

Make Interpreter in Rust, Week 1

A Tour of Rust, Part 1

Chris Ohk

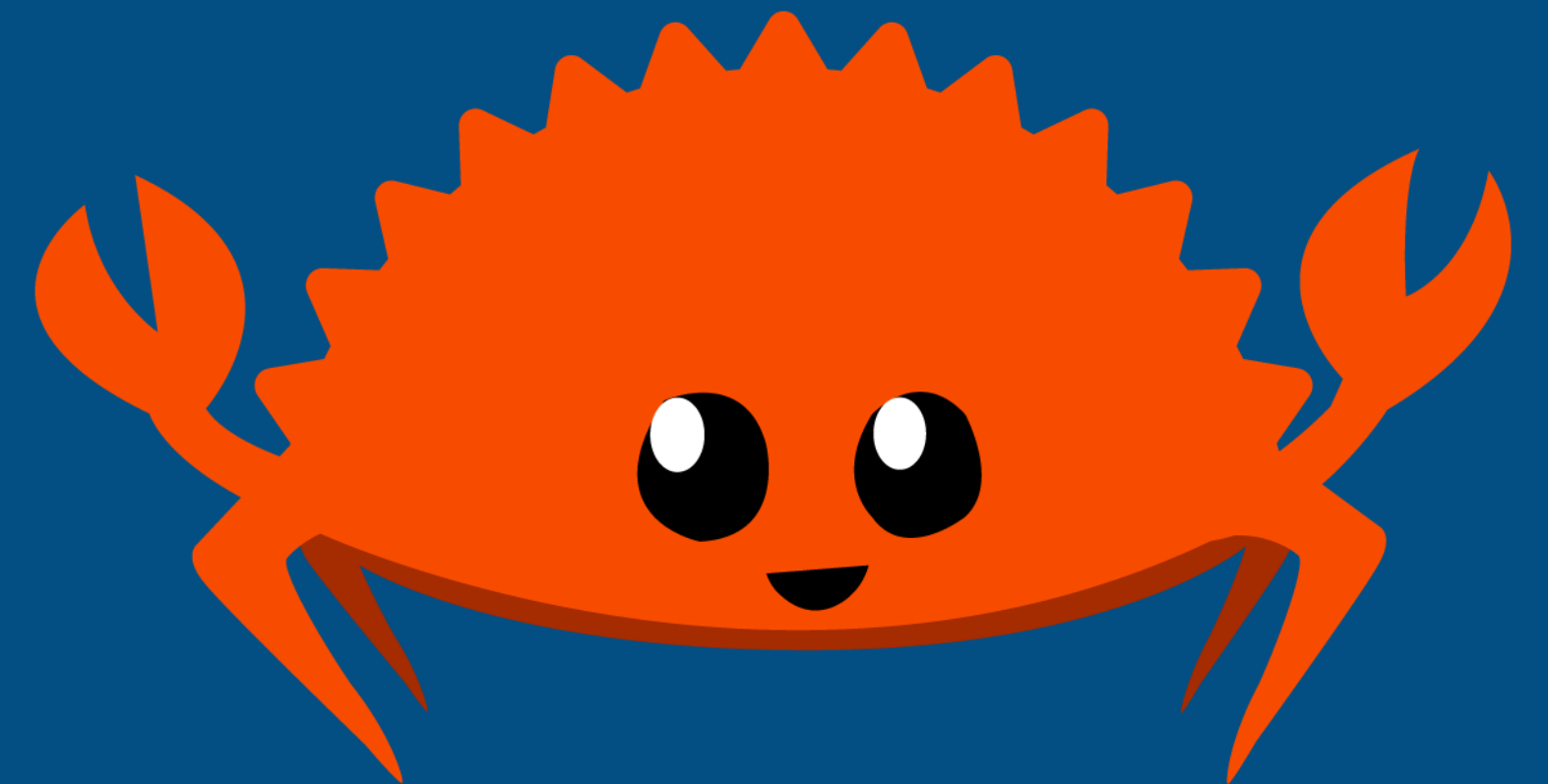
utilForever@gmail.com

- Rust란?
- Part 1
 - 기초
 - 기초적인 흐름 제어
 - 기본 데이터 구조 자료형
 - Generic 자료형
 - 소유권과 데이터 대여
- Part 2
 - 텍스트
 - 객체 지향 프로그래밍
 - 스마트 포인터
 - 프로젝트 구성과 구조

Rust란?

Make Interpreter in Rust
A Tour of Rust, Part 1

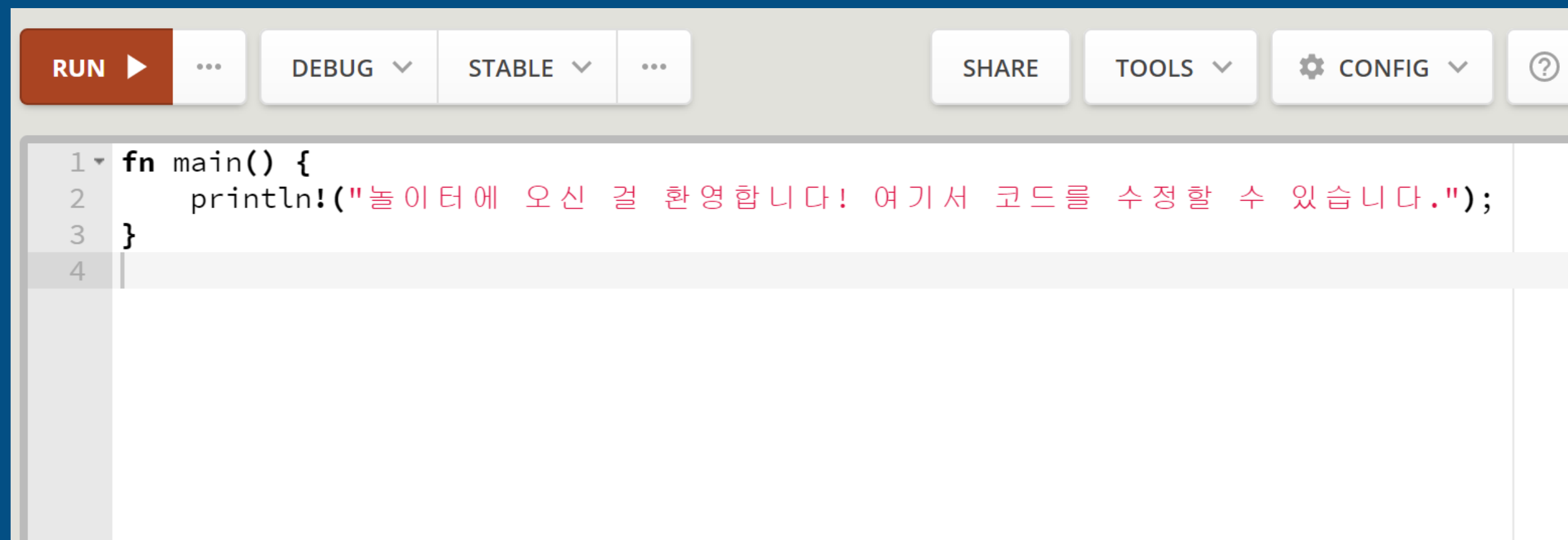
- <https://www.rust-lang.org/>
- 모질라 재단에서 2010년 7월 7일 처음 발표
- 현재는 러스트 재단으로 독립해서 개발되고 있다.
- Rust 언어의 특징
 - 안전한 메모리 관리
 - 철저한 예외나 에러 관리
 - 특이한 enum 시스템
 - 트레이트
 - 하이지닉 매크로
 - 비동기 프로그래밍
 - 제네릭



Rust Playground

Make Interpreter in Rust
A Tour of Rust, Part 1

- <https://play.rust-lang.org/>



Rust Disassembly

Make Interpreter in Rust
A Tour of Rust, Part 1

- <https://rust.godbolt.org/>

The screenshot displays the Godbolt Rust Disassembler interface. On the left, the Rust source code for a `main` function is shown, which prints the value of `x` (initially 20, then 10). On the right, the corresponding x86-64 assembly generated by `rustc 1.59.0` is displayed. The assembly includes stack frame setup, argument passing for `println!`, and the final `ret` instruction. The status bar at the bottom indicates the compilation took 766ms for 103248 bytes of code, resulting in 322 lines of assembly.

```
1 pub fn main() {
2     let mut x = 20;
3     println!("{}", x);
4
5     x = 10;
6     println!("{}", x);
7 }
```

```
1 example::main:
2     sub     rsp, 184
3     mov     dword ptr [rsp + 36], 20
4     lea     rax, [rsp + 36]
5     mov     qword ptr [rsp + 104], rax
6     mov     rdi, qword ptr [rsp + 104]
7     mov     rsi, qword ptr [rip + core::fmt::num::imp::<impl core::fmt::Display for i32>::fmt@GOTPCREL]
8     call    qword ptr [rip + core::fmt::ArgumentV1::new@GOTPCREL]
9     mov     qword ptr [rsp + 16], rax
10    mov     qword ptr [rsp + 24], rdx
11    mov     rax, qword ptr [rsp + 24]
12    mov     rcx, qword ptr [rsp + 16]
13    mov     qword ptr [rsp + 88], rcx
14    mov     qword ptr [rsp + 96], rax
15    lea     rcx, [rsp + 88]
16    lea     rdi, [rsp + 40]
17    lea     rsi, [rip + .L__unnamed_1]
18    mov     edx, 2
19    mov     r8d, 1
20    call     core::fmt::Arguments::new_v1
21    lea     rdi, [rsp + 40]
22    call     qword ptr [rip + std::io::stdio::_print@GOTPCREL]
23    mov     dword ptr [rsp + 36], 10
24    lea     rax, [rsp + 36]
25    mov     qword ptr [rsp + 176], rax
26    mov     rdi, qword ptr [rsp + 176]
27    mov     rsi, qword ptr [rip + core::fmt::num::imp::<impl core::fmt::Display for i32>::fmt@GOTPCREL]
28    call     qword ptr [rip + core::fmt::ArgumentV1::new@GOTPCREL]
29    mov     qword ptr [rsp], rax
30    mov     qword ptr [rsp + 8], rdx
31    mov     rax, qword ptr [rsp + 8]
32    mov     rcx, qword ptr [rsp]
33    mov     qword ptr [rsp + 160], rcx
34    mov     qword ptr [rsp + 168], rax
35    lea     rcx, [rsp + 160]
36    lea     rdi, [rsp + 112]
37    lea     rsi, [rip + .L__unnamed_1]
38    mov     edx, 2
39    mov     r8d, 1
40    call     core::fmt::Arguments::new_v1
41    lea     rdi, [rsp + 112]
42    call     qword ptr [rip + std::io::stdio::_print@GOTPCREL]
43    add     rsp, 184
44    ret
45
46 core::fmt::ArgumentV1::new:
47     sub     rsp, 56
```

- `let` 키워드를 사용
- 변수의 자료형을 대부분 유추할 수 있다.
- 변수 숨김 (Variable Shadowing)을 지원
- 변수의 이름은 언제나 `snake_case` 형태로 짓는다.

```
fn main() {  
    let x = 13;  
    println!("{}", x);  
  
    let x: f64 = 3.14159;  
    println!("{}", x);  
  
    let x;  
    x = 0;  
    println!("{}", x);  
}
```

- Rust에서 변수는 기본적으로 변경 불가(Immutable) 타입이다.
- 변경 가능(Mutable)한 값을 원한다면 **mut** 키워드로 표시해줘야 한다.

```
fn main() {  
    let mut x = 42;  
    println!("{}", x);  
    x = 13;  
    println!("{}", x);  
}
```

- 부울 값 - 참/거짓 값을 나타내는 `bool`
- 부호가 없는 정수형 - 양의 정수를 나타내는 `u8`, `u16`, `u32`, `u64`, `u128`
- 부호가 있는 정수형 - 양/음의 정수를 나타내는 `i8`, `i16`, `i32`, `i64`, `i128`
- 포인터 사이즈 정수 - 메모리에 있는 값들의 인덱스와 크기를 나타내는 `usize`, `isize`
- 부동 소수점 - `f32`, `f64`
- 튜플(tuple) - stack에 있는 값들의 고정된 순서를 전달하기 위한 `(값, 값, ...)`
- 배열(array) - 컴파일 타임에 정해진 길이를 갖는 유사한 원소들의 모음(collection)인 `[값, 값, ...]`
- 슬라이스(slice) - 런타임에 길이가 정해지는 유사한 원소들의 collection
- `str`(문자열 slice) - 런타임에 길이가 정해지는 텍스트

- 자료형 변환을 할 때는 **as** 키워드를 사용한다.
(Rust에서는 숫자형 자료형을 쓸 때 명시적으로 써야 한다.)

```
fn main() {  
    let a = 13u8;  
    let b = 7u32;  
    let c = a as u32 + b;  
    println!("{}", c);  
  
    let t = true;  
    println!("{}", t as u8);  
}
```

- 상수는 변수와 달리 반드시 명시적으로 자료형을 지정해야 한다.
- 상수의 이름은 언제나 **SCREAMING_SNAKE_CASE** 형태로 짓는다.

```
const PI: f32 = 3.14159;

fn main() {
    println!(
        PI
    );
}
```

- 고정된 길이로 된 모든 같은 자료형의 자료를 갖는 Collection
- `[T; N]`으로 표현한다.
 - `T`는 원소의 자료형
 - `N`은 컴파일 타임에 주어지는 고정된 길이
- 각각의 원소는 `[x]` 연산자로 가져올 수 있다.

```
fn main() {  
    let nums: [i32; 3] = [1, 2, 3];  
    println!("{:?}", nums);  
    println!("{}", nums[1]);  
}
```

- 함수는 0개 또는 그 이상의 인자를 가진다.
- 함수의 이름은 언제나 **snake_case** 형태로 짓는다.

```
fn add(x: i32, y: i32) -> i32 {  
    return x + y;  
}  
  
fn main() {  
    println!("{}", add(42, 13));  
}
```

여러개의 리턴 값

Make Interpreter in Rust
A Tour of Rust, Part 1

- 함수에서 튜플(Tuple)을 리턴하면 여러개의 값을 리턴할 수 있다.

```
fn swap(x: i32, y: i32) -> (i32, i32) {  
    return (y, x);  
}  
  
fn main() {  
    let result = swap(123, 321);  
    println!("{}", result.0, result.1);  
  
    let (a, b) = swap(result.0, result.1);  
    println!("{}", a, b);  
}
```

아무것도 리턴하지 않기

Make Interpreter in Rust
A Tour of Rust, Part 1

- 함수에 리턴형을 지정하지 않은 경우 빈 튜플을 리턴하는데, **()**로 표현한다.

```
fn make_nothing() -> () {  
    return ();  
}  
  
fn make_nothing2() {  
    // Do nothing  
}  
  
fn main() {  
    let a = make_nothing();  
    let b = make_nothing2();  
  
    println!("The value of a: {:?}", a);  
    println!("The value of b: {:?}", b);  
}
```

if/else if/else

- 조건문에 괄호가 없다.

```
fn main() {  
    let x = 42;  
    if x < 42 {  
        println!("Less than 42");  
    } else if x == 42 {  
        println!("Equal 42");  
    } else {  
        println!("Greater than 42");  
    }  
}
```

loop

- 무한 반복문이 필요할 때 사용



```
fn main() {  
    let mut x = 0;  
    loop {  
        x += 1;  
        if x == 42 {  
            break;  
        }  
    }  
    println!("{}", x);  
}
```


while

- 반복문에 조건을 간단히 넣을 수 있다.
- 조건의 평가 결과가 **false**인 경우, 종료한다.

```
fn main() {  
    let mut x = 0;  
    while x != 42 {  
        x += 1;  
    }  
}
```

for

- **..**연산자는 시작 숫자에서 끝 숫자 전까지의 숫자들을 생성하는 반복자를 만든다.****
- **..**연산자는 시작 숫자에서 끝 숫자까지의 숫자들을 생성하는 반복자를 만든다.****

```
fn main() {  
    for x in 0..5 {  
        println!("{}", x);  
    }  
  
    for x in 0..=5 {  
        println!("{}", x);  
    }  
}
```

for

- `.rev()` 함수를 사용해 값을 감소시키는 반복자로 바꿀 수 있다.
- `.step_by(n)` 함수를 사용해 값을 `n`만큼 증가시키는 반복자로 바꿀 수 있다.

```
fn main() {  
    for x in (0..5).rev() {  
        println!("{}", x);  
    }  
  
    for x in (0..5).step_by(2) {  
        println!("{}", x);  
    }  
}
```

match

- `switch`를 대체하는 구문
- 모든 케이스를 빠짐 없이 처리해야 한다.

```
fn main() {  
    let x = 42;  
  
    match x {  
        0 => {  
            println!("Found 0");  
        }  
        1 | 2 => {  
            println!("Found 1 or 2!");  
        }  
        3..=9 => {  
            println!("Found between 3 and 9!");  
        }  
        matched_num @ 10..=100 => {  
            println!("Found {} between 10 and 100!", matched_num);  
        }  
        _ => {  
            println!("Found something else!");  
        }  
    }  
}
```

- 필드(Field)들의 Collection
- 메모리 상에 필드들을 어떻게 배치할 지에 대한 컴파일러의 청사진

```
struct SeaCreature {  
    animal_type: String,  
    name: String,  
    arms: i32,  
    legs: i32,  
    weapon: String,  
}
```

- 스테틱 메소드(Static Methods)
 - 자료형 그 자체에 속하는 메소드
 - `::` 연산자를 이용해 호출
- 인스턴스 메소드(Instance Methods)
 - 자료형의 인스턴스에 속하는 메소드
 - `.` 연산자를 이용해 호출

```
fn main() {  
    let s = String::from("Hello world!");  
    println!("The length of {} is {}.", s, s.len());  
}
```

메모리에 데이터 생성하기

Make Interpreter in Rust
A Tour of Rust, Part 1

- 코드에서 구조체를 인스턴스화(Instantiate)하면 프로그램은 연관된 필드 데이터들을 메모리 상에 나란히 생성한다.
- 구조체의 필드값들은 **.** 연산자를 통해 접근한다.

```
struct SeaCreature {  
    animal_type: String,  
    name: String,  
    arms: i32,  
    legs: i32,  
    weapon: String,  
}  
  
fn main() {  
    let ferris = SeaCreature {  
        animal_type: String::from("crab"),  
        name: String::from("Ferris"),  
        arms: 2,  
        legs: 4,  
        weapon: String::from("claw"),  
    };  
}
```

- `enum` 키워드를 통해 몇 가지 태그된 원소의 값을 갖는 새로운 자료형을 생성할 수 있다.
- `match`와 함께 사용하면 품질 좋은 코드를 만들 수 있다.

```
enum Species {
    Crab,
    Octopus,
    Fish,
    Clam,
}

struct SeaCreature {
    species: Species,
    name: String,
}

fn main() {
    let ferris = SeaCreature {
        species: Species::Crab,
        name: String::from("Ferris"),
    };

    match ferris.species {
        Species::Crab => println!("{}", ferris.name),
        Species::Octopus => println!("{}", ferris.name),
        Species::Fish => println!("{}", ferris.name),
        Species::Clam => println!("{}", ferris.name),
    }
}
```


- `struct`나 `enum`을 부분적으로 정의해, 컴파일러가 컴파일 타임에 코드 사용을 기반으로 완전히 정의된 버전을 만들 수 있게 해준다.

```
struct BagOfHolding<T> {
    item: T,
}

fn main() {
    let i32_bag = BagOfHolding::<i32> { item: 42 };
    let bool_bag = BagOfHolding::<bool> { item: true };
    let float_bag = BagOfHolding { item: 3.14 };
    let bag_in_bag = BagOfHolding {
        item: BagOfHolding { item: "boom!" },
    };

    println!(
        "{} {} {} {}",
        i32_bag.item, bool_bag.item, float_bag.item, bag_in_bag.item.item
    );
}
```

Option

- null을 쓰지 않고도 Nullable한 값을 표현할 수 있는 내장된 Generic 열거체

```
enum Option<T> {  
    None,  
    Some(T),  
}
```

```
struct BagOfHolding<T> {  
    item: Option<T>,  
}  
  
fn main() {  
    let i32_bag = BagOfHolding::<i32> { item: None };  
    if i32_bag.item.is_none() {  
        println!("Nothing!")  
    } else {  
        println!("Found Something!")  
    }  
  
    let i32_bag = BagOfHolding::<i32> { item: Some(42) };  
    if i32_bag.item.is_some() {  
        println!("Found Something!")  
    } else {  
        println!("Nothing!")  
    }  
  
    match i32_bag.item {  
        Some(v) => println!("Found {}!", v),  
        None => println!("Nothing"),  
    }  
}
```

- 실패할 가능성이 있는 값을 리턴할 수 있도록 해주는 내장된 Generic 열거체

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

```
fn do_something_that_might_fail(i: i32) -> Result<f32, String> {  
    if i == 42 {  
        Ok(13.0)  
    } else {  
        Err(String::from("Not match!"))  
    }  
}  
  
fn main() {  
    let result = do_something_that_might_fail(12);  
  
    match result {  
        Ok(v) => println!("Found {}", v),  
        Err(e) => println!("Error: {}", e),  
    }  
}
```

- `Result`와 함께 쓸 수 있는 강력한 연산자 ?

```
do_something_that_might_fail()?

match do_something_that_might_fail() {
    Ok(v) => v,
    Err(e) => return Err(e),
}
```

- **Result**와 함께 쓸 수 있는 강력한 연산자 ?

```
fn do_something_that_might_fail(i: i32) -> Result<f32, String> {  
    if i == 42 {  
        Ok(13.0)  
    } else {  
        Err(String::from("Not match!"))  
    }  
}  
  
fn main() -> Result<(), String> {  
    let v = do_something_that_might_fail(42)?;  
    println!("Found {}", v);  
    Ok(())  
}
```

추한 옵션/결과 처리

Make Interpreter in Rust
A Tour of Rust, Part 1

- 간단한 코드를 짤 때에도 `Option/Result`를 쓰는 것은 귀찮은 일일 수 있다.
- `unwrap`이라는 함수를 사용해 빠르고 더러운 방식으로 값을 가져올 수 있다.
- `Option/Result` 내부의 값을 꺼내오고
- `enum`이 `None/Err`인 경우에는 panic!

```
fn do_something_that_might_fail(i: i32) -> Result<f32, String> {
    if i == 42 {
        Ok(13.0)
    } else {
        Err(String::from("Not match!"))
    }
}

fn main() -> Result<(), String> {
    let v = do_something_that_might_fail(42).unwrap();
    println!("Found {}", v);

    let v = do_something_that_might_fail(1).unwrap();
    println!("Found {}", v);

    Ok(())
}
```

- `Vec` 구조체로 표현하는 가변 크기의 리스트
- `vec!` 매크로를 통해 손쉽게 생성할 수 있다.
- `iter()` 메소드를 통해 반복자를 생성할 수 있다.

```
fn main() {  
    let mut float_vec = Vec::new();  
    float_vec.push(1.3);  
    float_vec.push(2.3);  
    float_vec.push(3.4);  
  
    let string_vec = vec![String::from("Hello"), String::from("World")];  
  
    for word in string_vec.iter() {  
        println!("{}", word);  
    }  
}
```

소유권 및 범위 기반 리소스 관리

Make Interpreter in Rust
A Tour of Rust, Part 1

- 자료형을 인스턴스화해 변수명에 할당(Binding)하면, Rust 컴파일러가 전체 생명 주기(Lifetime) 동안 검증할 메모리 리소스를 생성한다.
- 할당된 변수는 리소스의 소유자(Owner)라고 한다.
- Rust는 범위(Scope)가 끝나는 곳에서 리소스를 소멸하고 할당 해제한다.
이 소멸과 할당 해제를 의미하는 용어로 drop을 사용한다.
(C++에서는 Resource Acquisition Is Initialization(RAII)라고 부른다).
- 구조체가 Drop될 때 구조체 자신이 제일 먼저 Drop되고, 이후 그 자식들이 각각 Drop된다.

- 소유자가 함수의 인자로 전달되면, 소유권은 그 함수의 매개 변수로 이동(Move)된다.
- 이동된 이후에는 원래 함수에 있던 변수는 더 이상 사용할 수 없다.

```
struct Foo {  
    x: i32,  
}  
  
fn do_something(f: Foo) {  
    println!("{}", f.x);  
}  
  
fn main() {  
    let foo = Foo { x: 42 };  
    do_something(foo);  
}
```

소유권 리턴하기

Make Interpreter in Rust
A Tour of Rust, Part 1

- 소유권은 함수에서도 리턴될 수 있다.

```
struct Foo {  
    x: i32,  
}  
  
fn do_something() -> Foo {  
    Foo { x: 42 }  
}  
  
fn main() {  
    let foo = do_something();  
}
```

참조로 소유권 대여하기

Make Interpreter in Rust
A Tour of Rust, Part 1

- **&** 연산자를 통해 참조로 리소스에 대한 접근 권한을 대여할 수 있다.
- 참조도 다른 리소스와 마찬가지로 Drop된다.

```
struct Foo {  
    x: i32,  
}  
  
fn main() {  
    let foo = Foo { x: 42 };  
    let f = &foo;  
    println!("{}", f.x);  
}
```

참조로 변경 가능한 소유권 대여하기

Make Interpreter in Rust
A Tour of Rust, Part 1

- `&mut` 연산자를 통해 리소스에 대해 변경 가능한 접근 권한도 대여할 수 있다.
- 리소스의 소유자는 변경 가능하게 대여된 상태에서 이동되거나 변경될 수 없다.

```
struct Foo {  
    x: i32,  
}  
  
fn do_something(f: Foo) {  
    println!("{}", f.x);  
}  
  
fn main() {  
    let mut foo = Foo { x: 42 };  
    let f = &mut foo;  
  
    // do_something(foo)  
    // foo.x = 13;  
  
    f.x = 13;  
  
    println!("{}", foo.x);  
  
    foo.x = 7;  
  
    do_something(foo);  
}
```

- `&mut` 참조를 이용해 `*` 연산자로 소유자의 값을 설정할 수 있다.
- `*` 연산자로 소유자의 값의 복사본도 가져올 수 있다. (복사 가능한 경우만)

```
fn main() {  
    let mut foo = 42;  
    let f = &mut foo;  
    let bar = *f;  
    *f = 13;  
    println!("{}", bar);  
    println!("{}", foo);  
}
```

대여한 데이터 전달하기

Make Interpreter in Rust
A Tour of Rust, Part 1

- Rust의 참조 규칙
 - 단 하나의 변경 가능한 참조 또는 여러개의 변경 불가능한 참조만 허용하며, 둘 다는 안된다.
 - 참조는 그 소유자보다 더 오래 살 수 없다.
- 보통 함수로 참조를 넘겨줄 때에는 문제가 되지 않는다.

```
struct Foo {  
    x: i32,  
}  
  
fn do_something(f: &mut Foo) {  
    f.x += 1;  
}  
  
fn main() {  
    let mut foo = Foo { x: 42 };  
    do_something(&mut foo);  
    do_something(&mut foo);  
}
```

- 참조는 참조에도 사용될 수 있다.

```
struct Foo {  
    x: i32,  
}  
  
fn do_something(a: &Foo) -> &i32 {  
    return &a.x;  
}  
  
fn main() {  
    let mut foo = Foo { x: 42 };  
    let x = &mut foo.x;  
    *x = 13;  
    let y = do_something(&foo);  
    println!("{}", y);  
}
```

- Rust의 컴파일러는 모든 변수의 생명 주기를 이해하며 참조가 절대로 그 소유자보다 더 오래 존재하지 못하도록 검증을 시도한다.
- 함수에서는 어떤 매개 변수와 리턴 값이 서로 같은 생명 주기를 공유하는지 식별할 수 있도록 심볼로 표시해 명시적으로 생명 주기를 지정할 수 있다.
- 생명 주기 지정자는 언제나 **'**로 시작한다. (예 : **'a**, **'b**, **'c**)

명시적인 생명 주기

Make Interpreter in Rust
A Tour of Rust, Part 1

```
struct Foo {  
    x: i32,  
}  
  
fn do_something<'a>(foo: &'a Foo) -> &'a i32  
{  
    return &foo.x;  
}  
  
fn main() {  
    let mut foo = Foo { x: 42 };  
    let x = &mut foo.x;  
    *x = 13;  
    let y = do_something(&foo);  
    println!("{}", y);  
}
```

여러 개의 생명 주기

Make Interpreter in Rust
A Tour of Rust, Part 1

- 생명 주기 지정자는 컴파일러가 스스로 함수 매개 변수들의 생명 주기를 판별하지 못하는 경우, 이를 명시적으로 지정할 수 있게 도와준다.

```
struct Foo {  
    x: i32,  
}  
  
fn do_something<'a, 'b>(foo_a: &'a Foo, foo_b: &'b Foo) -> &'b i32  
{  
    println!("{}", foo_a.x);  
    println!("{}", foo_b.x);  
    return &foo_b.x;  
}  
  
fn main() {  
    let foo_a = Foo { x: 42 };  
    let foo_b = Foo { x: 12 };  
    let x = do_something(&foo_a, &foo_b);  
    println!("{}", x);  
}
```

- `static` 변수는 컴파일 타임에 생성되어 프로그램의 시작부터 끝까지 존재하는 메모리 리소스다. 이들은 명시적으로 자료형을 지정해 주어야 한다.
- `static` 생명 주기는 프로그램이 끝날 때까지 무한정 유지되는 메모리 리소스다. 따라서 `'static`이라는 특별한 생명 주기 지정자를 갖는다.
- `'static`한 리소스는 절대 Drop되지 않는다.
- 만약 `static` 생명 주기를 갖는 리소스가 참조를 포함하는 경우, 그들도 모두 `'static`이어야 한다. (그 이하의 것들은 충분히 오래 살아남지 못한다.)

정적인 생명주기

Make Interpreter in Rust
A Tour of Rust, Part 1

```
static PI: f64 = 3.1415;

fn main() {
    static mut SECRET: &'static str = "swordfish";

    let msg: &'static str = "Hello World!";
    let p: &'static f64 = &PI;
    println!("{}", msg, p);

    unsafe {
        SECRET = "abracadabra";
        println!("{}", SECRET);
    }
}
```

데이터 자료형의 생명주기

Make Interpreter in Rust
A Tour of Rust, Part 1

- 함수와 마찬가지로, 데이터 자료형의 구성원들도 생명 주기 지정자로 지정할 수 있다.
- Rust는 참조가 품고 있는 데이터 구조가 참조가 가리키는 소유자보다 절대 오래 살아남지 못하도록 검증한다.
- 아무것도 아닌 것을 가리키는 참조를 들고 다니는 구조체는 있을 수 없다!

```
struct Foo<'a> {  
    i:&'a i32  
}  
  
fn main() {  
    let x = 42;  
    let foo = Foo {  
        i: &x  
    };  
    println!("{}",foo.i);  
}
```

- <https://www.rust-lang.org/>
- <https://doc.rust-lang.org/book/>
- <https://tourofrust.com/>
- The Rust Programming Language (No Starch Press, 2019)

감사합니다!

강의 듣느라 고생 많았습니다.