Tipos de datos avanzados

Víctor Fabián Serna Villa
vfabian@decry.io

NACION CRYPTO By DECRY

Arreglos (Array)

El Tipo [T; N] ⟶ Un arreglo de N valores del tipo T.

nombre_arreglo.len() ⟶ Devuelve la cantidad de elementos

Nombre_arreglo [i] ⟶ Se refiere al elemento en la posición **i** del vector.

Primer elemento **v [0]** y el último **v[v.lent ( ) - 1]**

```
// Un arreglo de Número naturales
let arr = [1, 2, 3, 4, 5];

let arr : [u32;5] = [1, 2, 3, 4, 5];

let vnombre = [“animal”, “oso”];
```

Es una lista de elementos del mismo tipo, por defecto es inmutable y se almacena en el stack.

Arreglos (Array)

Inicializar un arreglo con valores por defecto

let numbers: [T;N] = [V; L]; ⟶ Donde V es el valor y L la longitud

```
fn main() {
  // Inicializa el valor con el número 3
    let numbers: [i32; 5] = [3; 5];

    println!("Array of numbers = {:?}", numbers);
}
```

Vectores

Vec < T > ⟶ arreglo de elementos del tipo T

Nombre_vector[i] ⟶ Se refiere al elemento en la posición **i** del vector.

let mut v: vec<i32> = vec::new(); ⟶ crear un vector vacío

Es una estructura de datos del mismo tipo dinámica, alojada en el heap.

```
//inicializando un vector
let mut primos = vec![2, 3, 5, 7];

Primos.push (11);
Primos.push (12);
```

Vectores (métodos)

vector.get (index) → Accede al elemento del vector en la posición index
vector.push (valor) → Adiciona el elemento valor al final del vector.
vector.pop ( ) → remueve el último elemento del vector.
vector.remove (index) → remueve el elemento de la posición index.

Tuplas

```
// Tupla de longitud 3

let tupla = ('E', 5i32, true);
```

Elemento	Value	Tipo de datos
0	E	`char`
1	5	`i32`
2	true	`bool`

Es una agrupación de diferentes tipos de valores en una estructura de tamaño fija.

Tuplas
Index

```
// declaramamos 3 elementos en esta tupla
let tuple_e = ('E', 5i32, true);

// usamos el index de la tupla para acceder a los valores
println!("Is '{}' la {}th letra del alfabeto? {}", tuple_e.0, tuple_e.1, tuple_e.2);
```

Tuplas

Destructurando

```
let tuple = ("Juan Jose", 18, 175);
let (nombre, edad, altura) = tuple;
```

Acceder a los datos así:

- Nombre en lugar de tuple.0
- edad en lugar de tuple.1
- altura en lugar de tuple.2

Podemos descomponer una tupla dentro de variables.

Slice

&[T] ⟶ poseen el tipo T Generico

```
let numbers = [0, 1, 2, 3, 4];
let middle = &numbers[1..4]; // Un slice de number: solo los elementos 1, 2, y 3
let complete = &numbers[..]; // Un slice conteniendo todos los elementos de a.
```

Es una referencia ó región de otra estructura de datos, como puede ser un arreglo, vector ó cadena.

Estructura

1. Definir la estructura con nombre y definir el tipo de dato de sus campos.
2. Se crea la instancia de la estructura con otro nombre.

```
struct struct_name {
   field1: data_type,
   field2: data_type,
   field3: data_type
}
```

Una estructura es un tipo de datos compuesto por diferentes tipos de datos pero a diferencia de las tuplas, cada tipo de dato tiene un identificador.

Estructura

Tipos

1. **Clásicas ó tipo nombre:** cada campo de la estructura tiene nombre y un tipo de dato. Una vez definida se accede sus campos así <struct>.<field>
2. **Tipo tupla:** sus campos no tienen nombres. A fin de acceder a los campos de una estructura de tupla, usamos la misma sintaxis que para indexar una tupla: `<tuple>.<index>`.
3. **Tipo de unidad:** suelen usarse como marcadores. No tienen elementos, puede ser muy útil cuando trabajamos con traits.

Estructura

Tipos: creación

```
// Estructura con nombre
struct Student { name: String, level: u8, remote: bool }

// Estructura tipo tupla
struct Grades(char, char, char, char, f32);

// Estructura tipo unidad
struct Unit;
```

Estructura

```
// Instantiate classic struct, specify fields in random order, or in specified order
let user_1 = Student { name: String::from("Constance Sharma"), remote: true, level: 2 };
let user_2 = Student { name: String::from("Dyson Tan"), level: 5, remote: false };

// Instantiate tuple structs, pass values in same order as types defined
let mark_1 = Grades('A', 'A', 'B', 'A', 3.75);
let mark_2 = Grades('B', 'A', 'A', 'C', 3.25);

println!("{}, level {}. Remote: {}. Grades: {}, {}, {}, {}. Average: {}",
      user_1.name, user_1.level, user_1.remote, mark_1.0, mark_1.1, mark_1.2, mark_1.3, mark_1.4);
println!("{}, level {}. Remote: {}. Grades: {}, {}, {}, {}. Average: {}",
      user_2.name, user_2.level, user_2.remote, mark_2.0, mark_2.1, mark_2.2, mark_2.3, mark_2.4);
```

* conversión de referencia a string: String::from(&str)

Enum

Se define con la palabra **enum**

Puede tener cualquier combinación de las variantes de enumeración.

Puede tener campos con nombres ó sin nombre.

```
enum IpAddrKind {
    V4,
    V6,
}
struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};

let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Tipo de datos que puede ser una de un conjunto de variantes. Esto se conoce en computación como datos algebraicos.

Enum

Definición

```
enum WebEvent {
   // estructura sin datos
   WELoad,
   // puede ser una estructura tipo tupla
   WEKeys(String, char),
   // puede ser una estructura con nombre, clasica.
   WEClick { x: i64, y: i64 }
}
```

```
//Definición usando estructuras
// Define a tuple struct
struct KeyPress(String, char);

// Define a classic struct
struct MouseClick { x: i64, y: i64 }

// Redefine the enum variants to use the data from the new
structs
// Update the page Load variant to have the boolean type
enum WebEvent { WELoad(bool), WEClick(MouseClick),
WEKeys(KeyPress) }
```

Enum

Instancia

Usamos la palabra clave **let**
Acceder a la variante especifica `<enum>::<variant>`

Ejemplo

```
let we_load = WebEvent::WELoad(true);
```

```
// Define a tuple struct
struct KeyPress(String, char);
// Instantiate a KeyPress tuple and bind the key values
let keys = KeyPress(String::from("Ctrl+"), 'N');
// Set the WEKeys variant to use the data in the keys tuple
let we_key = WebEvent::WEKeys(keys);
```

a. HashMap
b. TALLER
c. Traits
    i. Traits bound
    ii. Traits definition
d. Tipos de datos genéricos

HashMap

```
use std::collections::HashMap;
let mut nombre: HashMap<i32, String> = HashMap::new();
```

- `let mut nombre` - declara una variable mutable llamada nombre.
- `HashMap<i32, String>` - tipo de HashMap donde la clave es un entero y el valor un string.
- `HashMap::new()` - este método crea un nuevo HashMap.

Es una estructura de datos que permite almacenar valores con su clave.
Se pueden redimensionar.
Las claves son únicas.

HashMap

Operaciones

Adicionar ó agregar elementos

insert(<key>, <value>) → Se utiliza este método
Sintaxis:
<hash_map_name>.insert()

```
let mut fruits: HashMap<i32, String> = HashMap::new();
// insert elements to hashmap
fruits.insert(1, String::from("Apple"));
fruits.insert(2, String::from("Banana"));
```

HashMap

Operaciones

Acceder a un valor

get(<key>) → Se utiliza este método
Sintaxis:
<hash_map_name>.get(key)

```
let mut fruits: HashMap<i32, String> = HashMap::new();

fruits.insert(1, String::from("Apple"));
fruits.insert(2, String::from("Banana"));

let first_fruit = fruits.get(&1);
```

El método get devuelve un tipo Option<&Value>. Rust encapsula el resultado de la llamada de método con la notación "Some()".

HashMap

Operaciones

Remover ó eliminar un valor.

remove(<key>) → Se utiliza este método
Sintaxis:
<hash_map_name>.remove(key)

```
let mut fruits: HashMap<i32, String> = HashMap::new();

fruits.insert(1, String::from("Apple"));
fruits.insert(2, String::from("Banana"));

fruits.remove(&1);
```

Si usamos el método get para una clave de mapa hash no válida, el método get devuelve "None".

HashMap

Operaciones

Cambiando/actualizando elementos

insert(<key>) → Se utiliza este método
Sintaxis:
<hash_map_name>.insert(key)

```
let mut fruits: HashMap<i32, String> = HashMap::new();

// insert values in the hashmap
fruits.insert(1, String::from("Apple"));
fruits.insert(2, String::from("Banana"));

// update the value of the element with key 1
fruits.insert(1, String::from("Mango"));
```

HashMap

Operaciones
Otros métodos

- len( ) → retorna la longitud del HashMap
- contains_key( ) → verifica si el valor existe para esa clave.
- Values ( ) → retorna un iterador sobre los valores del HashMap
- Keys ( ) → retorna un iterador sobre las claves del HashMap
- Clone ( ) → crea y retorna una copia del HashMap

Genericos

Es un tipo de datos que se define en términos de otros:

- HashMap<K, V>
- Vec<T>
- enumeración Option<T>

Permite escribir funciones, estructuras y métodos lo cuales pueden trabajar con diferentes tipos de datos sin conocerlos previamente, estos en lugar de especificar el tipo de dato, utiliza un parámetro de tipo que se define en tiempo de compilación. Los tipos genéricos son usados para crear código reutilizable que puede ser aplicado a diferentes tipos de datos, reduciendo así la cantidad de código duplicado.

Genericos

Es un tipo de datos que se define en términos de otros:

- HashMap<K, V>
- Vec<T>
- enumeración Option<T>

Permite escribir funciones, estructuras y métodos lo cuales pueden trabajar con diferentes tipos de datos sin conocerlos previamente, estos en lugar de especificar el tipo de dato, utiliza un parámetro de tipo que se define en tiempo de compilación. Los tipos genéricos son usados para crear código reutilizable que puede ser aplicado a diferentes tipos de datos, reduciendo así la cantidad de código duplicado.

Genericos

Anotación

Se utiliza un solo caracter para indicar el genérico en la definición.

- T, U son usados para representar cualquier tipo
- K, V son usados para valores tipo clave
- E es usado para representar un error.

Genericos

Estructura con tipos genericos

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let no_work = Point { x: 1.0, y: true };
}
```

Genericos

Funciones con parámetros genéricos.

```
// Funcion con un tipo de dato genérico
fn my_function<T>(x: T, y: T) -> T {
   // function body
   // do something with `x` and `y`
}

// Función con dos tipos de datos genéricos
fn my_function<T, U>(x: T, y: U) {
   // function body
   // do something with `x` and `y`
}
```

.

```
struct Point<T> {
   x: T,
   y: T,
}

fn main() {
   let integer = Point { x: 5, y: 10 };
   let no_work = Point { x: 1.0, y: true };
}
```

Traits

- Implementar std::io::Write puede escribr bytes
- Implementar std::iter::Iterator can producir una secuencia de valores
- Implementar std::clone::Clone can hacer copias de algo en la memoria.
- Implementar std::fmt::Debud permite imprimir usando println!( ) con el formato {:?}

.

Los traits son una colección de métodos para cualquier tipo de datos en Rust. Son como las interface en Java ó C#,
La palabra **self** puede ser usada para a otros métodos declarados en el mismo traits

Traits

impl Trait for Type, donde Trait es el nombre del rasgo que se implementa y Type es el nombre de la estructura del implementador o la enumeración.

.

```
trait TraitName {
    fn method_one(&self, [arguments: argument_type]) -> return_type;
    fn method_two(&mut self, [arguments: argument_type]) -> return_type;
    ...
}
```

Traits

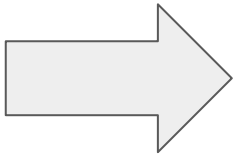
```
trait MyTrait {
   fn method_one(&self);
   fn method_two(&mut self, arg: i32) -> bool;
}
```

Implementación:

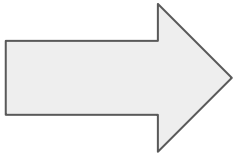
```
impl TraitName for TypeName {
   fn method_one(&self, [arguments: argument_type]) -> return_type {
      // implementation for method_one
   }

   fn method_two(&mut self, [arguments: argument_type]) -> return_type {
      // implementation for method_two
   }

   ...
}
```



Definición



Implementación

Traits

```
struct Circle {
    radius: f64,
}

struct Rectangle {
    width: f64,
    height: f64,
}

impl Area for Circle {
    fn area(&self) -> f64 {
        use std::f64::consts::PI;
        PI * self.radius.powf(2.0)
    }
}

impl Area for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
}
```

```
// accedemos a los métodos definidos después de
inicializar las estructuras
let circle = Circle { radius: 5.0 };
let rectangle = Rectangle {
    width: 10.0,
    height: 20.0,
};

println!("Circle area: {}", circle.area());
println!("Rectangle area: {}", rectangle.area());
```

Traits

Traits en funciones genericas:

```
trait nombre_trait {
fn fn_name (&self) -> String;
}
```

```
fn fn_implementa_trait (value: &impl nombre_trait) {
println!("hola...");
}
```

Acepta todos los tipos de datos que implementen el trait. Puede utilizar los métodos definidos en el trait.

Se puede escribir con la siguiente sintaxis:

```
fn fn_implementa_trait <T: nombre_trait>(value: &T) {
...
}
```

Traits

Ejemplo

```
// Define a trait Printable
trait Printable {
    fn print(&self);
}

// Define a struct to implement a trait
struct Person {
    name: String,
    age: u32,
}

// Implement trait Printable on struct Person
impl Printable for Person {
    fn print(&self) {
        println!("Person {{ name: {}, age: {} }}",
self.name, self.age);
    }
}

// Define another struct to implement a trait
struct Car {
    make: String,
    model: String,
}
```

```
// Define trait Printable on struct Car
impl Printable for Car {
    fn print(&self) {
        println!("Car {{ make: {}, model: {} }}",
self.make, self.model);
    }
}

// Utility function to print any object that implements
the Printable trait
fn print_thing<T: Printable>(thing: &T) {
    thing.print();
}

fn main() {
    // Instantiate Person and Car
    let person = Person { name: "Hari".to_string(), age:
31 };
    let car = Car { make: "Tesla".to_string(), model:
"Model X".to_string() };

    // Call print_thing with reference of Person and Car
    print_thing(&person);
    print_thing(&car);
}
```

Traits

Palabra clave derive

Uso

```
#[derive(Trait)]
```

Es usado para generar la implementación de ciertos tipos de traits en tipos de datos como struct y Enum.

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p1 = Point { x: 1, y: 2 };
    let p2 = Point { x: 4, y: -3 };

    if p1 == p2 {
        println!("equal!");
    } else {
        println!("not equal!");
    }

    println!("{:?}", p1);
   println!("{:?}", p1);
}
```

BIBLIOGRAFIA:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
-