Introduction to Rust

Nếu C/C++ là ngôn ngữ lập trình cho tốc độ thực thi nhanh, quản lý bộ nhớ một cách tuyệt vời. Còn Python, Ruby thì nổi tiếng với sự đơn giản, dễ sử dụng, thời gian viết code nhanh. Thì Rust sinh ra để trung hòa hai thứ trên. Rust cho chúng ta tốc độ thực thi chương trình nhanh, độ trễ thấp và sự kiểm soát chặt chẽ về bộ nhớ (không bị crash, leak memory,...).

Rust là một ngôn ngữ lập trình được phát triển bởi Mozilla Research và sau đó chủ yếu được phát triển bởi cộng đồng mã nguồn mở. Cha đẻ của Rust là một nhà thiết kế ngôn ngữ Graydon Hoare, Rust được giới thiệu lần đầu vào năm 2010. Ngoài ra Rust còn có các chức năng hỗ trợ chạy nhiều tiến trình song song trên các máy tính đa lõi.

Song song với Rust là Cargo. Tương tự với pip của Python thì Cargo là phần mềm cho phép chúng ta quản lý thư viện, biên dịch, chạy chương trình, debug, tạo project, chạy unit test,..

Tiếp theo sau đây chúng ta sẽ tìm hiểu những thứ cơ bản về Rust.

Mục lục:

Variable

Concept

Trong Rust, một biến được mặc định là bất biến (immutable). Tức là nó hoàn toàn không thay đổi trong suốt thời gian chương trình được thực hiện. Và chúng ta hoàn toàn có thể sử dụng biến có thể thay đổi giá trị được

(mutable)

Declair and use variable

Syntax:

- Với immutable variable:

```
let <variable_name> = <value>;
let <variable_name>: <data_type>  = <value>;
```

- Với mutable variable ta có cách thứ hai để khai báo biến:

```
let mut <variable_name> = <value>;
let mut <variable_name>: <data_type>;
let mut <variable_name>: <data_type>  = <value>;
```

Ví dụ

```
1 fn main()
2 {
3     let x: i32 = 12;
4     let mut y = 1.2;
5     let mut z = true;
6     let mut t: i128 = 1_000_000_000; // t = 1000000000
7 }
```

Rust có hỗ trợ chúng ta một cách viết số lớn một các dễ dàng với ký tự phân cách là '_' mà không làm ảnh hưởng đến giá trị của biến.

Use variable

Rust có một cách tổ chức biến khác với ngôn ngữ lập trình thường dùng như C/C++, Python,... Đó là Rust có chức năng rất độc đáo là Ownership (em sẽ nói kỹ hơn chức năng này khi đến phần hàm).

Constant

Syntax:

```
const <variable_name>: <data_type> = <value>;
```

Ví dụ:

```
const PI: f32 = 3.14;
```

Ta thấy, giữa hai cách khai báo hằng và biến (`const` và `let`) có vẻ giống nhau vì cả hai mặc định đều không thể thay đổi (immutable). Nhưng thật ra nó có một vài điểm khác nhau đặc biệt lưu ý như sau:

1. Tuyệt đối không được sử dụng từ khóa `mut` khi khai báo hằng.
2. Kiểu dữ liệu của `const` bắt buộc phải có khi khai báo, còn `let` thì không
3. Giá trị khi gán vào `const` không được là giá trị trả về của một hàm, mà bắt buộc phải là một giá trị cụ thể, cố định (hoặc `INFINITY`, `NEG_INFINITY`). Còn giá trị của `let` có thể là giá trị trả về của hàm.

(tham khảo: https://nickymeuleman.netlify.app/garden/rust-let-const)

Datatypes

Scalar types

Scalar types là kiểu dữ liệu chỉ chứa độc nhất một giá trị.

Integer

Số nguyên có hai loại:

- Có dấu (kiểu bù 2)
- Không dấu

Các loại số nguyên có dấu và không dấu sử dụng từ 8-128 bit được trình bày trong bảng sau:

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Bảng 1. Kiểu số nguyên trong Rust

Ngoài ra, Rust còn hỗ trợ nhiều loại hệ cơ số khác nhau được trình bày trong bảng sau:

Number literals	Example
Decimal	`1_350_000`
Hex	`0xf87ac2`

Number literals	Example
Octal	`0o137`
Binary	`0b1011_1001`
Byte (`u8` only)	`b'A'`

Bảng 2. Hệ cơ số trong Rust

(tham khảo: https://doc.rust-lang.org/book/ch03-02-data-types.html#scalar-types)

Floating-point

Rust có hai kiểu số thực là `f32` và `f64` lần lượt sử dụng 32 bit và 64 bit (về độ lớn). Mặc định khi khai báo máy sẽ sử dụng `f64`.

Ví dụ:

```
fn main()
{
    let x = 3.2;           // x có kiểu dữ liệu là f64
    let y: f32 = 1.2345;   // y có kiểu dữ liệu là f32
}
```

Boolean

Kiểu luận lý của Rust cũng tương tự như các ngôn ngữ khác, tồn tại một trong hai trạng thái `true` hoặc `false`.

Ví dụ:

```
fn main()
{
    let x = true;
    let y: bool = false;
}
```

Character

Kiểu ký tự `char` của Rust sử dụng dãy ký tự alphabet và cho phép sử dụng emoji.

Ví dụ:

```
fn main()
{
    let x = 'x';
```

```
    let y: char = '😘';
}
```

Compound types

Compound types là loại kiểu dữ liệu có thể tích hợp nhiều kiểu dữ liệu khác nhau. Compound types có hai loại: Tuple và Array.

Tuple

`Tuple` là một loại dữ liệu phức hợp thường được dùng để nhóm nhiều kiểu dữ liệu khác lại với nhau. Kích thước của `Tuple` là bất biến, khi khai báo sẽ không thể thay đổi.

Để truy cập tới phần tử của `tuple` ta dùng toán tử "`.`". Với phần tử đầu tiên bắt đầu từ 0 và cứ thế đến hết.

Ví dụ:

```
fn main()
{
    let x = ("Con gà", 1.4, 145); // con gà nặng 1.4kg, giá 145k
    println!("{} nặng {}kg, giá {}k", x.0, x.1, x.2);
}
```

```
#RESULT
Con gà nặng 1.4kg, giá 145k
```

Array

`Array` (mảng) là một tập hợp phần tử có cùng kiểu. Độ dài của mảng sẽ không thể thay đổi trong khi chạy chương trình.

Cú pháp khai báo:

```
1 let <var_name> = [<value_1>, <value_2>,...,<value_n>];
2 let <var_name> = [<value>; <num_of_value>];
3 let <var_name>:[<data_type>; <num_of_value>];
```

Với cách khai báo ở dòng thứ 2, kết quả sau khi thực hiện câu lệnh sẽ là tạo một `array` với `num_of_value` giá trị `value`. Cách khai báo này sẽ rất hữu dụng khi ta cần khởi tạo toàn bộ giá trị của `array` bằng một giá trị nào đó.

Ví dụ:

```
fn main()
{
    let arr1 = [1,2,3,4];
    let arr2:[u32;15];
    let arr3 = [2;3];
}
```

Để truy cập tới phần tử của `array` ta dùng toán tử "`.`" tương tự như `tuple`.

Control-flow

If-else

`If-else` là câu lệnh điều kiện trong Rust với cú pháp cũng tương tự như những ngôn ngữ lập trình khác.

Cú pháp:

```
if <condition> {
    // code
}

if <condition> {
    // code 1
}
else {
    // code 2
}
```

Và ta cũng hoàn toàn có thể sử dụng `if-else` lồng nhau.

Ví dụ:

```
fn main()
{
    let my_age = 19;
    if my_age < 18 {
        println!("Chưa đủ tuổi thi bằng lái!");
    }
    else {
        println!("Đủ tuổi thi bằng lái!");
    }
}
```

Kết quả của chương trình trên:

```
Đủ tuổi thi bằng lái!
```

Loop statement

Câu lệnh lặp trong Rust có ba loại: `for`, `loop`, `while`.

For - in

Ví dụ với `for - in`:

```
fn main()
{
    for i in 1..10
    {
        print!("{} ", i);
    }
}
```

Kết quả của câu lệnh trên:

```
1 2 3 4 5 6 7 8 9
```

`1..10` Có nghĩa là lặp với `i` thuộc nửa đoạn `[1, 10)`

Với câu lệnh `for - in` ta sử dụng khi biết trước số lần lặp. Khi ta chỉ cần lặp với một số lần cố định mà không quan tâm biến lặp, ta có thể thay biến đếm (ở ví dụ trên là `i`) thành `_`
Câu lệnh trở thành

```
for _ in 1..10 {
}
```

While

Ví dụ với `while`

```
fn main()
{
    let mut i = 1;
    while i < 10
    {
        print!("{} ", i);
        i = i + 1;
```

```
    }
}
```

Kết quả của câu lệnh trên:

```
1 2 3 4 5 6 7 8 9
```

Khi điều kiện ngay sau while thì sẽ lập tức dừng vòng lặp, chương trình sẽ tiếp tục thực hiện những câu lệnh sau.

Loop

`loop` trong Rust là một từ khóa có ý nghĩa như thực hiện một vòng lặp vô hạn trong dấu `{}` sau từ khóa `loop`. Nếu muốn dừng vòng lặp vô hạn ta phải dùng câu lệnh `break`.

Ví dụ:

```
fn main()
{
    let mut i = 1;
    loop
    {
        print!("{} ", i)
        if i < 10
        {
            i = i + 1;
            continue;
        }
        break;
    }
}
```

Kết quả của câu lệnh trên:

```
1 2 3 4 5 6 7 8 9
```

Function

Cú pháp viết hàm

```
fn <func_name>(<some_parameter>) {
    // some code
}
```

Ví dụ:

```
fn main()
{
    hello_world();
}

fn hello_world()
{
    println!("Hello, worlds");
}
```

Để có thể phát huy hết sức mạnh của hàm trong Rust, ta cần phải tìm hiểu một số khái niệm quan trọng của Rust.

Ownership & borrowing

`Ownership & borrowing` là một cơ chế trong Rust, nó có thể được phát biểu như sau:

> 1. Mỗi giá trị chỉ có duy nhất một chủ sở hữu (`owner`).
> 2. Tại một thời điểm, một giá trị chỉ có duy nhất một `owner`(tức là không tồn tại chuyện hai `owner` của một giá trị tại một thời điểm, một rừng không thể có hai hổ).
> 3. Khi `owner` ra khỏi vùng hoạt động, giá trị sẽ bị drop.

Chẳng hạn khi ta chạy chương trình sau:

```
fn main()
{
    let x = String::from("Hello");
    test_ownership(x);
    println!("x = {}", x);
}

fn test_ownership(x: String)
{
    println!("{}", x);
}
```

Chương trình sẽ báo lỗi như sau:

```
Jun@DESKTOP-S96CU36 MINGW64 ~/Desktop/Rust/program (master)
$ cargo run
   Compiling program v0.1.0 (C:\Users\Jun\Desktop\Rust\program)
error[E0382]: borrow of moved value: `x`
 --> src\main.rs:6:24
  |
4 |     let x = String::from("Hello");
```

```
  |         - move occurs because `x` has type `String`, which does not implement
the `Copy` trait
5 |     test_ownership(x);
  |                    - value moved here
6 |     println!("x = {}", x);
  |                        ^ value borrowed here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `program` due to previous error
```

"Value borrowed here after move" Báo hiệu rằng giá trị trong x đã bị drop khi truyền vào hàm `test_ownership` và `x` bây giờ không chứa bất kì giá trị nào.

Để khắc phục vấn đề trên, ta có thể làm như sau:

```
fn main()
{
    let x = String::from("Hello");
    let y = test_ownership(x);
    println!("y = {}", y);
}

fn test_ownership(x: String) -> String
{
    println!("{}", x);
    return x
}
```

Kết quả khi thực hiện chương trình trên là:

```
Hello
y = Hello
```

Ở hàm test_ownership có sử dụng ký tự `-> String` là để biểu thị giá trị trả về của hàm là `String`. Ngoài ra câu lệnh `return x` có thể đơn giản hóa bằng tên biến ta muốn trả về chẳng hạng như `x`.

Recap

> Viết hàm có giá trị trả về theo cú pháp `fn <func_name>(<some_parameter>) -> <data_type> {}`
> Khi khai báo hàm, nếu có truyền tham số, nhất định phải khai báo kiểu dữ liệu.
> Chức năng `ownership` và ứng dụng nó với hàm.
> Statement không trả về giá trị. Chỉ có expression mới làm được việc đó.

Reference

Khi việc sử dụng chức năng `ownership` khá là bất tiện khi phải lấy rồi trả các biến, ta có thử sử dụng `reference` để truyền thẳng `owner` vào.

Immutable reference

Để tham chiếu một biến đến một hàm bất kì ta có thể sử dụng toán tử `&`.

Ví dụ:

```
fn main()
{
    let x = String::from("Xin chào");
    test_reference(&x);
    println!("{}", x);
}

fn test_reference(x: &String)
{
    println!("{}", x);
}
```

Kết quả của chương trình trên:

```
Xin chào
Xin chào
```

Từ kết quả của chương trình trên cho ta thấy khi truyền `x` vào hàm `test_reference` và khi kết thúc hàm thì `x` vẫn tồn tại và `value` không bị drop.

Mutable reference

Khi làm việc với một hàm mà ta cần phải thay đổi giá trị của nó chẳng hạn như hàm `swap` thì ta phải sử dụng `Mutable reference`.

Ví dụ:

```
fn main()
{
    let mut x = String::from("Xin chào ");

    mut_ref_test(&mut x);
    println!("{}", x);
}

fn mut_ref_test(x: &mut String)
{
    x.push_str("đồ án nhập môn CNTT hê hê.");
}
```

Kết quả:

```
Xin chào đồ án nhập môn CNTT hê hê.
```

Recap

> Reference là truyền thẳng `owner` cùng `value` của nó vào hàm và không cần trả về giá trị.
> Có hai loại `reference`, `mutable reference` và `immutable reference`.
> Dùng toán tử `&` và `&mut` để tham chiếu.

Struct

Concept

`Struct` là một chức năng cho phép chúng ta có thể gom nhiều kiểu dữ liệu có liên quan lại với nhau và tạo thành một kiểu dữ liệu mới.

Defining structs

```
struct <struct_name>(<list_of_data_type>);
```

Ví dụ:

```
struct Color(i32, i32, i32);
fn main()
{
    let cyan = Color(0, 255, 255);
}
```

Ta thấy kiểu `struct` này khá giống với `tuple`. Trong thực tế thì khá ít người sử dụng kiểu này, trừ những trường hợp đặc biệt (như khai báo màu, point,...). Thế nên ta có kiểu khai báo `struct` như sau:

```
struct <struct_name> {
    <field_name>: <data_type>,
}
```

Ví dụ:

```
struct Color{
    red: i32,
    green: i32,
    blue: i32
}
fn main()
```

```
{
    let cyan = Color{
        red: 0,
        green: 255,
        blue:  255
    };
}
```

Khi muốn truy cập đến các phần tử trong `struct` ta có thể sử dụng toán tử "`.`"

Ngoài ra, vì `Struct` là một kiểu dữ liệu nên ta cũng có thể viết hàm và trả về một `struct` khác.

Ví dụ:

```
use std::borrow::Borrow;

struct Color{
    red: i32,
    green: i32,
    blue: i32
}
fn main()
{
    let cyan = Color{
        red: 0,
        green: 255,
        blue:  255
    };

    let another_cyan = duplicated_color(cyan);
    println!("The RGB color of Cyan is {} {} {}", another_cyan.red,
another_cyan.green, another_cyan.blue);
}

fn duplicated_color(_color: Color) -> Color
{
    return _color
}
```

Rust còn cho phép chúng ta nhiều kiểu khởi tạo `struct` khác nhau. Sau đây là một ví dụ trong số đó:

```
#[derive(Debug)]
struct  Student {
    name: String,
    age: i32,
    height: i32
}

fn main()
{
```

```
    let student_1 = Student {
        name: String::from("Nguyễn Hoàng Phúc"),
        age: 19,
        height: 175
    };
    // một học sinh khác có cùng tuổi, cùng chiều cao nhưng khác tên thì ta có thể khai báo thông qua student_1 như sau:

    let student_2 = Student {
        name: String::from("Trần Gia Nghi"),
        ..student_1
        /*
        ..student_1 nghĩa là age và height của student_2 sẽ giống như của student_1
        */
    };

    println!("{:?}", student_1);
}
```

Kết quả của chương trình trên:

```
Student { name: "Trần Gia Nghi", age: 19, height: 175 }
```

Method

`Method` hay còn gọi là `phương thức`, về cơ bản nó khá giống với `hàm` nhưng vẫn có một vài đặc điểm khác nhau đáng chú ý. Như phương thức thì chỉ tồn tại trong một `struct` hoặc `class`.

Cú pháp:

```
impl <struct_name>
{
    fn <method_name>(<some_parameter>)
    {
        // some code
    }
}
```

Để gọi đến một `method` ta sử dụng toán tử "`.`". Khi thực hiện định nghĩa một `method`, thứ ta truyền vào sẽ là `self`, đại diện cho những biến chứa trong `struct` hiện hành. Sau đây là một ví dụ cơ bản về tính diện tích hình tròn:

```
struct Circle
{
    x: f32,
    y: f32,
```

```
    r: f32
}
fn main()
{
   let o = Circle
   {
        x: 1.0,
        y: 1.0,
        r: 2.0
   };
   println!("Diện tích hình tròn là: {}", o.area());
}

impl Circle
{
    fn area(self) -> f32
    {
        return self.r * 3.14
    }
}
```

Kết quả của chương trình trên là:

```
Diện tích hình tròn là: 6.28
```

Enum

`Enum` là một Cấu trúc dữ liệu được sử dụng khá phổ biến trong những ngôn ngữ Hướng Đối tượng, cho phép ta định nghĩa một dạng dữ liệu bằng cách liệt kê những tên(không phải giá trị) mà nó có thể có, thường được dùng để so sánh.

Cú pháp khai báo một `enum`.

```
enum <enum_name> {<list_of_value>}
```

Ví dụ về một `enum` chứa các môn chuyên ngành ở trường Đại học Khoa học Tự nhiên:

```
fn main()
{
    enum SUBJECT
    {
        IntroductionToProgramming,
        ProgrammingTechnique,
        DataStructureAndAlgorithm,
        ComputerNetwork,
        OjectOrientedProgramming
```

```
    }

    let subject = SUBJECT::ProgrammingTechnique;
}
```

Ngoài ra, mỗi phần tử trong `enum` còn có thể lưu giá trị nếu chúng được định nghĩa như ví dụ sau:

```
fn main()
{
    enum SUBJECT
    {
        IntroductionToProgramming(f32),
        ProgrammingTechnique(f32),
        DataStructureAndAlgorithm(f32),
        ComputerNetwork(f32),
        OjectOrientedProgramming(f32)
    }

    let subject = SUBJECT::IntroductionToProgramming(5.0);
}
```

Để sử dụng `enum` một cách tối ưu nhất, thông thường người ta sẽ sử dụng cùng với câu lệnh `match`.

Match

`Match` trong Rust có cấu trung tương đồng với `switch-case` trong C++. `Match` kiểm tra giá trị ở biến kiểm tra và trả về giá trị tương ứng của nó.

Ví dụ về `match` kết hợp với `enum`:

```
fn main()
{
    enum LANGUAGE
    {
        IntroductionToProgram,
        ProgrammingTechnique,
        DataStructureAndAlgorithm,
        ComputerNetwork,
        OjectOrientedProgramming
    }

    let subject = LANGUAGE::ProgrammingTechnique;

    let semester = match subject
    {
        LANGUAGE::IntroductionToProgram     => "1",
        LANGUAGE::ProgrammingTechnique      => "2",
        LANGUAGE::DataStructureAndAlgorithm | LANGUAGE::ComputerNetwork => "3",
```

```
        LANGUAGE::OjectOrientedProgramming  => "4",
        _ => "hông biết"
    };

    println!("Học môn Kỹ thuật lập trình ở kỳ {}", semester);
}
```

Kết quả của chương trình trên là:

```
Học môn Kỹ thuật lập trình ở kỳ 2
```

Trong chương trình trên, câu lệnh `match` trả về một giá trị có kiểu dữ liệu là `&str`. Dấu "_" được xem như là giá trị `default` trong câu lệnh `match`, khi không có giá trị nào được match, thì giá trị được trả về sẽ sau dấu "_".

Hashmap

Theo mặc định, `HashMap` sử dụng thuật toán băm được lựa chọn để cung cấp khả năng chống lại các cuộc tấn công `HashDoS`. Thuật toán được tạo `seed` ngẫu nhiên và một nỗ lực tốt nhất hợp lý được thực hiện để tạo ra `seed` này là từ một nguồn ngẫu nhiên chất lượng cao, an toàn được cung cấp bởi máy chủ mà không chặn chương trình. Bởi vì điều này, tính ngẫu nhiên của `seed` phụ thuộc vào chất lượng đầu ra của máy tạo số ngẫu nhiên của hệ thống khi `seed` được tạo ra. Đặc biệt, `seed` được tạo ra khi `entropy pool` của hệ thống thấp bất thường như trong quá trình khởi động hệ thống có thể có chất lượng thấp hơn.

Thuật toán băm mặc định hiện tại là `SipHash 1-3`, mặc dù điều này có thể thay đổi tại bất kỳ thời điểm nào trong tương lai. Mặc dù hiệu suất của nó rất cạnh tranh đối với các key cỡ trung bình, các thuật toán băm khác sẽ vượt trội hơn nó đối với các key nhỏ như số nguyên(`integers`) cũng như các key lớn như chuỗi dài (`long strings`), mặc dù các thuật toán đó thường sẽ không bảo vệ chống lại các cuộc tấn công như `HashDoS`.

Thuật toán băm có thể được thay thế trên cơ sở per-`HashMap` bằng cách sử dụng các phương pháp `default`, `with_hasher` và `with_capacity_and_hasher`. Có rất nhiều thuật toán băm thay thế có sẵn trên crates.io.

Các key phải kế thừa các trait Eq và Hash, mặc dù điều này thường có thể đạt được bằng cách sử dụng `#[derive (PartialEq, Eq, Hash)]`. Nếu bạn tự thực hiện những điều này, điều quan trọng là đặc tính sau đây được thỏa: `K1 == K2 -> hash(K1) == hash(K2)`

Nói cách khác, nếu hai key bằng nhau thì giá trị băm của chúng phải bằng nhau.

Nếu các vector lưu giá trị bằng một danh mục các số nguyên (`index`) thì `HashMap` lưu giá trị bằng các `key`. Các `key` của `HashMap` có thể là kiểu dữ liệu `booleans`, `intergers`, `strings` hay bất kì kiểu nào khác kế thừa `trait Eq` và `Hash`.

Như `vector`, `HashMap` có thể mở rộng, nhưng cũng có thể thu nhỏ nếu như có không gian dư. Bạn có thể tạo một `HashMap` với một dung lượng khởi đầu nhất định bằng cách sử dụng `HashMap::with_capacity(uint)` hoặc `HashMap::new()` để có một `HashMap` với một dung lượng khởi tạo mặc định(khuyên dùng).