RUST 언어 튜토리얼

김지현

2020.2.8

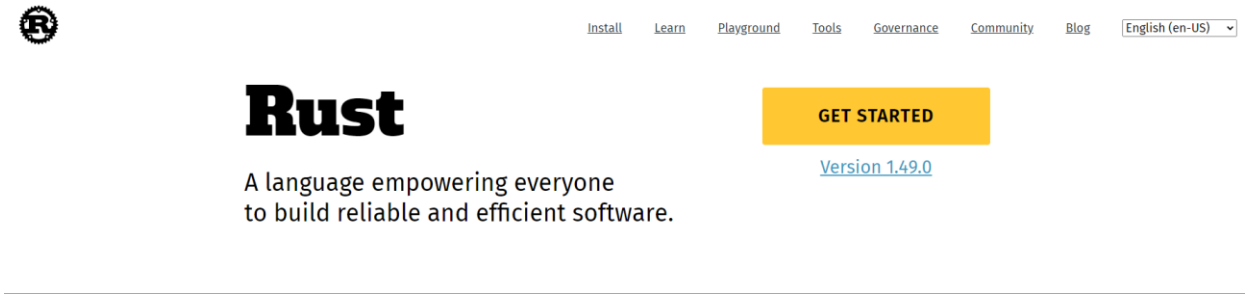
강점 : 컴파일 기반 언어, 동시성 프로그래밍/병렬프로그래밍

목표 : c나 c++와 동등한 수준의 속도를 달성하면서 메모리 오류를 완전히 없애는 것

목차

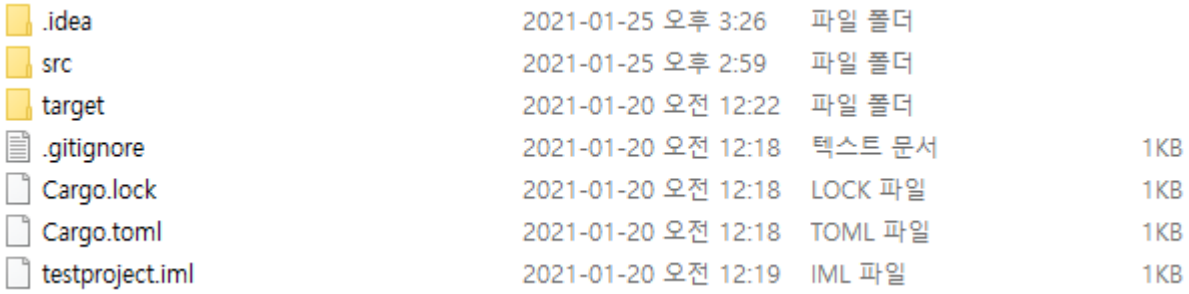
- Cargo 패키지 시스템
- cargo로 프로젝트 생성
- 상속 대신. **enum**, **struct**, **match**
- 자바 class 대신. **Trait**과 **Generic**
- **Ownership**과 **Lifetime**
- **Macro**, **Closures**
- Null, try catch 대신. **Option**과 **Result**

1. RUST 공식사이트(https://www.rust-lang.org/) 에서 컴파일러 설치



2. IDE Intelij(https://www.jetbrains.com/ko-kr/idea/) 설치

3. 새 프로젝트 생성(위치 : C:₩Users₩c₩IdeaProjects)
4. 구조



Primitives 기본형

1. 리터럴과 오퍼레이터

1)리터럴

- 정수형 : u 는 부호가 없고, i 는 부호가 있는것
- 부동형 : 1.2
- 문자형 : ‘a’
- 문자열 : “abc”
- Boolean : true, false
- 유닛형 : () –빈 튜플

***숫자는 ' _ '를 통해 가독성을 높일 수 있음

2)오퍼레이터

 연산자 허용과 우선순위는 C와 유사함

```
// 정수 더하기
println!("1 + 2 = {}", 1u32 + 2);

// 정수 빼기
println!("1 - 2 = {}", 1i32 - 2);

// 짧은 boolean 논리 연산
println!("true AND false is {}", true && false);
println!("true OR false is {}", true || false);
println!("NOT true is {}", !true);

// 비트 연산
println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
println!("1 << 5 is {}", 1u32 << 5);
println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);
```

2. 튜플과 배열

-튜플
- 서로 다른 타입의 값들의 집합체
- (T1,T2,...) T1, T2는 멤버의 타입
- 튜플은 함수의 인자, 반환값으로 사용가능함
- 색인으로 값을 추출 가능(index=0부터 시작)
- 튜플이 튜플의 멤버 가능
- 0을 제외한 어떤 숫자라도 인자로 받을 수 있음

-배열
같은 타입 T의 개체의 집합
선언 : let arrayname: [T; size] = [];
배열은 자동적으로 조각으로 변환하여 대여할 수 있음

-조각들(slices)
Slices의 크기는 컴파일 시에 알 수 없음
두-단어 개체 (첫단어 : 데이터 지칭, 두번째단어: 조각의 크기)
&[T]를 통해 배열의 구획을 대여해 사용할 수 있음

Custom types

- Rust 사용자 정의 데이터 타입
- 상수들은 Const 혹은 static 키워드로 생성 가능

1. Structures

세 타입의 구조체를 **struct** 키워드를 사용해 생성

- 튜플이라 이름 지어진 집합 구조체
- C 구조체들
- 필드를 갖지 않는 단위 구조체(제네릭에 유용함)

```
// 유닛 구조체
struct Nil;

// 튜플 구조체
struct Pair(i32, f32);

// 두 필드를 갖는 구조체
struct Point {
    x: f32,
    y: f32,
}

// 구조체는 다른 구조체의 필드로 사용될 수 있다.
#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}
```

2. Enums

1) 하나 혹은 몇 가지 서로 다른 변수형들로 이루어진 타입을 생성
   **struct 로서 유효한 변수형은 enum으로도 유효

2) Use : 범위 지정없이 사용가능, enum 내부 값을 편하게 사용가능

3) C의 enum과 유사하게 사용될 수 있음
  -> enums는 정수형으로 변환 사용가능

```
The poor have no money...
Civilians work!
```

```rust
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

enum Status {
    Rich,
    Poor,
}

enum Work {
    Civilian,
    Soldier,
}

fn main() {
    // `use`를 명시적으로 이름마다 사용하기에 범위 지정없이 사용 가능하다.
    use Status::{Poor, Rich};
    // `Work` 내부의 각 이름마다 자동으로 `use` 된다.
    use Work::*;

    let status = Poor;// `Status::Poor`와 동일.
    let work = Civilian;

    match status {
        // 앞에서 명시한 `use` 으로 인해 범위에 빈 틈이 있다.
        Rich => println!("The rich have lots of money!"),
        Poor => println!("The poor have no money..."),
    }

    match work {
        Civilian => println!("Civilians work!"),
        Soldier  => println!("Soldiers fight!"),
    }
}
```

2. Enums

1) 하나 혹은 몇 가지 서로 다른 변수형들로 이루어진 타입을 생성
   **struct 로서 유효한 변수형은 enum으로도 유효

2) Use : 범위 지정없이 사용가능, enum 내부 값을 편하게 사용가능

3) C의 enum과 유사하게 사용될 수 있음
** -> enums는 정수형으로 변환 사용가능**

```rust
// An attribute to hide warnings for unused code.
#![allow(dead_code)]

// 암시적으로 식별되는 enum (0에서 시작)
enum Number {
    Zero,
    One,
    Two,
}

// 명시적으로 식별 가능한 enum
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
    Blue = 0x0000ff,
}

fn main() {
    // `enums` 은 정수형으로 변환 사용 가능.
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

```
zero is 0
one is 1
roses are #ff0000
violets are #0000ff
```

Variable Bindings

- 컴파일러에 의한 추론할 수 있는 문맥상 타입

1. Mutuability

변수 바인딩은 불가변성이 기본 설정
재정의를 하기 위해 "mut" 식별자 사용

```
23      let _immutable_binding = 1;
24      let mut mutable_binding :i32 = 1;
25
26      println!("Before mutation: {}", mutable_binding);
27
28      // Ok
29      mutable_binding += 1;
30
31      println!("After mutation: {}", mutable_binding);
32
33      // Error!
34      //_immutable_binding += 1;
```

2. Scope and shadowing

- 변수 바인딩이 유지되는 범위는 블록(block)으로 제약
- { } 안에 바인딩은 블록 바깥에서 사용 불가

- Shadowing 허용 : 이전에 바인딩한 변수를 동일한 이름으로 바인딩 가능

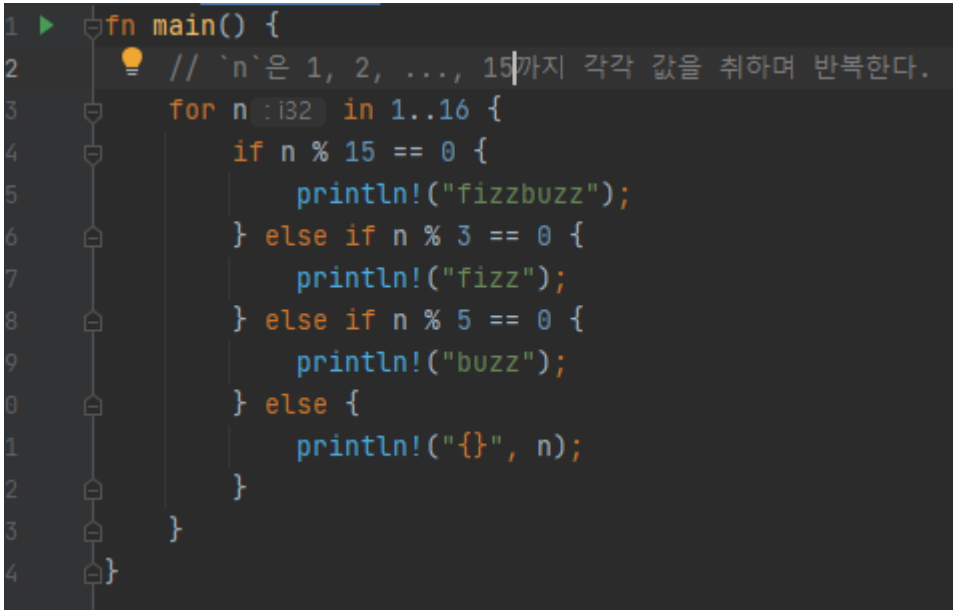
```
fn main() {
    // 이 바인딩은 메인 함수 내에 유효하다.
    let long_lived_binding :i32 = 1;

    // 이는 블럭으로 main 함수보다 작은 범위를 갖는다.
    {
        // 이 바인딩은 오직 이 블록 안에서만 존재한다.
        let short_lived_binding :i32 = 2;

        println!("inner short: {}", short_lived_binding);2

        // 이 바인딩은 외부 변수의 *shadows*
        let long_lived_binding :f32 = 5_f32;

        println!("inner long: {}", long_lived_binding); 5
    }
    // 블럭 종료.

    // 에러! `short_lived_binding` 이 범위에는 존재치 않음.

    println!("outer long: {}", long_lived_binding);   1

    // 이 바인딩 또한 이전 바인딩의 *shadows*
    let long_lived_binding :char = 'a';

    println!("outer long: {}", long_lived_binding);   a
}
```

Flow Control

- 제어 흐름을 변경시키는 것은 필수적

1. For and range

For in 문은 iterator를 통해 반복하는데 사용 가능함
쉬운 방법은 범위 표기법 a..b를 사용하는것
=> **a(포함) 부터 b(제외)**까지 1씩 증가된 값



2. match

```rust
fn main() {
    let number : i32 = 20;
    // TODO ^ `number`에 다른 값을 넣어보세요.

    println!("Tell me about {}", number);
    match number {
        // 하나의 값에 매치.
        1 => println!("One!"),
        // 다수의 값에 매치.
        2 | 3 | 5 | 7 | 11 => println!("This is a prime"),
        // 범위에 포함되는 매치.
        13...19 => println!("A teen"),
        // 나머지 경우의 처리.
        _ => println!("Ain't special"),
    }

    let boolean : bool = true;
    // 매치는 표현문이기도 함.
    let binary : i32 = match boolean {
        // 매치는 모든 가능한 값들을 범주에 포괄해야 함.
        false => 0,
        true => 1,

    };

    println!("{} -> {}", boolean, binary);
}
```

Match 키워드를 통해 패턴 매칭
⇒ C언어의 switch 처럼 사용

```
Tell me about 20
Ain't special
true -> 1
```

2. match

1) Match로 역구조화(**destructuring**) 할 수 있음

- 입력된 **튜플**과 enums 값을 match를 통해 값을 알 수 있음

```rust
fn main() {
    let pair : (i32, i32) = (0, -2);

    println!("Tell me about {:?}", pair);
    // 매치는 튜플의 역구조화 하는데 사용될 수 있다.
    match pair {
        // 두 번째 역구조화
        (0, y : i32 ) => println!("First is `0` and `y` is `{:?}`", y),
        (x : i32 , 0) => println!("`x` is `{:?}` and last is `0`", x),
        _      => println!("It doesn't matter what they are"),
        // `_`의 의미는 값을 변수에 바인드하지 않는 것.
    }
}
```

```
Tell me about (0, -2)
First is `0` and `y` is `-2`
```

- structs

```rust
fn main() {
    struct Foo { x: (u32, u32), y: u32 }

    // 구조체의 멤버를 역구조화
    let foo = Foo { x: (1, 2), y: 3 };
    let Foo { x: (a : u32 , b : u32 ), y : u32 } = foo;

    println!("a = {}, b = {},  y = {} ", a, b, y);

    // 구조체를 역구조화 시키거나 변수의 이름을 변경할 수 있고,
    // 순서는 중요치 않다.
    let Foo { y: i : u32 , x: j : (u32, u32) } = foo;
    println!("i = {:?}, j = {:?}", i, j);

    // 몇 가지 변수를 무시하는 것 또한 가능하다.
    let Foo { y : u32 , .. } = foo;
    println!("y = {}", y);

    // 이는 에러를 발생한다: `x` 필드에 대한 취급 패턴이 없기 때문.
    // let Foo { y } = foo;
}
```

```
a = 1, b = 2,  y = 3
i = 3, j = (1, 2)
y = 3
```

2. match

- pointers/ref
역참조 : * 사용
역구조화 : &, ref, ref mut 사용

```rust
fn main() {
    // 타입 `i32`의 참조를 할당. `&`는 참조가 할당될 것을 선언한다.
    let reference : &i32 = &4;

    match reference {
        // 만약 `reference`가 `&val`에 대해 패턴 매치 시키게 되면,
        // 그 결과는 `&i32`와 비교하는 것과 같다.
        // 우리가 살펴볼 것은 만약 매칭되면 `&`이 드랍되어,
        // `i32`가 `val`로 할당되게 된다는 점이다.
        &val : i32 => println!("Got a value via destructuring: {:?}", val),
    }

    // `&`를 생략하고 싶으면, 매칭시키기 전에 역참조하라.
    match *reference {
        val : i32 => println!("Got a value via dereferencing: {:?}", val),
    }

    // Rust가 제공하는 `ref`는 이를 명시하고자 하는 의도이다. 이는 할당을 수정하여
    // 요소를 위한 참조를 생성한다; 이 참조는 할당된다.
    let ref _is_a_reference = 3;

    // 따라서 참조 없이 두 값을 정의하면,
    // 참조는 `ref`와 `ref mut`을 통해 반환할 수 있다.
    let value : i32 = 5;
    let mut mut_value : i32 = 6;

    // `ref`키워드를 사용해 참조를 생성하게 된다.
    match value {
        ref r : &i32 => println!("Got a reference to a value: {:?}", r),
    }

    // `ref mut`의 사용도 유사하다.
    match mut_value {
        ref mut m : &mut i32 => {
            // 참조를 얻었다. 이를 역참조하여 여기에 어떤 값이든 더할 수 있다.
            *m += 10;
            println!("We added 10. `mut_value`: {:?}", m);
        },
    }
}
```

4

4

5

16

2. match

2) Binding
간접적으로 변수에 접근하는 방식은 분기하고서 이를 다시 사용하고자 하면 re-binding 하지 않으면 사용하는 것이 불가능 함
Match가 제공하는 @문장은 값을 이름으로 바인딩 함

```rust
// `age` 함수는 `u32`를 반환한다.
fn age() -> u32 {
    15
}

fn main() {
    println!("Tell me type of person you are");

    match age() {
        0             => println!("I'm not born yet I guess"),
        // `match`를 1 ... 12 까지 직접적으로 할 수 있지만
        // 그러면 그 하위에서는 몇 age인지 알 수가 없으니 `n`으로
        // 1 ... 12 순차를 바인드한다. 이제 age를 알 수 있게 된다.
        n :u32 @ 1  ... 12 => println!("I'm a child of age {:?}", n),
        n :u32 @ 13 ... 19 => println!("I'm a teen of age {:?}", n),
        // 바인드 없이 결과를 반환.
        n :u32             => println!("I'm an old person of age {:?}", n),
    }
}
```

3. if/while let

경우에 따라 match를 적용하기 힘들 때가 있다.
->문제점 : 공간 낭비

```rust
fn main() {
    let optional = Some(7);

    match optional {
        Some(i :i32 ) => {
            println!("This is a really long string and `{:?}`", i);
            // ^ 2번 들여쓰기 하고 나서야 `i`를 option에서 역구조화 할 수 있다.
        },
        _ => {},
        // ^ `match`가 완벽해야 하기 때문에 필요하다. 공간 낭비
    };
}
```

If/while let 을 이용하여, 필요할 때 실패에 대한 경우를 작성할 수 있다.

```rust
fn main() {
    // 모든 타입이 `Option`
    let number = Some(7);
    let letter: Option<i32> = None;
    let emoticon: Option<i32> = None;

    // 실패에 대한 기재가 필요할 경우, else를 사용
    if let Some(i :i32 ) = letter {
        println!("Matched {:?}!", i);
    } else {
        // 역구조화가 실패했다. 실패 사례로 변경.
        println!("Didn't match a number. Let's go with a letter!");
    };
}
```

Functions

- **fn** 키워드 사용
- 반환 값이 있을 경우, 반환타입은 -> 뒤에 명시

1. Methods

- 메소드들은 객체 내 데이터나 다른 메소드를 **self** 키워드를 통해 접근 함
- 메소드는 **impl** 블록안에서 정의

정의

```rust
struct Rectangle {
    p1: Point,
    p2: Point,
}

impl Rectangle {
    // 이것은 인스턴스 메소드이다.
    // `&self`는 문법적으로 `self: &Self`이고, `Self`는 호출되는 객체의 타입이다.
    // 여기서 `Self` = `Rectangle` 이다.
    fn area(&self) -> f64 {
        // `self`는 점 연산자를 통해 구조체 필드에 접근권을 준다.
        let Point { x: x1 : f64 , y: y1 : f64 } = self.p1;
        let Point { x: x2 : f64 , y: y2 : f64 } = self.p2;
        ((x1 - x2) * (y1 - y2)).abs()
    }

    fn perimeter(&self) -> f64 {
        let Point { x: x1 : f64 , y: y1 : f64 } = self.p1;
        let Point { x: x2 : f64 , y: y2 : f64 } = self.p2;
        2.0 * ((x1 - x2).abs() + (y1 - y2).abs())
    }

    // 이 메소드는 호출하는 객체가 가변적일 것을 요구한다.
    // `&mut self`는 `self: &mut Self`로 번역된다.
    fn translate(&mut self, x: f64, y: f64) {
        self.p1.x += x;
        self.p2.x += x;

        self.p1.y += y;
        self.p2.y += y;
    }
}
```

이용

```rust
fn main() {
    let rectangle = Rectangle {
        // 전역 메소드는 더블 콜론을 사용해 호출된다.
        p1: Point::origin(),
        p2: Point::new( x: 3.0,  y: 4.0),
    };
    // `rectangle.perimeter()` === `rectangle::perimeter(&rectangle)`
    println!("Rectangle perimeter: {}", rectangle.perimeter());
    println!("Rectangle area: {}", rectangle.area());

    let mut square = Rectangle {
        p1: Point::origin(),
        p2: Point::new( x: 1.0,  y: 1.0),
    };
```

```
14
12
```

2. Closures

환경을 캡쳐할 수 있는 익명함수
1) ( ) 대신 | | 안에 매개변수들을 넣음
2) 단일 표현문은 { } -> option
3) 외부 환경의 변수를 캡쳐할 수 있는 능력

```rust
fn main() {
    // 함수와 클로저를 통한 증가.
    fn  function            (i: i32) -> i32 { i + 1 }

    // 클로저는 익명이다, 여기에선 참조로 바인딩 합니다.
    // 주해는 함수 주해와 동일하지만 몸체를 감싸는 `{}`처럼 선택 사항이다.
    // 이런 이름없는 함수들는 적절하게 명명된 변수들에 할당됩니다.
    let closure_annotated : fn(i32) → i32  = |i: i32| -> i32 { i + 1 };
    let closure_inferred : fn(?) → ?   = |i     |          i + 1  ;

    let i : i32  = 1;
    // 함수와 클로저를 호출한다.
    println!("function: {}", function(i));
    println!("closure_annotated: {}", closure_annotated(i));
    println!("closure_inferred: {}", closure_inferred(i));

    // 인자를 취하지 않는 클로저는 `i32`를 반환한다.
    // 반환 타입은 추정된다.
    let one : fn() → i32  = || 1;
    println!("closure returning one: {}", one());

}
```

```
function: 2
closure_annotated: 2
closure_inferred: 2
closure returning one: 1
```

Modules

- **논리적 유닛(모듈)을 계층적으로 분리할때 사용**
- 모듈간의 가시성(public/private)를 관리
- 여기서 모듈은 : 함수, 구조체, trait, impl 등의 집합

1. Visibility

- 모듈의 아이템들은 기본적으로 **private** 가시성을 가짐
- **Pub** 수정자로 재정의 될 수 있음
- Public 아이템만 외부의 모듈의 영역에서도 접근 가능

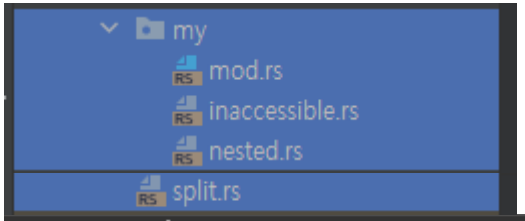
+중첩된 public 함수를 main에서 반복해서 호출할때 **use**를 사용해서 쉽게 사용 가능

+**self**를 이용해서 현재의 모듈 범위 참조
super 를 이용해서 외부모듈 범위 참조

```rust
// `my` 모듈
mod my {
    // 모듈의 아이템은 기본적으로 private의 가시성을 갖는다.
    fn private_function() {
        println!("called `my::private_function()`");
    }

    // `pub` 지시어를 통해 기본 가시성을 재정의 한다.
    pub fn function() {
        println!("called `my::function()`");
    }

    // 동일 모듈의 아이템은 다른 아이템이 private여도 접근할 수 있다.
    pub fn indirect_access() {
        print!("called `my::indirect_access()`, that\n> ");
        private_function();
    }

    // 모듈은 중첩 될 수 있다.
    pub mod nested {
        pub fn function() {
            println!("called `my::nested::function()`");
        }

        #[allow(dead_code)]
        fn private_function() {
            println!("called `my::nested::private_function()`");
        }
    }

    // 중첩 모듈도 동일한 가시성 규칙을 적용받는다.
    mod private_nested {
        #[allow(dead_code)]
        pub fn function() {
            println!("called `my::private_nested::function()`");
        }
    }
}

fn function() {
    println!("called `function()`");
}
```

Function() 함수가 중복되지만 허용함
단, my :: 를 통해서 구분해야함

2. File 계층 구조




```rust
//split.rs
mod my;

fn function() {
    println!("called `function()`");
}

fn main() {
    my::function();

    function();

    my::indirect_access();

    my::nested::function();
}
```

```
C:\Users\c\IdeaProjects\testproject\src>rustc split.rs && split
called `my::function()`
called `function()`
called `my::indirect_access()`, that
> called `my::private_function()`
called `my::nested::function()`
```

```rust
//mod.rs
mod inaccessible;
pub mod nested;

pub fn function() {
    println!("called `my::function()`");
}

fn private_function() {
    println!("called `my::private_function()`");
}

pub fn indirect_access() {
    print!("called `my::indirect_access()`, that\n> ");

    private_function();
}
```

```rust
//nested.rs
pub fn function() {
    println!("called `my::nested::function()`");
}

#[allow(dead_code)]
fn private_function() {
    println!("called `my::nested::private_function()`");
}
```

```rust
//inaccessible.rs
#[allow(dead_code)]
pub fn public_function() {
    println!("called `my::inaccessible::public_function()`");
}
```

Crates

- **RUST의 편집단위**
- 모듈들은 개별적으로 컴파일되지 않고 create파일로 병합되어 컴파일 됨

1. Library

라이브러리를 만들어 다른 create에 연결하는 방법

executable.rs

```rust
extern crate rary;

fn main() {
    rary::public_function();

    // 에러! `private_function` 은 private.
    //rary::private_function();

    rary::indirect_access();
}
```

rary.rs

```rust
// rary.rs
pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

1. Library

Rustc --create-type=lib (외부파일)

```
C:\Users\c\IdeaProjects\testproject\src>rustc --crate-type=lib rary.rs
```

라이브러리 생성

```
C:\Users\c\IdeaProjects\testproject\src>dir lib*
 C 드라이브의 볼륨에는 이름이 없습니다.
 볼륨 일련 번호: 84A0-6870

 C:\Users\c\IdeaProjects\testproject\src 디렉터리

2021-01-24  오후 04:20            11,958 library.rlib
               1개 파일              11,958 바이트
               0개 디렉터리  70,701,895,680 바이트 남음
```

라이브러리 생성 확인

Rustc (메인파일) --extern (외부파일이름)=(라이브러리파일)

```
C:\Users\c\IdeaProjects\testproject\src>rustc executable.rs --extern rary=library.rlib
```

Executable.rs 와 라이브러리 연결

```
C:\Users\c\IdeaProjects\testproject\src>executable.exe
called rary's `public_function()`
called rary's `indirect_access()`, that
> called rary's `private_function()`
```

연결 후 실행 결과

Generics

- 타입과 기능을 광범위한 범위로 일반화
- -fn foo <T> (T){ ... }

1. 구조체 내 Generics

두 타입을 이용한 제네릭
이어서 x와 y가 다른 타입
의 값일 수도 있는 Point

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

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

2. 메소드 내 Generics

T타입의 x필드에 대한 참조자를
반환하는 Point<T> 구조체 상에
x라는 이름의 메소드 정의

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

impl<T> Point<T> {
    fn x(&self) -> &T { &self.x }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

Scoping rules

- 범위는 소유, 대여, 생명주기에 중요한 역할
- 이는 컴파일러에게 대여가 유효한지, 리소스가 해제 될수 있는지, 변수가 생성되거나 소멸되는 것을 알림

1. RALL

언제 개체가 범위를 벗어나더라도 그 자신의 소멸자를 호출하여 자신의 리소스를 해제한다.
=> 자원 유출(leak) 을 막아줌

```rust
// raii.rs
fn create_box() {
    let _box1 = Box::new( x: 3i32);
    // `_box1`은 여기서 소멸되고, 메모리는 해제된다.
}

fn main() {
    // 힙에 정수를 할당
    let _box2 = Box::new( x: 5i32);
    {
        let _box3 = Box::new( x: 4i32);

        // `_box3`은 여기서 소멸되고, 메모리는 해제된다.
    }
    // `_box2`은 여기서 소멸되고, 메모리는 해제된다.
}
```

2. Ownership and moves

- 자원은 오직 하나의 소유자를 가짐.
  -> 리소스가 한번 이상 해제되는것을 방지.
- 주의!) 모든 변수가 자원을 소유하는 것은 아님
- Move(이동) : 할당을 수행할 때 or 함수에 인자를 값으로 전달할 때 자원의 소유권은 전달됨
  -> 자원 이동 후에, 이전 소유주는 더 이상 사용할 수 없음
  => dangling 포인터가 생성되는 것을 회피

↓

Dangling 포인터 : 해제된 메모리 영역을 가리키는 것

```rust
// 이 함수는 힙에 할당된 메모리의 소유권을 취한다.
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // `c`는 소멸되고 메모리는 해제된다.
}

fn main() {
    // _Stack_ 할당된 정수
    let x : u32 = 5u32;

    // `x`를 `y`에 *복사한다* - 리소스의 이동은 없다.
    let y : u32 = x;

    // 두 값은 독립적으로 사용될 수 있다.
    println!("x is {}, and y is {}", x, y);

    // `a`는 _heap_에 할당된 정수에 대한 포인터.
    let a : Box<i32> = Box::new( x: 5i32);

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

    // `a`를 `b`로 *옮긴다(Move)*
    let b : Box<i32> = a;
    // `a`의 포인터 주소가 b로 복사된다(데이터가 아닌).
    // 이제 둘다 동일한 힙에 할당된 데이터의 포인터지만
    // `b`가 현재 소유주이다.
    println!("b contains: {}", b);

    // 이 함수는 `b`로 부터 힙에 할당된 메모리에 대한 소유권을 취한다.
    destroy_box( c: b);

}
```

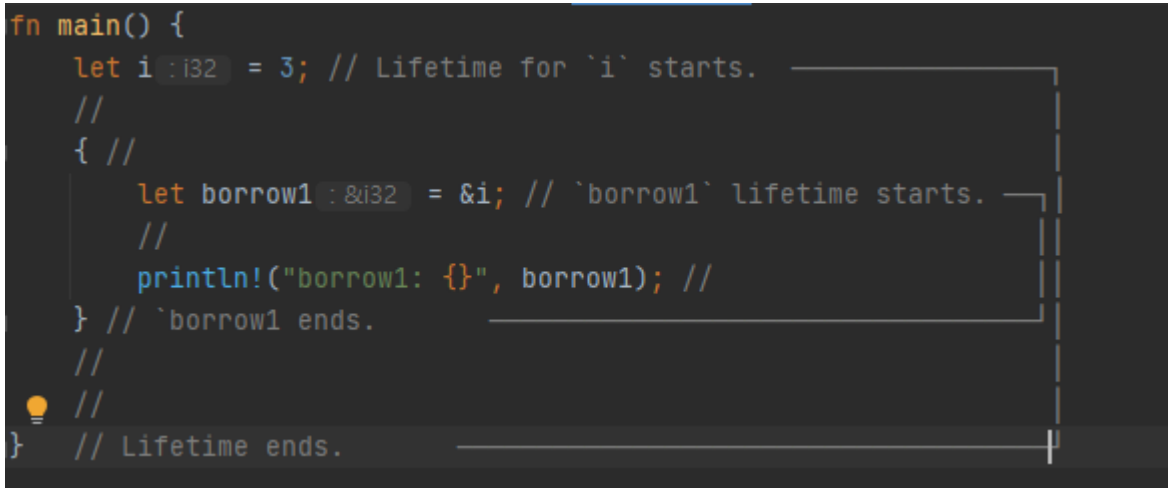
2. Ownership and moves

- Mutablility
  데이터의 가변성은 소유권이 이전되면서 변경될 수 있다.

```
fn main() {
    let immutable_box : Box<u32> = Box::new( x: 5u32);

    println!("immutable_box contains {}", immutable_box);

    // 가변성 에러!
    //*immutable_box = 4;

    // 박스를 *옮기자*, 소유권을 변경(가변성도)
    let mut mutable_box : Box<u32> = immutable_box;

    println!("mutable_box contains {}", mutable_box);

    // 박스의 내용을 수정.
    *mutable_box = 4;

    println!("mutable_box now contains {}", mutable_box);
}
```

3. Lifetimes

- 생명주기는 컴파일러에 의해 생성되어 모든 대여가 유효한지 확인하고자 사용
- 변수의 생명주기는 생성될때 시작하고, 소멸할때 끝난다.
- 변수는 **&**으로 대여, scope는 참조가 사용되는 곳에 의해 결정



3. Lifetimes

2) Explicit annotation (Function)
생명주기를 갖는 함수 선언은 몇 가지 제약이 있음

- 모든 참조는 반드시 생명주기 주해(')를 소지해야함
- 반환되는 모든 참조는 반드시 입력 인자 혹은 **static과 동일한 생명주기**를 가져야함

주의!) 인자없는 참조반환은 유효하지 않은 데이터의 참조를 결과로 반환하는 경우 금지됨

```rust
fn main() {
    let x :i32 = 7;
    let y :i32 = 9;

    print_one(&x);
    print_multi(&x, &y);

    let z :&i32 = pass_x(&x, &y);
    print_one( x: z);

    let mut t :i32 = 3;
    add_one( x: &mut t);
    print_one( x: &t);
}
```

7

7,9

7

4

```rust
// 생명주기 `'a`인 한 입력 참조는 반드시
// 최소 함수만큼 유지되어야 한다.
fn print_one<'a>(x: &'a i32) {
    println!("`print_one`: x is {}", x);
}

// 가변 참조도 생명주기와 함께 사용할 수 있다.
fn add_one<'a>(x: &'a mut i32) {
    *x += 1;
}

// 서로 다른 생명주기를 갖는 다수의 요소.
// 이 경우 동일 생명주기를 갖는다면 `'a`도 좋지만,
// 더 복잡한 경우라면, 다른 생명주기가 필요할 수 있다.
fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x is {}, y is {}", x, y);
}

// 전달된 참조의 반환은 허용된다.
fn pass_x<'a, 'b>(x: &'a i32, _: &'b i32) -> &'a i32 { x }

//fn invalid_output<'a>() -> &'a i32 { &7 }
// 위는 유효하지 않다. `'a`는 함수보다 반드시 오래 유지되어야 한다.
// 여기서 `&7`은 참조가 붙는 `i32`를 생성하고
// 이후 범위를 벗어나면서 데이터가 삭제되어
// 유효하지 않은 데이터의 참조가 반환되게 된다.
```

3. Lifetimes

생명주기를 지정하는 이유?
함수의 반환하는 값이 해제된 자원이면 에러, 해제 되지 않은 자원이면 에러없이 동작.
-> 모호성 해결을 위한것

3) Coercion
더 긴 생명주기를 짧은 것에 강제하여 일반적으로 동작할 수 없는 범위에서도 동작하게 할 수 있음

```rust
// 여기서, Rust는 가능한 짧은 생명주기를 추구한다.
// 그 후 두 참조에 해당 생명주기가 강제된다.
fn multiply<'a>(first: &'a i32, second: &'a i32) -> i32 {
    first * second
}

// `<'a: 'b,='' 'b=''>`는 생명주기 `'a`가 `'b`만큼 유지된다고 읽는다.
// 여기서 우리는 `&'a i32`를 취하고 `&'b i32`를 강제된 결과로 반환한다.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn main() {
    let first : i32 = 2; // 더 긴 생명주기

    {
        let second : i32 = 3; // 더 짧은 생명주기

        println!("The product is {}", multiply(&first, &second));
        println!("{} is the first", choose_first(&first, &second));
    };
}
```

```
The product is 6
2 is the first
```

3. Lifetimes

4) Static
'static' 생명주기는 가능한 가장 긴 생명주기이고 프로그램의 전체 생애주기를 가리킴
더 짧은 생명주기로 강제될 수 있음
변수를 'static'생명주기로 만드는 방법
-> 바이너리의 읽기전용 메모리에 저장

- Static 선언과 함께 상수 만들기
- &'static str 타입으로 string 리터럴 만들기

```
static_string: I'm in read-only memory
coerced_static: 18
NUM: 18 stays accessible!
```

```rust
// `'static` 생명주기로 상수를 만든다.
static NUM: i32 = 18;

// `'static`인 `NUM`의 참조를 반환한다.
// 생명주기는 입력 인자로 강제된다.
fn coerce_static<'a>(_: &'a i32) -> &'a i32 {
    &NUM
}

fn main() {
    {
        // `string` 리터럴을 만들고 출력한다:
        let static_string : &str  = "I'm in read-only memory";
        println!("static_string: {}", static_string);

        // `static_string`이 범위에서 벗어나면,
        // 참조는 더 이상 사용될 수 없지만 데이터는 바이너리에 남아있다.
    }

    {
        // `coerce_static`에 사용할 정수 생성.
        let lifetime_num : i32  = 9;

        // `NUM`을 `lifetime_num`의 생명주기로 강제한다:
        let coerced_static : &i32  = coerce_static(&lifetime_num);

        println!("coerced_static: {}", coerced_static);
    }

    println!("NUM: {} stays accessible!", NUM);
}
```

Traits

- 다른 종류의 추상화 가능
- 메소드의 모음으로 알 수 없는 타입에 정의된다.
- Self: 동일 trait에 선언된 다른 메소드에 접근할 수 있음

1. Derive

컴파일러는 #[derive] 속성을 통해 몇 trait를 위한 기본적인 구현을 제공함
Derive가능한 traits

1. Trait 비교자 : Eq, PartialEq, Ord, PartialOrd
2. Debug, {:?} : 형식자를 사용해 값을 형식화 하려면

```rust
fn main() {
    let foot = Inches(12);

    println!("One foot equals {:?}", foot);

    let meter = Centimeters(100.0);

    let cmp : &str =
        if foot.to_centimeters() < meter {
            "smaller"
        } else {
            "bigger"
        };

    println!("One foot is {} than one meter.", cmp);
}
```

```rust
// 비교할 수 있는 튜플 구조체 `Centimeters`
#[derive(PartialEq, PartialOrd)]
struct Centimeters(f64);

// 출력할 수 있는 튜플 구조체 `Inches`
#[derive(Debug)]
struct Inches(i32);

impl Inches {
    fn to_centimeters(&self) -> Centimeters {
        let &Inches(inches : i32 ) = self;
        Centimeters(inches as f64 * 2.54)
    }
}

// 추가 속성이 없는 튜플 구조체 `Seconds`
struct Seconds(i32);
```

2.Operator Overloading

- 연산자들이 trait을 통해 오버로드 될 수 있음
- 입력 인수에 맞추어 다른 작업을 수행할 수 있음

```rust
use std::ops;

struct Foo;
struct Bar;

#[derive(Debug)]
struct FooBar;

#[derive(Debug)]
struct BarFoo;
```

```rust
// 다음에 오는 블럭이 구현하는 연산: Foo + Bar = FooBar
impl ops::Add<Bar> for Foo {
    type Output = FooBar;

    fn add(self, _rhs: Bar) -> FooBar {
        println!("> Foo.add(Bar) was called");

        FooBar
    }
}

// 이 블럭이 구현하는 연산: Bar + Foo = BarFoo
impl ops::Add<Foo> for Bar {
    type Output = BarFoo;

    fn add(self, _rhs: Foo) -> BarFoo {
        println!("> Bar.add(Foo) was called");

        BarFoo
    }
}

fn main() {
    println!("Foo + Bar = {:?}", Foo + Bar);
    println!("Bar + Foo = {:?}", Bar + Foo);
}
```

3. Clone

리소스를 복사하는 것을 도와 줌.
Clone trait에 정의된 .clone()을 사용함.

```rust
// A unit struct without resources
#[derive(Debug, Clone, Copy)]
struct Nil;

// `Clone` trait을 구현하는 리소스를 가진 튜플 구조체
#[derive(Clone, Debug)]
struct Pair(Box<i32>, Box<i32>);

fn main() {
    // `Nil` 인스턴스 생성
    let nil = Nil;
    // `Nil` 복사, 이동시킬 리소스가 없다.
    let copied_nil : Nil = nil;

    // 두 `Nil` 모두 독립적으로 사용할 수 있다.
    println!("original: {:?}", nil);
    println!("copy: {:?}", copied_nil);

    // `Pair` 인스턴스 생성
    let pair = Pair(Box::new( x: 1), Box::new( x: 2));
    println!("original: {:?}", pair);

    // `pair`를 `moved_pair`로 복사하며, 리소스도 이동시킨다.
    let moved_pair : Pair = pair;
    println!("copy: {:?}", moved_pair);

    // `moved_pair`를 `cloned_pair`로 복제한다.(리소스 포함)
    let cloned_pair : Pair = moved_pair.clone();
    // std::mem:drop을 사용하여 원본 pair를 Drop시킨다.
    drop( _x: moved_pair);

    // .clone()을 통해 생성된 것은 아직 사용할 수 있다.
    println!("clone: {:?}", cloned_pair);
}
```

1) = 연산을 이용하면 leftvalue에 복사, 이동
2) .clone() 연산을 이용하면 복제, drop으로 원본 제거가능

Macro_rules!

- 매크로들은 함수와 유사하지만 그들의 이름은 !로 끝남
- 매크로는 소스 코드에 확장되어 나머지 프로그램과 함께 컴파일 됨

1. Designators

매크로의 인자는 접두사로 $ 표시가 들어가고 지정자(designator)로 타입적용됨

```
macro_rules! create_function {
    // 이 매크로는 지정자`ident`의 인수를 취하고
    // `$func_name`으로 이름지어진 함수를 생성한다.
    // `ident`지정자는 변수/함수 이름으로 사용된다.
    ($func_name:ident) => (
        fn $func_name() {
            // `stringify!`매크로가 `ident`를 string으로 변환한다.
            println!("You called {:?}()",
                     stringify!($func_name))
        }
    )
}

// 상기 매크로를 통해 `foo`와 `bar`로 이름지어진 함수들을 생성한다.
create_function!(foo);
create_function!(bar);

macro_rules! print_result {
    // 이 매크로는 `expr`타입의 표현식을 취해 결과와 함께 문자열로 출력한다.
    // `expr`지정자가 표현식에 사용된다.
    ($expression:expr) => (
        // `stringify!`는 표현식을 *있는 그대로* string으로 변환한다.
        println!("{:?} = {:?}",
                 stringify!($expression),
                 $expression)
    )
}
```

```
fn main() {
    foo();
    bar();

    print_result!(1u32 + 1);

    // 표현식이기도 한 블럭을 통해 다시 호출!
    print_result!({
        let x = 1u32;

        x * x + 2 * x - 1
    });
}
```

```
You called "foo"()
You called "bar"()
"1u32 + 1" = 2
"{ let x = 1u32; x * x + 2 * x - 1 }" = 2
```

2. Overlaod

매크로는 오버로드 되어 다양한 인자 조합을 허용할 수 있음
-> match블록과 유사하게 동작

```
// `test!`는 `$left`와 `$right`를 비교할 것이다.
// 당신이 어떻게 호출할 지에 따라 다른 방법으로:
macro_rules! test {
    // 인자들은 쉼표로 구분될 필요가 없다.
    // 모든 형태가 사용될 수 있다!
    ($left:expr; and $right:expr) => (
        println!("{:?} and {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                 $left && $right)
    );
    // ^ 각 문장은 반드시 세미콜론으로 끝나야 함!!
    ($left:expr; or $right:expr) => (
        println!("{:?} or {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                 $left || $right)
    );
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
    test!(true; or false);
}
```

```
"1i32 + 1 == 2i32" and "2i32 * 2 == 4i32" is true
"true" or "false" is true
```

Std library types

- Std 라이브러리들이 제공하는 많은 타입은 기본형들을 확장함

1. Option

때로는 panic! 호출대신 프로그램 일부분의 실패를 잡는 것이 바람직하다.
Option enum 을 사용
Option<T> enum은 두 형태를 가짐

- None : 실패 혹은 값이 부족한 경우
- Some(value) : 타입 T로 포장된 value인 튜플 구조체

```rust
// `panic!`하지 않는 정수 분할
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        // 실패는 `None`의 형태로 표현된다.
        None
    } else {
        // `Some`의 형태로 포장된 결과.
        Some(dividend / divisor)
    }
}

// 이 함수는 성공하지 못할 수 있는 분할을 처리한다.
fn try_division(dividend: i32, divisor: i32) {
    // `Option`값은 다른 enum과 마찬가지로 패턴 매칭 될 수 있다.
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient: i32) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        },
    }
}
```

```rust
fn main() {
    try_division( dividend: 4,  divisor: 2);
    try_division( dividend: 1,  divisor: 0);

    // 변수에 `None`을 바인딩 하려면 타입 주해가 필요하다.
    let none: Option<i32> = None;
    let _equivalent_none = None::<i32>;

    let optional_float = Some(0f32);

    // `Some`형을 언래핑하면 래핑됐던 값이 추출되게 된다.
    println!("{:?} unwraps to {:?}", optional_float, optional_float.unwrap());

    // `None` 형을 언래핑하면 `panic!`된다.
    println!("{:?} unwraps to {:?}", none, none.unwrap());
}
```

```
4 / 2 = 2
1 / 0 failed!
Some(0.0) unwraps to 0.0
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', src\main.rs:37:49
stack backtrace:
<4 internal calls>
   4: testproject::main
             at .\src\main.rs:37
<1 internal call>
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
error: process didn't exit successfully: `target\debug\testproject.exe` (exit code: 101)

Process finished with exit code 101
```

2. Result

Option으로 실패할 수 있는 함수 반환 값으로 사용될 수 있지만, 가끔 동작이 왜 실패하는지 표현하는게 중요함.

Result<T, E> enum은 두 가지 변수형을 가짐

- Ok(value)가 나타내는 것은 동작이 성공했다는 것, 연산이 반환하는 value를 포장
- Err(why)가 나타내는 것은 동작이 실패했다는것, 실패의 원인을 설명하는 why를 포장

```rust
// `op(x, y)` === `sqrt(ln(x / y))`
fn op(x: f64, y: f64) -> f64 {
    // 삼단 레벨 match 피라미드!
    match checked::div(x, y) {
        Err(why : MathError ) => panic!("{:?}", why),
        Ok(ratio : f64 ) => match checked::ln( x: ratio) {
            Err(why : MathError ) => panic!("{:?}", why),
            Ok(ln : f64 ) => match checked::sqrt( x: ln) {
                //입력값이 음수라서 err
                Err(why : MathError ) => panic!("{:?}", why),
                Ok(sqrt : f64 ) => sqrt,
            },
        },
    }
}

fn main() {
    // 이건 실패할까?
    println!("{}", op( x: 1.0,  y: 10.0));
}
```

```
thread 'main' panicked at 'NegativeSquareRoot', src\main.rs:49:29
stack backtrace:
<2 internal calls>
   2: testproject::op
             at .\src\main.rs:49
   3: testproject::main
             at .\src\main.rs:58
<1 internal call>
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
error: process didn't exit successfully: `target\debug\testproject.exe` (exit code: 101)

Process finished with exit code 101
```