combinan dos resultados de expresiones de comparación, dando como resultado un true o un false, ejemplo.

```
let a: u8 = 3;
let b: u8 = 6;
let c: u8 = 6;


let w: bool = a != c || a == b; //true
let y: bool = !(b < c); //true
let z: bool = a > b && a <= b; //false
```

Operadores de asignación:

Los operadores de asignación se usan para asignar un valor a una variable, con estos se puede reducir el uso de expresiones matemáticas.

- ‘=’: Asigna un valor a una variable.
- ‘+=’: Suma un valor a una variable ya definida.
- ‘-=’: Resta un valor a una variable ya definida.
- ‘*=’: Multiplica un valor a una variable ya definida.
- ‘/=’: Divide una variable ya definida por un valor.

Operadores de asignación:

Las expresiones de asignación se representan en Rust así:

```
// Esto se hace con fines prácitos, este ejemplo Rust lo detecta como warning
// ya que la variable está siendo sobreescrita varias veces sin cumplir con algun uso
// Para evitar el warning se debe poner "_" antes del nombre de la variable "_a"
// esto le dice a Rust que es intencinal lo que se está haciéndo y lo compile
let mut a = 10; // definimos una variable mutable = 10
a = 20; // se modifica el valor de la variable ahora es = 20
a += 5;// se suma 5 al anterior valor 20 + 5 = 25
a -= 10;// se resta 10 al valor anterior 25 - 10 = 15
a *= 2;// se multiplica  2 al valor anterior 15 * 2 = 30
a /= 2;// se divide por 2 al valor anterior 30 / 2 = 15
a %= 4;// se saca el módulo al valor anterior 30 % 4 = 3
```

LECTURAS RECOMENDADAS:

- https://polkadothub.io/rust/0-presentaci%C3%B3n/0-presentation
- https://doc.rust-lang.org/book/