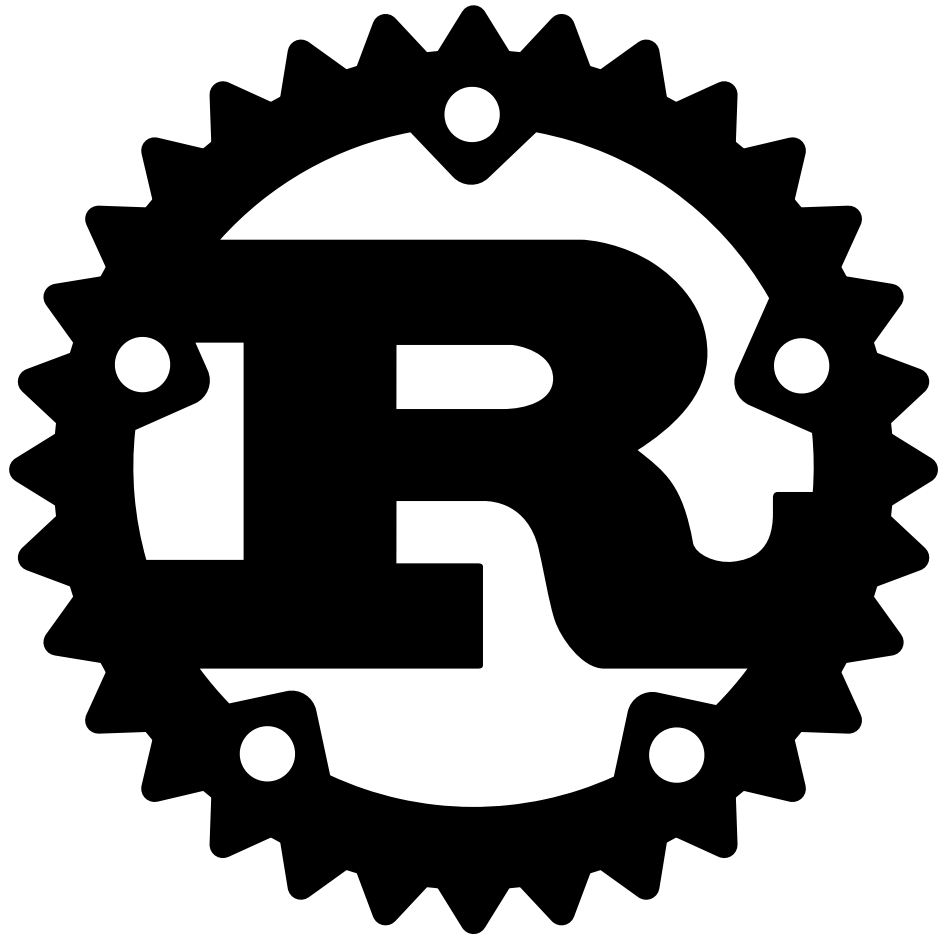
러스트의 멋짐



프로그래밍 언어 러스트

- 모질라 개발자 그레이던 호어(Gradon Hoare)의 사이드 프로젝트로 시작.
- 모질라에서 메인테이닝 (현재 러스트 재단 설립 추진 중)
- 파이어폭스 Servo, 구글 Fuchsia, 삼성, 애플, MS, 페이스북, AWS, Cloudflare, Discord 등.

버전이력

- 2012년 1월 - v0.1 릴리즈
- 2015년 5월 - v1.0 릴리즈
- 2018년 12월 - v1.31.0 (Rust 2018) 릴리즈
- 2020년 11월 - v1.48.0 릴리즈

가장 사랑받는 언어

Stack Overflow 2020 Developer Survey "Most Loved, Dreaded, and Wanted Languages"

빠르게 성장하는 언어

CHANGE IN PROGRAMMING LANGUAGE USE, 2018–2019

01	Dart	532%
02	Rust	235%
03	HCL	213%
04	Kotlin	182%
05	TypeScript	161%
06	PowerShell	154%
07	Apex	154%
08	Python	151%
09	Assembly	149%
10	Go	147%

Fastest growing languages

With Flutter in our trending repositories, it's not surprising that Dart gained contributors this year. We also saw trends toward statically typed languages focused on type safety and interoperability: the Rust, Kotlin, and TypeScript communities are still growing fast.*

GitHub The State of the Octoverse "Fastest growing languages"

러스트의 목표

- 안전성
- 동시성
- 실용성

러스트의 목표

- 비용없는 추상화
- 안전한 쓰레드 관리
- 작은 런타임
- 빠른 성능
- C/C++ 대체
- ...
- **GC없는 메모리 관리** ⭐

목차

문법 훑어보기

Hello, world!

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

```
$ cargo run
  Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.50s
      Running `target/debug/playground`
Hello, world!
```

변수

```
fn main() {
    let x: i8 = 10;
    let y: i8 = 20;

    println!("{}", x + y); // "30"
}
```

- `let` 키워드를 이용해 변수를 선언한다.
- 기본적으로 immutable.
- 타입 추론이 되기 때문에 타입은 생략 가능하다.

```rust
fn main() {
    let mut x: i8 = 10;
    x = x + 20; // 30

    println!("{}", x); // "30"
}
```

- mutable 변수를 만드려면 `mut` 키워드를 사용한다.

타입

integer

```
i8, i16, i32, i64, i128
```

unsigned integer

```
u8, u16, u32, u64, u128
```

floating-point

```
f32, f64
```

문자열

- `str` (`[u8]`) - 고정된 길이의 불변 문자열 타입

```
let x: str = "Hello, world";
```

- `String` (`Vec<u8>`) - 가변적인 문자열 타입 (힙 메모리에 저장)

```
let x: String = String::from("Hello, world");
```

- 다양한 타입으로 문자열 표현 가능

튜플

```
let x: (i32, f64) = (10, 3.14);
```

배열

```
let x: [i8] = [1, 2, 3];
```

벡터

```
let x: Vec<i8> = vec![1, 2, 3];
```

컨트롤 플로우

`if`

```
let result = if number < 5 {
  ...
} else {
  ...
}
```

`loop`

```
let result = loop {
  ...
}
```

- 그 외 `while`, `for ... in`, `match` 등.

함수

```
fn main() {
    let x = 1;
    let result = plus_one(x);
    println!("{}", result); // 2
}

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

`null` 대신 `Option`

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

- null pointer가 없음.
- `T` 를 꺼낼 때 반드시 null check를 거친다.

```rust
fn main() {
    let one = Some(1);
    let two = plus_one(one); // 2
    let zero = plus_one(None); // 0
}

fn plus_one(x: Option<i32>) -> i32 {
    match x {
        None => 0,
        Some(i) => i + 1,
    }
}
```

- 패턴 매칭 표현 `match` 를 이용해 null check를 수행한다.

`try ... catch` 대신 `Result`

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

- `try ... catch` 구문이 없음.
- 복구 가능한(recoverable) 에러를 처리하기 위한 타입.

```rust
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

- `Result` 타입을 반환하는 함수의 반환 값를 사용할 때 에러 처리가 강제된다.
- `panic` 은 프로그램을 종료해야 할 정도로 심각한 에러일 때 사용한다. (`throw`)

GC없는 메모리 관리: 오너십 (Ownership)

메모리 관리 문제

- 개발자가 직접 메모리를 관리하면 번거롭고, 실수할 위험이 크다.
- GC를 사용하면 프로그램 성능이 저하된다.

➡️ 그래서 오너십

오너십은 값에 대한 변수의 소유권

- 각 값은 오너라고 불리는 변수를 갖는다.
- 한 번에 하나의 오너만 가질 수 있다.
- 오너가 스코프를 벗어나면 값이 버려진다.
  (자동으로 `drop` 메서드가 호출되어 메모리에서 해제된다.)

```
{
    let s = String::from("hello");
}                                       // 여기서 스코프가 끝났고, 블록 밖에서
                                        // 변수 `s`는 더 이상 유효하지 않다.
```

값을 다른 변수에 할당할 때

```
fn main() {
    let x = 5;
    let y = x;

    println!("x: {}, y: {}", x, y); // "x: 5, y: 5"
}
```

- 스택 메모리에 올라가는 대부분의 원시 타입 값은 **복사(copy)** 된다.
- integer, boolean, floating point, character 등.

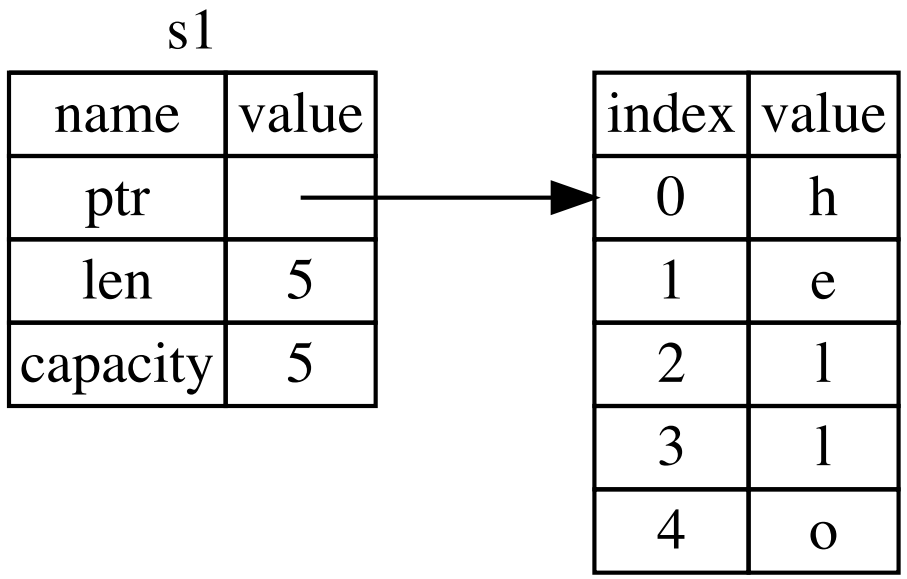
```rust
fn main() {
    let s1 = String::from("Hello, world!");
    let s2 = s1;

    println!("s2: {}", s2); // "Hello, world!"
    println!("s1: {}", s1); // ?
}
```

- 하지만 힙 메모리에 올라가는 `String` 타입의 경우엔...

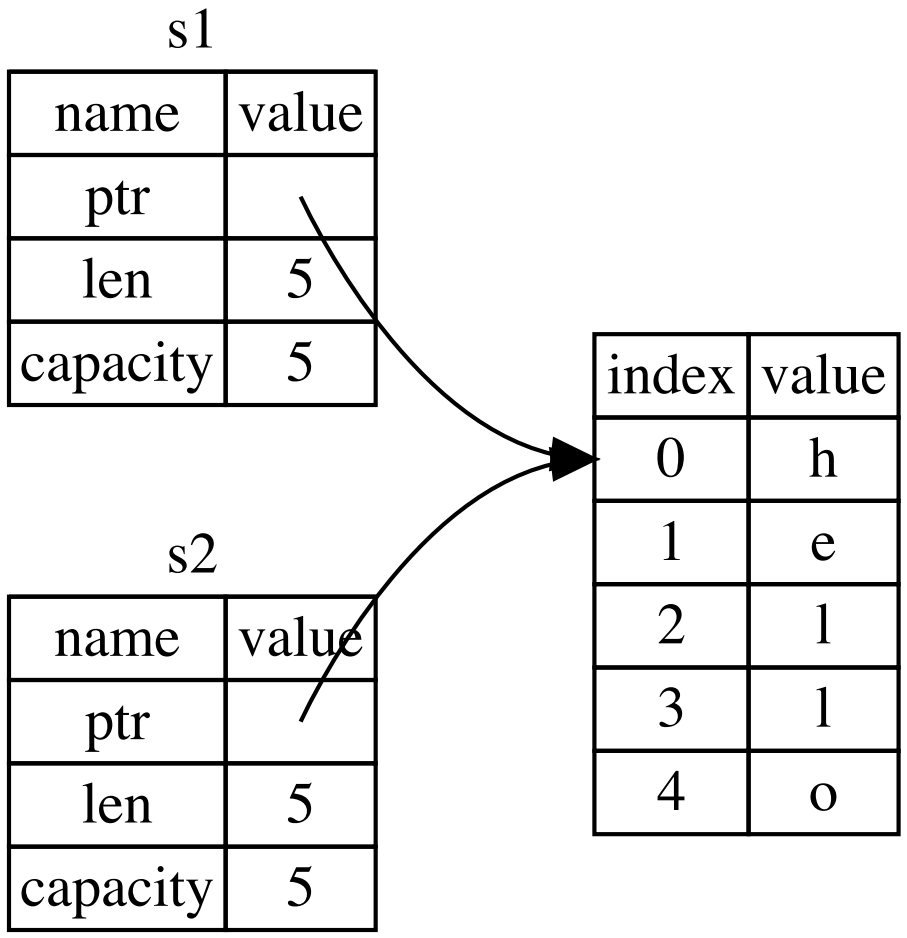
```
error[E0382]: borrow of moved value: `s1`
  |
2 |     let s1 = String::from("Hello, world!");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
...
6 |     println!("s1: {}", s1);
  |                        ^^ value borrowed here after move
```

- 값의 오너십이 변수 `s1` 에서 `s2` 로 **이동(move)** 했기 때문에 컴파일 에러가 발생한다.

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



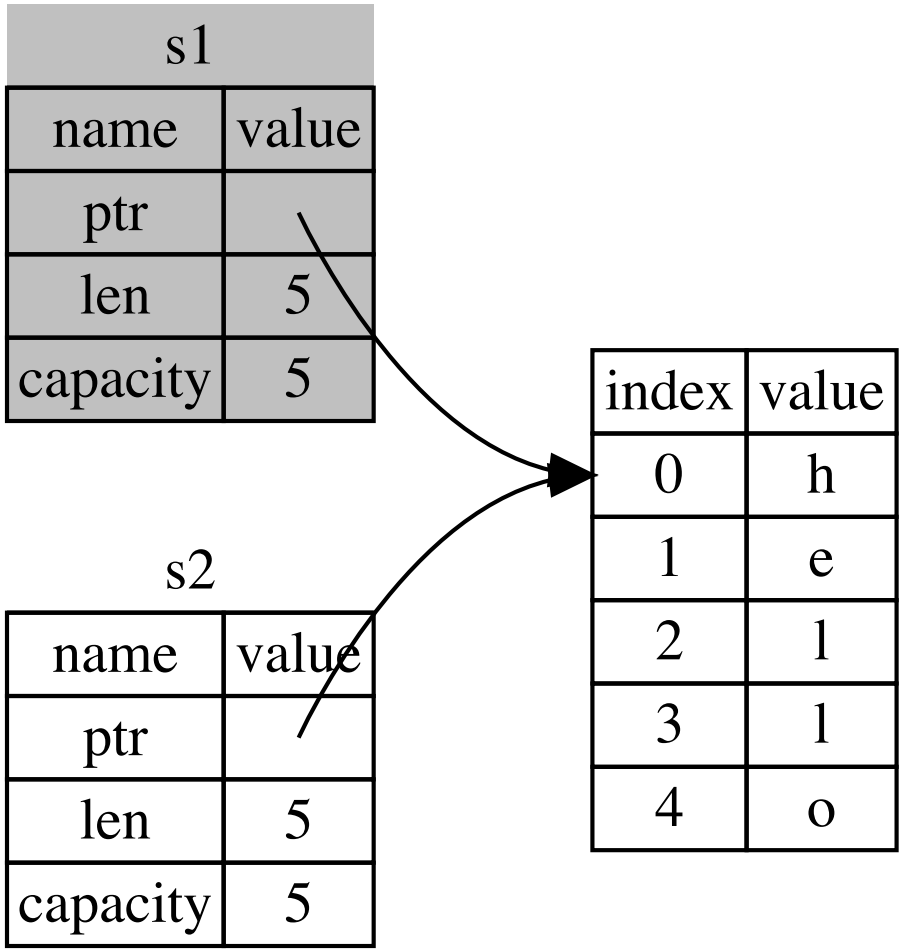
```
let s1 = String::from("hello");
let s2 = s1;
```



1. `s1` 과 `s2` 가 같은 메모리 주소를 가리킨다.
2. 둘 중 하나가 스코프를 벗어났을 때 메모리가 한 번 해제된다. (drop)
3. 그 뒤에 다른 하나가 스코프를 벗어났을 때 같은 메모리를 또 해제하게 된다.
   ⚠️ **불필요한 해제, 메모리 변형 등 보안 취약점 문제**

- `s2` 에 `s1` 을 바인딩할 때 기존 변수 `s1` 을 무효화하면,
- `s2` 가 스코프를 벗어날 때 `s1` 을 신경쓰지 않고 메모리를 한 번만 해제할 수 있다.

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



오너십의 이동

> "`s1`의 오너십이 `s2`로 이동했다"

- 힙 메모리에 올라가는 타입들은 값이 '복사'되는 대신 오너십이 '이동'한다.
- 복사가 불가능한 데이터에 대해 깊은 복사를 하려면 `clone` 메서드를 사용한다.

```rust
fn main() {
    let s1 = String::from("hello");

    let len = get_length(s1); // 여기서 `s1`의 오너십이 `get_length`로 이동한다.

    println!("{}", s1); // error[E0382]: borrow of moved value: `s1`
}

fn get_length(s: String) -> usize {
    s.len()
}
```

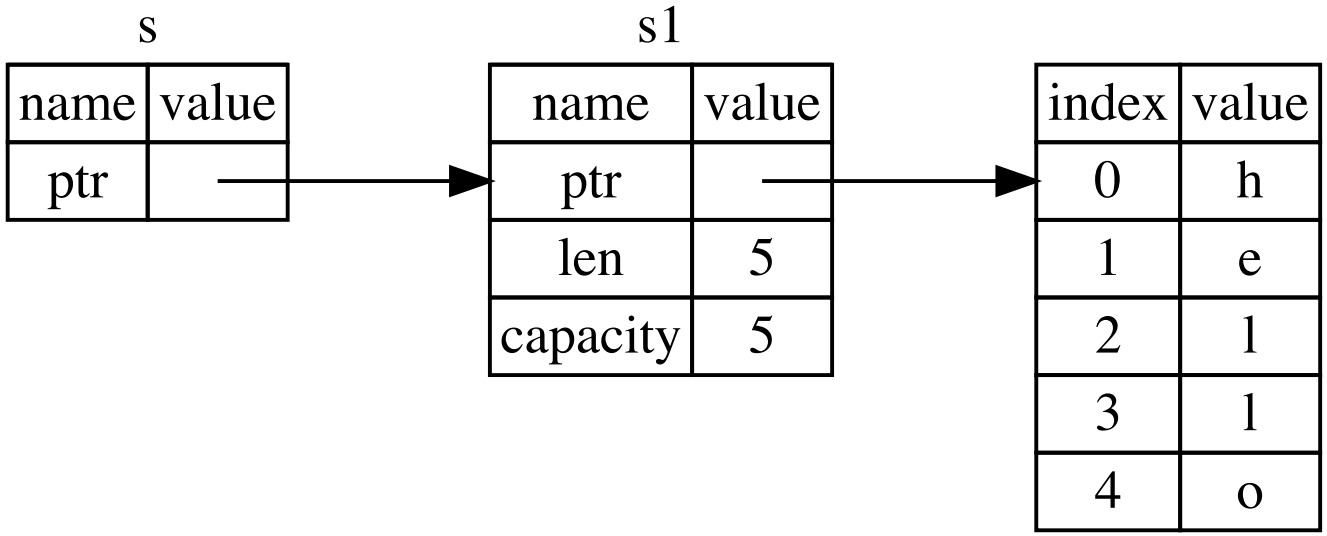
- 함수 인자로 값을 전달할 때도 오너십이 이동한다.
- 오너십 방식을 통해 컴파일 타임에 메모리 안전성을 검증할 수 있다.

참조와 빌림 (References & Borrowing)

오너십을 이동시키고 싶지 않을 때

- 참조(reference)를 넘겨주면 된다.
- 이를 빌림(borrowing)이라고 한다.
    - 소유권을 넘겨주는 것이 아니라 빌려주는 것.

```rust
fn main() {
    let s1 = String::from("hello");
    let len = get_length(&s1); // `s1`의 오너십을 주는 대신 참조만 넘겨준다.
                               // `s1`은 여전히 유효하다.
    println!("{:?}", s1.as_ptr()); // "0x5581762b0a40"
    println!("{}", s1); // "hello"
}

fn get_length(s: &String) -> usize {
    println!("{:?}", s.as_ptr()); // "0x5581762b0a40"
    s.len()
}
```



참조를 이용해 값을 바꾸고 싶다면

```
fn main() {
    let mut s1 = String::from("hello");
    append_world(&mut s1); // 가변 참조를 넘겨준다.
    println!("{}", s1); // "hello, world"
}

fn append_world(s: &mut String) {
    s.push_str(", world"); // 인자로 받은 문자열 가변 참조에 ", world"를 덧붙인다.
}
```

- `&mut` 키워드로 가변 참조(mutable reference)를 넘기면 값을 변경할 수 있다.
- 이를 가변 빌림(mutable borrowing)이라고 한다.

⚠️ 한 스코프 안에서 가변 참조는 한 번만

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

let r1 = &mut s;
let r2 = &mut s;

r1.push_str(", world"); // ?
```

- `r1`에 가변 참조를 빌려준 다음, 바로 `r2`에게도 참조를 빌려줬다.
- 개발자는 `r1`이 `"hello, world"`가 되길 기대하지만...

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     r1.push_str(", world");
  |     -- first borrow later used here
```

- 러스트 컴파일러가 친절히 안 된다고 알려준다.
- 이 제약 덕분에 경쟁 상태(race condition)를 사전에 방지할 수 있다.
  - 두 개 이상의 포인터가 동시에 같은 데이터에 접근하며,
  - 최소 하나의 포인터가 데이터 변경을 시도하고,
  - 데이터를 동기화하는 매커니즘이 없는 경우.

변수의 일생: 라이프타임 (Lifetimes)

라이프타임

- 모든 참조자는 라이프타임(lifetime)을 갖는다.
- 라이프타임은 참조자가 유효한 스코프.
- 러스트의 가장 독특한 기능.

```
{
    let r;           // -------+-- 'a
                     //        |
    {                //        |
        let x = 5;   // -+-----+-- 'b
        r = &x;      //  |     |
    }                // -+     |
                     //        |
    println!("r: {}", r); //  |
                     //        |
                     // -------+
}
```

- 변수 `r` 의 라이프타임 `'a` . (바깥 블록 전체에서 유효)
- 변수 `x` 의 라이프타임 `'b` . (안쪽 블록에서만 유효)

```
error: `x` does not live long enough
   |
6  |          r = &x;
   |              - borrow occurs here
7  |      }
   |      ^ `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here
```

- 변수 `r` 이 `x` 의 참조를 빌렸지만,
- 변수 `r` 을 참조하는 시점에 `x` 는 이미 유효하지 않기 때문에 컴파일 에러.
  > "변수 `x` 가 충분히 오래 살지 못한다"
- 이미 drop된 데이터를 참조하는 댕글링 참조자(dangling reference)를 방지할 수 있다.

함수 시그니처의 라이프타임

```
fn do_something<'a>(x: &'a str) -> &'a str {
    ...
}
```

- 사실 함수의 파라미터와 반환 값의 라이프타임이 같다는 것이 생략된 것.
- 명시할 수 있지만, 굳이 명시하지 않아도 추론된다.
- 라이프타임 `'a` 는 특정 값이 아니라 라벨.

라이프타임을 명시해야 하는 경우

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- 서로 다른 파라미터 `x`, `y`를 받는 함수.
- 파라미터로 받은 참조자 중에서 문자열 길이가 더 긴 참조자를 반환한다.

```
error[E0106]: missing lifetime specifier
  |
1 | fn longest(x: &str, y: &str) -> &str {
  |               ----     ----     ^ expected named lifetime parameter
  |
  = help: this function's return type contains a borrowed value,
    but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
  |
1 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
  |            ^^^^    ^^^^^^^     ^^^^^^^     ^^^
```

- 반환하는 참조자 `&str`이 `x`를 참조하는지, `y`를 참조하는지 결정할 수 없어서 컴파일 에러.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- `x`, `y`가 모두 적어도 라이프타임 `'a`만큼 살아있다는 것을 명시해줘야 한다.

```
fn main() {
    let s1 = String::from("long string is long");      // ------+-- 'x
                                                       //       |
    {                                                  //       |
        let s2 = String::from("xyz");                  // -+----+-- 'y
        let result = longest(&s1, &s2);                //  |    |
        println!("The longest string is {}", result); // -+    |
    }                                                  //       |
}                                                      // ------+
```

- 파라미터로 넘긴 `s1` 와 `s2` 의 참조자는 둘 다 적어도 `'y` 만큼 살 수 있다.
- 따라서 함수 입장에선 두 참조자의 라이프타임을 동일한 것으로 취급할 수 있다.

정리

- 러스트는 **오너십** 방식으로 메모리를 관리한다.
- 오너가 스코프를 벗어나면 drop되어 메모리에서 해제된다.
- 스택 메모리 타입은 값이 **복사**되고, 힙 메모리 타입은 오너십이 **이동**한다.
- 모든 참조자는 자신이 유효한 스코프인 **라이프타임**을 갖는다.

➡️ 컴파일 타임에 메모리 안전성을 보장한다!

자잘하게 멋진 것들

편리한 툴체인

```
$ rustup component add rustfmt clippy # rustfmt, clippy 추가
$ cargo fmt # formatter
$ cargo clippy # linter
```

- 쉬운 설치, 쉬운 실행.
- 다양한 에디터와 플러그인에서 범용적으로 사용할 수 있다.

빌트인 테스트 프레임워크

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_adds_two() {
        assert_eq!(4, add_two(2));
    }
}
```

```
$ cargo test
   Compiling adder v0.1.0 (file:///projects/adder)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running target/debug/deps/adder-92948b65e88960b4

running 1 test
test tests::it_adds_two ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

빌트인 문서화 도구

```
/// Adds one to the number given.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = my_crate::add_one(arg);
///
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

```
$ cargo doc
```

- 문서화 주석의 코드를 테스트해 유효한지 체크해준다.

my_crate

Functions

add_one

Crates

my_crate

Click or press ‘S’ to search, ‘?’ for more options…

Function my_crate::add_one [–][src]

```
pub fn add_one(x: i32) -> i32
```

[–] Adds one to the number given.

Examples

```
let arg = 5;
let answer = my_crate::add_one(arg);

assert_eq!(6, answer);
```

WASM

```rust
#[no_mangle]
pub fn add(a: i32, b: i32) -> i32 {
  return a + b
}
```

```javascript
import { add } from './add.rs'
console.log(add(2, 3)) // 5
```

- WASM을 통해 러스트 코드를 자바스크립트에서 그대로 사용할 수 있다.
- 덕분에 웹 프론트엔드에도 러스트를 사용할 수 있다.

참고자료

- Steve Klabnik, Carol Nichols, "The Rust Programming Language".
- ingeeKim, "Rust 는 처음이죠? 도전해봅시다", Mozilla 웹 기술 블로그, 2015.
- MDN web docs “Compiling from Rust to WebAssembly”, 2019.
- "Rust by Example".