Rust Study, 2022.05.13. Friday

시작하기

■ 설치

설치하기:

```
$ curl https://sh.rustup.rs -sSf | sh
```

설치되면,

```
Rust is installed now. Great!
```

터미널 재시작하면 러스트는 시스템 패스에 추가됨.

재시작 하지 않고 러스트 바로 사용하려면:

```
$ source $HOME/.cargo/env
```

설치확인:

```
$ rustc --version
```

■ 테스트 프로그램 실행

~/project/hello_world 디렉토리에 main.rc 만들어 실행하기:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
$ vi main.rc

fn main() {
    println!("Hello, world!");
}
```

```
$ rustc main.rs
$ ./main
Hello, world!
```

Note 1: 들여쓰기는 Tab(X), Space x 4 (O)
Note 2: println은 함수, println!은 매크로
Note 3: 라인의 끝은 항상 ;로

■ 쉬운 의존성 추가를 위한 빌드 시스템 및 패키지 매니저를 사용한 테스트 프로그램 작성

빌드 시스템 및 패키지 매니저 설치 확인:

```
$ cargo --version
```

프로젝트 생성:

```
$ cargo new hello_cargo --bin
```

Note: cargo new hello_cargo --bin 으로 hello_cargo란 프로젝트를 위한 관련 디렉토리 및 파일 생성

- hello_cargo/Cargo.toml 생성
  (Tom's Obvious, Minimal Language, 의존성 관리 등 환경설정 파일)
- hello_cargo/src/main.rc 생성

Cargo.toml 내용:

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
[dependencies]
```

Note: 의존성 부분엔 러스트에서 코드 패키지 역할을 하는 크레이트(crate)가 적힘.

main.rc를 이전 테스트 함수처럼 쓴 뒤에 빌드하기:

```
$ cd hello_cargo
$ vi ./src/main.rc
```

```
fn main() {
    println!("Hello, world!");
}
$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
$ ./target/debug/hello_cargo # or .\target\debug\hello_cargo.exe on Windows
Hello, world!
```

Note 1:

- rustc를 이용하면 main.rc가 포함된 디렉토리에 실행파일 만드는 반면, cargo build를 사용해 빌드하면 ./target/degub/ 에 실행 파일 생성

빌드와 실행 한번에 하기 (이미 빌드 했을 때 결과):

```
$ cd ~/project/hello_cargo
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

→ 코드가 빌드된 상태에서 새로 빌드할 필요가 없어 빌드한다는 경과 출릭이 없다.

빌드와 실행 한번에 하기 (코드 수정 후 새로 빌드할 때 결과):

```
$ cd ~/project/hello_cargo
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

→ 새로 빌드해야 하기 때문에 빌드(compiling)한다는 경과 출력이 있다.

실행하지는 않고, 빌드 가능한지 여부만 확인하기:

```
$ cd ~/project/hello_cargo
$ cargo check
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs
```

추리 게임 튜토리얼

[1] 프로젝트 생성

```
$ cd ~/projects/
$ cargo new guessing_game --bin
$ cd guessing_game
```

result:

```
[yhhyun@localhost guessing_game]$ ls
Cargo.toml  src
```

[2] 추리값 입력 받아 출력하기

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    printnl!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("Failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Note:

- extern create rand; 외부의존 크레이트 명시
- use rand:: 로 rand내 모든 것 호출.
  - use rand::Rng → Rng 정수생성 메소드 정의한 trait
  - rand::thread_rng 함수는 OS가 시드(seed)를 정하고 현재 스레드에서만 사용되는 특별한 정수생성기를 돌려 줍니다?
- use std::io; → "표준라이브러리 std의 입출력관련라이브러리 io를 사용하겠다."
  - io 라이브러리를 스코프로 가져오기.
  - Rust는 모든 프로그램 영역에 prelude내 타입들을 가져온다.
- let mut guess = String::new();
  - let foo = bar; → 불변(immutable) 변수 지정
  - let mut bar = 5; → 가변(mutable) 변수 지정
  - guess 변수는 사용자가 매번 추리값을 입력한 값이 저장되므로 가변 변수로 mut guess로 정의
  - String::new(); → String 타입의 함수 new() 호출
- read_line()은 io::Result {Ok, Err}를 return
- println!("... {}", guess); {}:placeholder
  - println!("x = {} and y = {}", x, y); → {} 여러개면 변수 순서대로 대입
- let guess: u32 = guess.trim().parse().expect("Please type a number!");
  - guess의 타입을 32비트 정수형 u32(unsigned 32bit)로 바꿈
  - trim()으로 앞뒤빈칸, ₩n등을 없앰
  - parse()로 문자열을 숫자형으로 바꿈
- guess.cmp(&secret_number) → geuss와 secret_number 비교하기
  - cmp는 {Less, Greater, Equal}의 Result형식의 결과를 Ordering에 줌.
  - 각각은 {Ordering::Less, Ordering::Greater, Ordering::Equal}로 표현됨.

[3] 비밀번호 생성하기

Cargo.toml 수정:

```
[dependencies]

rand = "0.3.14"
```

→ 버전을 명시하는 Semantic Versioning(semver) 이용으로 0.3.14와 호환되는 API를 제공하는 모든 버전을 뜻함(?).

Build 하면:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
```

```
 Downloading rand v0.3.14
 Downloading libc v0.2.14
   Compiling libc v0.2.14
   Compiling rand v0.3.14
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

→ rand 크레이트를 의존성에 더했지만, rand는 libc에 의존해서 libc도 컴파일 됨을 볼 수 있다.
→ 빌드 후 Cargo.lock에 빌드 성공한 패키지의 버전들을 기록해놓는다.
후에 모든 버전을 확인하지 않고도 lock파일로 재현가능한 빌드가 가능해진다.

Update하기:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating rand v0.3.14 -> v0.3.15
```

→ Cargo는 기본적으로 0.3.0보다 크고 0.4.0보다 작은 버전에서 최상의 최신 버전을 찾는다.
→ 0.4.x의 파일들로 바꾸고 싶으면 Cargo.toml 파일을 rand = "0.4.0"으로 바꿔라.

보편적인 프로그래밍 개념

다른 프로그램언어와 마찬가지로 변수나 함수명으로 쓸 수 없는 keyword들이 있다.

- as - 캐스팅하거나, 항목을 포함하는 특정 트레잇을 명확히 하거나, use 와 extern crate 구문에서 항목의 이름을 변경
- break - 반복문 즉각 탈출
- const - 상수 혹은 상수 로우 포인터 정의
- continue - 다음 반복 루프로 넘어감
- crate - 외부 크레이트를 링크하거나 해당 매크로가 정의되어 있는 크레이트를 대표하는 매크로 변수를 생성합니다.
- else - if 와 if let 제어 흐름 구조에 대한 대비책
- enum - 열거형 정의
- extern - 외부 크레이트, 함수 혹은 변수를 링크
- false - Boolean 의 거짓(false)을 나타내는 상수
- fn - 함수 혹은 함수 포인터 타입 정의
- for - 반복자의 항목들을 반복하거나, 트레잇을 구현하거나, 더 높은 수준의 라이프타임을 명시
- if - 조건식 결과를 이용한 분기

- impl - 내재된 특성 혹은 트레잇 특성 구현
- in - for 반복문 문법의 일부
- let - 변수 바인딩
- loop - 무조건적인 반복
- match - 패턴에 값을 매치
- mod - 모듈 정의
- move - 클로저가 사용하는 모든 값에 대해 소유권을 갖도록 만듬
- mut - 레퍼런스, 로우 포인터, 배턴 바인딩에 대한 가변성 표시
- pub - 구조체 필드, impl 블록, 모듈의 public 가시성 표시
- ref - 레퍼런스로 바인딩
- return - 함수의 반환
- Self - 트레잇을 구현하고 있는 타입의 별칭
- self - 메소드의 주체 혹은 현재 모듈
- static - 글로벌 변수 혹은 전체 프로그램 실행에서 지속되는 라이프타임
- struct - 구조체 선언
- super - 현재 모듈의 부모 모듈
- trait - 트레잇 선언
- true - Boolean 의 참(true)을 나타내는 상수
- type - 타입 별칭 혹은 관련 타입 선언
- unsafe - 코드, 함수, 트레잇, 구현이 안전하지 않다는 것을 표시
- use - 심볼을 범위 내로 불러옴
- where - 특정 타입으로 제한하는 절을 나타냄
- while - 표현식의 결과에 따라 반복

추후 예약된 keywords:

- abstract
- alignof
- become
- box
- do
- final
- macro
- offsetof
- override
- priv
- proc
- pure
- sizeof

- typeof
- unsized
- virtual
- yield

변수와 가변성

불변 변수 선언

```
let x = 5;
let x = 6;
```

- let으로만 변수 재정의 가능
- 재정의 할 때, 새 변수 x로 기존 변수 x를 show한다고 표현한다.
- 같은 변수명일 뿐, 변수를 재정의하는 것이므로,
  데이터형,데이터값 모두 바꿀 수 있다.
- 일반 변수값 대입 방식으로(x = 6;) 데이터값 교체 불가능

가변 변수 선언

```
let mut x = 5;
x = 6;
```

- 데이터형 유지, 변수의 데이터값만 교체 가능

불변변수/가변변수 차이 예제

```
let spaces = "   ";
let spaces = spaces.len();
```

- 잘 compile 됨
- let으로 재정의하므로 재정의할 데이터형이 문자열이든 숫자든 상관 없다.

```
let mut spaces = "   ";
spaces = spaces.len();
```

- Error 남
- 가변변수로 정의했으므로, 데이터형은 유지된다.
  그런데 문자열 데이터형으로 정의된 가변변수 spaces에 숫자를 넣어 에러가 난다.

상수 선언

```
const MAX_POINTS: u32 = 100_000;
```

- u32로 unsigned 32비트 데이터타입을 선언

데이터형들

스칼라 타입:

- **정수형**
  - Signed → i → **i8, i16, i32, i64**
  - **U**nsigned → u → **u8, u16, u32, u64**
  - **isize, usize** → 로컬 머신의 bit수(32bit or 64bit)에 따라 정해지는 데이터형
  - 접미사로 숫자의 데이터형 표시 가능
    - **1234u8**
  - "_"로 콤마와 같은 시각적 구분 표시 가능
    - **1234_5678** = 1234,5678
  - 진법 표기
    - 10진법: **98_222**
    - 16진법: **0xff**
    - 8진법: **0o77**
    - 2진법: **0b1111_0000**
    - Byte: **b'A'**
- **부동소숫점숫자**
  - **f32** → f32타입으로 정의하려면 let y: f32 = 3.0; 처럼 변수:타입 형태로 타입 명시해줘야함.
  - **f64** → 기본 타입
  - 기본 연산: **+, -, *, /, %**
- **Boolean**

```
fn main() {
    let t = true;
    let f: bool = false; // with explicit type annotation
}
```

- **문자**
  - let c = 'z';

- '문자'는 작은 따옴표,
  "문자열"은 큰 따옴표.

다차원 타입:

- **튜플**
  - let tup: (i32, f64, u8) = (500, 6.4, 1);
    let first_element = x.0;
    let second_element = x.1;
  - 변수명: 튜플데이터형 = 데이터값 의 모양.
    튜플의 원소는 변수.(n-1) 로 불러낸다.
- **배열**
  - let a = [1, 2, 3, 4];
  - 요소 지정: let first = a[0]; → 첫요소가 0번.
  - 없는 요소를 부르면 에러남, let tenth = a[9]; → Error
  - 데이터가 heap보다 stack에 할당된다.
  - 배열은 데이터 구조가 가변적이지 않다.
  - 벡터타입은 데이터 구조의 확장, 축소가 가능하니
    벡터, 배열 중 확실치 않으면 벡터로 선언한다.

함수 동작 원리

메인 함수 및 기타 함수 정의:

```
fn main() {
    println!("Hello, world!");
    another_function();
}
fn another_function() {
    println!("Another function.");
}
```

- fn 함수명() {...}의 함수정의 구문들은 순서는 상관없다.
  즉, 위에서 another_fnction()이 main()보다 위에서 정의되어도 상관없음.
- 프로그램의 실행은 main() 함수 내의 순서대로 진행

매개변수와 전달인자:

```
fn main() {
    another_function(5);
```

```
}
fn another_function(x: i32) {
    println!("The value of x is: {}", x);
}
```

→ another_function(x: i32) → another_function의 매개변수(parameter) x 를 타입 i32로 정의

→ another_function(5) 로 5를 전달인자(argument)로 매개변수 x에 전달하며 함수 호출.

다변수 함수 형태의 매개변수 정의:

```
fn main() {
    another_function(5, 6);
}
fn another_function(x: i32, y: i32) {
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

→ another_function(x: i32, y: i32) {... 형태로 정의하기만 하면 됨.

구문(statement)과 표현식(expression):

- 함수 내에는 구문과 표현식의 라인들로 쓰여진다.
  - 구문(statement): 반환값이 없는 줄, ;로 끝나거나 let과 같은 명령문
    - x = 6; → ;때문에 반환값 없음.
    - let x = 3; → let은 반환값은 없음.
    - let x = (y = 6); → ;없어서 6이 반환된 후 x에 묶임.
    - let x = (y = 6;); → ;으로 반환값이 없어서 Error
    - let x = (let y = 6); → let은 반환값이 없어서 Error
  - 표현식(expression): 반환값이 있는 줄
    - x = 3 → 세미콜론이 없다.
    - let y = {
      let x = 3;
      x + 1
      }; → 반환값은 ;없는 줄의 결과값.

함수 반환값 타입 정의:

```
fn main() {
    let x = plus_one(5);
    println!("The value of x is: {}", x);
```

```
}
fn plus_one(x: i32) -> i32 {
    x + 1
}
```

- fn 함수명(매개변수: 매개변수타입) → 반환값타입 { 함수내용 } 형태

주석

```
// Hello, world.

fn main() {
    let lucky_number = 7; // I’m feeling lucky today.
}
```

제어문

조건문 - if 문:

```
fn main() {
    let number = 6;
    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

- number < 5 표현식이 Boolean 타입 결과값을 반환한다.
- if bool_resut { ... 식으로 true, false를 가지는 boolean 타입 변수를 바로 넣을 수 있다.

if 문을 표현식으로 이용하기:

```
    let number = if condition {
        5
    } else {
```

```
        6
    };
```

- if ... {...} 자체가 반환값이 있는 표현식(expression)이므로 구문(statement) 안에 넣을 수 있다.
- let 안에 넣어서 if와 else의 반환값의 공통된 타입으로 number의 타입이 정의된다.
  if와 else 반환값의 타입이 다르면 number 타입을 결정 못해서 에러가 난다:

```
    let number = if condition {
        5
    } else {
        "six"
    };

error[E0308]: if and else have incompatible types
 --> src/main.rs:4:18
  |
4 |       let number = if condition {
  |  __________________^
5 | |         5
6 | |     } else {
7 | |         "six"
8 | |     };
  | |_____^ expected integral variable, found reference
  |
  = note: expected type `{integer}`
             found type `&str`
```

반복문 - loop:

```
    loop {
        println!("again!");
    }
```

- 무한 반복된다.
- 실행중 ^c 또는 loop내의 Error코드, break문으로 반복이 멈춘다.

반복문 -while:

```
    let mut number = 3;
    while number != 0 {
        println!("{}!", number);
        number = number - 1;
    }
```

- 조건문이 true인 동안만 실행된다.

반복문 - for:

```
let a = [10, 20, 30, 40, 50];
for element in a {
    println!("the value is: {}", element);
}
```

- for문의 a는 다른 표현식으로 적을 수도 있다.
  - (ex) (1..4).rev()
  - (ex) a.rev()

소유권 이해하기

소유권이란?

- 러스트의 핵심 기능
- 보통은 프로그램 실행 중 컴퓨터 사용되지 않는 메모리를 계속해서 찾는 garbage collection을 한다.
- 또는 프로그래머가 직접 메모리를 할당하고 해제하기도 한다.
- 러스트는 소유권이란 개념으로 제 3의 방법을 택하였다.

스택과 힙:

- 스택
  - 접시 쌓기 식으로 메모리를 넣고 뺀다. (last in, first out)
  - 1차원적이라 데이터 처리가 빠르다.
    (데이터를 어디 넣을지 고민할 필요가 없다. 항상 스택 위에 얹는 방식임)
  - 데이터 넣기 (pushing on the stack), 데이터 빼기 (popping off the statck)
  - 접시 크기가 정해져 있어야 쌓을 수 있다.
    (스텍에 들어갈 데이터는 고정된 크기를 가져야 한다.)
- 힙
  - 컴파일시 데이터 크기가 결정되지 않는 데이터는 힙에 저장된다.
  - 데이터를 힙에 넣을 때, 운영체제가 여유 공간을 할당 후 포인터를 건내준다.
    (allocating on the heap, 느린 이유 중 하나)
  - 스택에 포인터를 푸싱하는 것은 할당에 해당되지 않는다.
  - 힙에 저장된 데이터에 접근하는 것은 느리다.
    데이터를 가리키는 포인터를 따라가야 하기 때문.

소유권 규칙

1. 러스트의 각각의 값은 해당값의 *오너*(*owner*)라고 불리우는 변수를 갖고 있다.
2. 한번에 딱 하나의 오너만 존재할 수 있다.
3. 오너가 스코프 밖으로 벗어나는 때, 값은 버려진다(dropped).

변수의 스코프

```
{                    // s는 유효하지 않습니다. 아직 선언이 안됐거든요.
    let s = "hello";   // s는 이 지점부터 유효합니다.
    // s를 가지고 뭔가 합니다.
}                    // 이 스코프는 이제 끝이므로, s는 더이상 유효하지 않습니다.
```

- Scope {...} 안에 변수가 등장한 후 끝날 때까지만 메모리에 상주한다.

String 타입은 힙에 할당된다:

```
let t = "hello";
let s = String::from("hello");
```

- t는 스트링 리터럴로 컴파일시 데이터 크기가 정해진다. → 스택으로?
- s는 스트링 타입을 사용해 사용자입력으로 저장이 가능하다. → 힙으로 할당
- 스트링리터럴 "hello"을 힙에 저장하는 스트링타입으로 변환할 땐, String::from("hello") 로 쓸 수 있다.
- ::는 네임스페이스 연산자.

```
let mut s = String::from("hello");
s.push_str(", world!"); // push_str()은 해당 스트링 리터럴을 스트링에 붙여줍니다.
println!("{}", s); // 이 부분이 `hello, world!`를 출력할 겁니다.
```

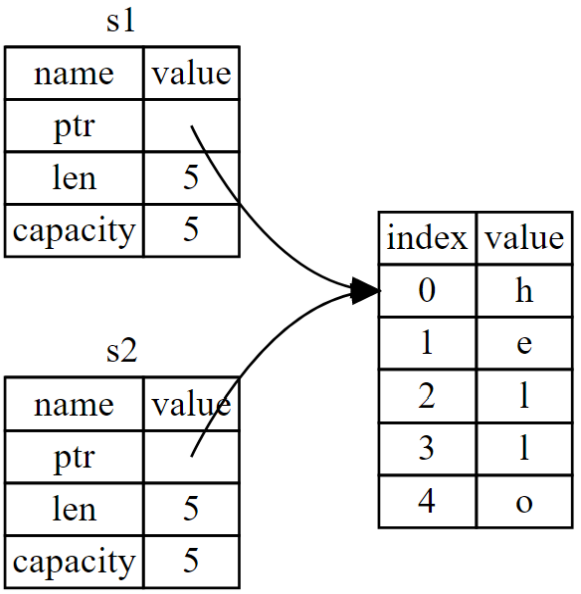
- 스트링리터럴이 아닌 스트링이므로 문자열 변경이 가능했다.

힙에 쓰고 지울 때 메모리 할당 과정:

1. 런타임동안 운영체제에 메모리를 요청해야 한다.
2. String 사용이 끝나면 메모리를 반납해야 한다.
    1. 보통 Garbage collector(GC)가 안 쓰는 메모리 조각을 찾아 지워간다.
    2. GC가 안 지운 게 있으면 메모리 낭비,
       너무 지우면 써야할 데이터가 없어지고,
       두번 지우면 버그다.
    3. 이 문제는 프로그래밍 언어마다 공통된 어려운 문제이다.
    4. Rust에서는 변수가 소속된 스코프 {...}를 벗어나면 자동으로 힙메모리가 반납된다.

변수와 데이터가 메모리에서 다뤄지는 방법:

```
let s1 = String::from("hello");
let s2 = s1;
```



- 스택에 포인터(주소), 데이터길이, 할당된메모리용량이 저장
- 힙에 데이터가 순서대로 저장.
- s2로 s1을 복사시, 스택의 메모리가 복사됨, 힙데이터는 그대로 유지. (얕은 복사)
- Scope를 벗어나면 s1, s2의 힙데이터를 모두 지우려 한다.
  이는 두번 지우기, double free 오류를 만들며 메모리 손상의 원인이 된다.
  - 이 때문에 Rust 는,
    let s2 = s1 에서 s1을 무효화하고, s2가 유효화 시키는 move의 방식을 취한다.
  - 만약 s1을 호출하면 다음과 같은 Error를 볼 수 있다.

```
let s1 = String::from("hello");
let s2 = s1;
println!("{}, world!", s1);

error[E0382]: use of moved value: `s1`
 --> src/main.rs:4:27
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
```

```
  |
  = note: move occurs because `s1` has type `std::string::String`,
which does not implement the `Copy` trait
```

힙데이터도 새로 복사하는 방식, 클론:

```
let s1 = String::from("hello");
let s2 = s1.clone();
println!("s1 = {}, s2 = {}", s1, s2);
```

- let s2 = s1.clone(); 의 구문을 이용하여 힙데이터도 복사한다.
- 이 경우 유호화의 move 방식이 아닌, 같은 내용의 새로운 힙데이터를 만들었기에,
  다시 호출해도 에러가 나지 않는다.

스택-Only data: Copy (스택에만 있는 데이터는 메모리 중복 헤제의 문제가 없다):

```
let x = 5;
let y = x;
println!("x = {}, y = {}", x, y);
```

- 이 경우 스택에서 복사본이 만들어진 것이므로, Error가 나지 않는다.
- 스택으로 Copy가 이루어지는 데이터형:
  - u32와 같은 모든 정수형 타입들
  - true와 false값을 갖는 부울린 타입 bool
  - f64와 같은 모든 부동 소수점 타입들
  - Copy가 가능한 타입만으로 구성된 튜플들. (i32, i32)는 Copy가 되지만, (i32, String)은 안됩니다.

소유권과 함수

```
fn main() {
    let s = String::from("hello");  // s가 스코프 안으로 들어왔습니다.
    takes_ownership(s);             // s의 값이 함수 안으로 이동했습니다...
                                    // ... 그리고 이제 더이상 유효하지 않습니다.
    let x = 5;                      // x가 스코프 안으로 들어왔습니다.
    makes_copy(x);                  // x가 함수 안으로 이동했습니다만,
                                    // i32는 Copy가 되므로, x를 이후에 계속
                                    // 사용해도 됩니다.
} // 여기서 x는 스코프 밖으로 나가고, s도 그 후 나갑니다. 하지만 s는 이미 이동되었으므로,
  // 별다른 일이 발생하지 않습니다.
fn takes_ownership(some_string: String) { // some_string이 스코프 안으로 들어왔습니다.
    println!("{}", some_string);
```

```
} // 여기서 some_string이 스코프 밖으로 벗어났고 `drop`이 호출됩니다. 메모리는
  // 해제되었습니다.
fn makes_copy(some_integer: i32) { // some_integer이 스코프 안으로 들어왔습니다.
    println!("{}", some_integer);
} // 여기서 some_integer가 스코프 밖으로 벗어났습니다. 별다른 일은 발생하지 않습니다.
```

- 힙메모리를 쓰는 변수를 한번 다른 함수로 보내면(소유권 이동)
  그 함수에서 사용한 후 그 함수이 scope를 벗어나는 순간 메모리 해제가 이루어져
  그 변수는 무효화된다.
- 스택메모리를 쓰는 변수는 함수로 보낸 뒤 그 함수의 scope를 벗어나도 아무런 일이 일어나지 않는다.

반환값과 스코프

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership은 반환값을 s1에게
                                        // 이동시킵니다.
    let s2 = String::from("hello");     // s2가 스코프 안에 들어왔습니다.
    let s3 = takes_and_gives_back(s2);  // s2는 takes_and_gives_back 안으로
                                        // 이동되었고, 이 함수가 반환값을 s3으로도
                                        // 이동시켰습니다.
} // 여기서 s3는 스코프 밖으로 벗어났으며 drop이 호출됩니다. s2는 스코프 밖으로
  // 벗어났지만 이동되었으므로 아무 일도 일어나지 않습니다. s1은 스코프 밖으로
  // 벗어나서 drop이 호출됩니다.
fn gives_ownership() -> String {             // gives_ownership 함수가 반환 값을
                                             // 호출한 쪽으로 이동시킵니다.
    let some_string = String::from("hello"); // some_string이 스코프 안에 들어왔습니다.
    some_string                              // some_string이 반환되고, 호출한 쪽의
                                             // 함수로 이동됩니다.
}
// takes_and_gives_back 함수는 String을 하나 받아서 다른 하나를 반환합니다.
fn takes_and_gives_back(a_string: String) -> String { // a_string이 스코프
                                                      // 안으로 들어왔습니다.
    a_string  // a_string은 반환되고, 호출한 쪽의 함수로 이동됩니다.
}
```

- 모든 함수가 소유권을 가졌다가 반납하는 것은 번거롭다.
- 넘겨준 변수를 함수 호출 후에도 다시 쓰려면 그 변수값을 함수값으로 받는 번거러운 과정(아래 예시)을 거쳐야 한다.
- 이를 해결해줄 방법이 "참조자(references)"이다.

```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of '{}' is {}.", s2, len);
}
fn calculate_length(s: String) -> (String, usize) {
```

```
    let length = s.len(); // len()함수는 문자열의 길이를 반환합니다.
    (s, length)
}
```

참조자(References)와 빌림(Borrowing)

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

- &s1 : 참조할 변수의 주소
- &String : 넘겨받은 주소로 참조할 데이터형
- s1을 넘겨서 소유권까지 넘겨 메모리 해제를 하게 만들지 않으려면,
  &s1의 형태로 s1의 주소를 데이터로 하는 변수를 넘겨서 그 변수의 주소데이터를 참조하여
  힙데이터를 사용하도록 하면 된다.
  그런 이유로 &s1을 참조자라고 한다.
- fn calculate_lenth(s: &String) 에서 받는 변수는 그 주소가 가리키는 데이터형이므로
  &String으로 표현한다. [&(주소가가리키는)String(스트링형자료)]
- 소유권을 넘겨받은 것이 아니므로, Scope를 벗어나도 메모리해제는 하지 않는다.
- 참조만 할 뿐이므로 s.push_str(", world"); 형태로 변형하려 하면 Error가 난다.

가변 참조자(Mutable References)로 참조한 데이터 바꾸기

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}
fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

- let mut 변수명.. 으로 가변 변수를 정의 후 &mut 변수명으로 참조시키면
  다른 함수에서 데이터를 바꿀 수 있다.

- 스코프내에 가변참조자는 딱 한번만 가능하다.
  let r1 = &mut s;
  let r2 = &mut s; → Error
- 다른 스코프에서는 다시 만들 수 있다.
  let mut s = String::from("hello");
  {
      let r1 = &mut s;
  } // 여기서 r1은 스코프 밖으로 벗어났으므로, 우리는 아무 문제 없이 새로운 참조자를 만들 수 있습니다.
  let r2 = &mut s;
- 불변 변수를 가변 참조자로 쓸 수 없다.
  let mut s = String::from("hello");
  let r1 = &s; // 문제 없음
  let r2 = &s; // 문제 없음
  let r3 = &mut s; // 큰 문제
- 위와 같은 규칙은 데이터 레이스(Data race)를 막기 위함이다.
  데에티 레이스가 발생하는 특정 조건은 다음과 같다.
  - 두 개 이상의 포인터가 동시에 같은 데이터에 접근한다.
  - 그 중 적어도 하나의 포인터가 데이터를 쓴다.
  - 데이터에 접근하는데 동기화를 하는 어떠한 메커니즘도 없다.
- 포인트를 넘기면서 메모리를 해제해버리면 아무 쓸모없는 댕글링 참조자(Dangling References)가 되어버린다.
  Rust는 이런 경우 Error를 준다.
  이를 막으려면, 그냥 참조자가 아닌 데이터를 보내줘라.

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String {
    let s = String::from("hello");
    &s  // 그냥 s로 하면 s의 소유권이 이동되어 아무 문제가 없다.
}
```

슬라이스(Slices)

슬라이스를 이용한 참조자:

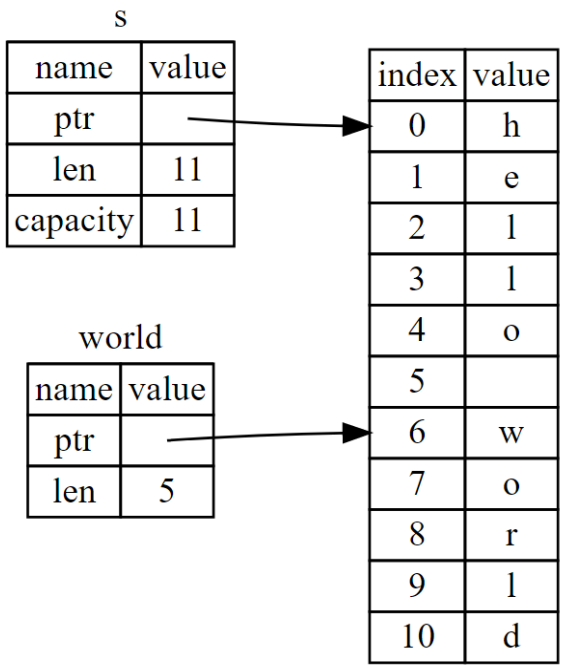
```
fn main() {
    let mut s = String::from("hello world");
```

```
    let word = first_word(&s); // word는 5를 갖게 될 것입니다.
    s.clear(); // 이 코드는 String을 비워서 ""로 만들게 됩니다.
    // word는 여기서 여전히 5를 갖고 있지만, 5라는 값을 의미있게 쓸 수 있는 스트링은 이제 없습니다.
    // word는 이제 완전 유효하지 않습니다! 이게 문제....
}

fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();  // s 문자열을 바이트(문자)별로 나눠배열로 저장
    for (i, &item) in bytes.iter().enumerate() {  // iter로 반복시키고, enumerate로 튜플만든다
        if item == b' ' {  // 공백인 바이트이면....
            return i;   // 여기선 5를 반환
        }
    }
    s.len()
}
```

- 참조자는 하나의 문자열 원소에 대응한 주소를 보낸다.
- 참조할 범위를 참조자형태로 쓸 수 있다. &s[0..5] 형식으로.

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```




```
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s); // word는 스트링 시작위치와 슬라이스 요소 개수를 받는다.
```

```
    s.clear(); // Error! 불변 참조자를 만들어서, clear()를 실행하기 위한 가변참조자를 만들
수 없다.
}
fn first_word(s: &String) -> &str {  // string slice의 타입은 str로 쓴다
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i]; 주소와 범위를 반환
        }
    }
    &s[..]
}
```

동일 표현들

- &s[0..2] = &s[..2]
- &s[3..len] = &s[3..] (when len=s.len())
- &s[0..len] = &s[..]

기타 슬라이스들

```
let s = "Hello, world!"; // 스트링 리터럴인 s의 타입은 &str이다.
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

연관된 데이터들을 구조체로 다루기

구조체 정의하고 초기화하기

사용자 계정 정보 저장 user 구조체 정의:

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

- struct 구조체명 { 필드: 타입, 필드:타입, ... ,} 의 형태로 정의
- 필드명이 부여된 다양한 데이터타입의 배열구조

- 필드의 순서는 상관없다.
- 위에서 보듯이 &str 문자 슬라이스 타입 대신에 String 타입을 사용해서 구조체가 데이터 소유권을 가지며, 구조체가 유효한 동안 그 데이터를 계속 소유한다.

구조체 User의 인스턴스 생성

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

- 필드: String::from("데이터")
  필드: 데이트값
  형태로 인스턴스 생성

특정 필드 변경:

```
 let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
user1.email = String::from("anotheremail@example.com");
```

- 구조체명.필드명 = 바꿀데이터 방식으로 바꿈
- 인스턴스를 바꾸려면 구조체가 mutable로 define 되어야 함

필드 값을 입력 받아 인스턴스를 반환하는 함수

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

- 매개변수와 필드명을 같게 하는 것이 좋은데, 반복이 번거롭다면, 필드명 뒤에 매개변수명을 생략한다. (다음 예제 참조)

```
fn build_user(email: String, username: String) -> User {
```

```
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

다른 구조체의 필드의 데이터를 활용해서 인스턴스 생성

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    active: user1.active,
    sign_in_count: user1.sign_in_count,
};
```

- 새로 정의한 데이터 이외를 활용하려면, ..구조체 형태를 사용

```
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

필드명 없이, 데이터타입만 ordering한 튜플 구조체

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

구조체를 이용한 예제

메소드 문법

메소드 정의하기

```
#[derive(Debug)]
struct Rectangle {
    length: u32,
```

```
    width: u32,
}
impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }
}
fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };
    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

- 구조체에 추가로 구조체 데이터를 이용하는 함수를 정의하기 = 메소드
- impl Rectangle {...} 로 메소드 추가
- fn area(&self) -> u32 {.. 형태로 매개변수를 구조체 자체를 받음.

더 많은 파라미터를 가지는 메소드 정의하기

```
fn main() {
    let rect1 = Rectangle { length: 50, width: 30 };
    let rect2 = Rectangle { length: 40, width: 10 };
    let rect3 = Rectangle { length: 45, width: 60 };
    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}

struct Rectangle {
    length: u32,
    width: u32,
}
impl Rectangle {
    fn area(&self) -> u32 {
        self.length * self.width
    }
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.length > other.length && self.width > other.width
    }
}
```

- 구조체명.메소드명(추가구조체데이터)
  fn 구조체명(&self, 변수: 추가구조체참조) ...
  의 형태로 쓴다.

연관함수(associated fnctions, 구조체 내의 &self를 매개변수로 갖지 않는 함수)

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle { length: size, width: size }
    }
}

fn main() {
     let sq = Rectangle::square(3);
}
```

- 구조체.메소드()
  구조체::연관함수() 의 형태로 쓴다.
- String::from("문자열") 도 연관함수의 예제.

열거형과 패턴 매칭

열거형 정의

사용자정의데이터타입(variants)을 통한 구조체형식의 데이터구조 (데이터 타입 세트)

```
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"),
};
```

- enum 열거형이름 { 사용자정의데이터타입명, ... }
- 호출할 땐 열거형이름::사용자정의데이터타입명

구조체 없이 열거형에 직접 데이터 넣기

```
enum IpAddr {
    V4(String),
    V6(String),
}
let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

```
enum Message {
    Quit,  // 연관 데이터 없음
    Move { x: i32, y: i32 }, // 익명 구조체 포함
    Write(String), // 하나의 String 포함
    ChangeColor(i32, i32, i32), //  세개의 i32 포함
}

struct QuitMessage; // 유닛 구조체
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // 튜플 구조체
struct ChangeColorMessage(i32, i32, i32); // 튜플 구조체
```

- 위에서 보듯이 열거형으로 여러 구조체들을 하나의 열겨형의 variant로 묶을 수 있다.

열거형의 연관함수 만들기

```
impl Message {
    fn call(&self) {
        // 메소드 내용은 여기 정의할 수 있습니다.
    }
}
let m = Message::Write(String::from("hello"));
m.call();
```

Option 열거형과 Null값보다 좋은 점들

- 러스트는 다른 언어들에서 흔하게 볼 수 있는 null 특성이 없다.
- *Null* 은 값이 없다는 것을 표현하는 하나의 값이다.
- null 을 허용하는 언어에서는, 변수는 항상 두 상태중 하나가 될 수 있다:
  - null 혹은 null 이 아님
- Rust에서는 Null 대신 Option<T>가 있다.

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

- Option:: 을 붙이지 않고 variants인 some(T), None을 바로 사용할 수 있다.
- Some이 아닌 None을 사용하려면, Some의 타입은 알려줘야한다.

```
let some_number = Some(5);
let some_string = Some("a string");
let absent_number: Option<i32> = None;
```

- 그런데 Null보다 Option<T>

match 흐름 제어 연산자

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}
fn value_in_cents(coin: Coin) -> u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

if let을 사용한 간결한 흐름 제어

```
let some_u8_value = Some(0u8);
```

```
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

- 위 코드를 다음과 같이 짧게 쓸 수 있다.

```
if let Some(3) = some_u8_value {
    println!("three");
}
```