

Konkuk University GDSC + EDGE 스터디

Week 6: Closures

Chris Ohk

utilForever@gmail.com

- 변수 캡처하기
 - 빌리는 클로저
 - 훔치는 클로저
- 함수와 클로저 타입
- 클로저의 성능
- 클로저와 안전성
 - 죽이는 클로저
 - FnOnce / FnMut
 - 클로저를 위한 Copy와 Clone
- 콜백
- 효율적인 클로저 사용법

들어가며

- 구조체 City를 정렬하고 싶다. 무엇을 기준으로 정렬해야 할까?
 - 컴파일해보면 City가 std::cmp::Ord를 구현하지 않았다고 오류를 출력한다.

```
● ● ●

struct City {
    name: String,
    population: i64,
    country: String,
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort(); // ???
}
```

들어가며

- 이 때는 정렬 순서를 지정해 주면 된다.
 - 함수 `city_population_descending`은 City 레코드를 받아서 정렬의 기준으로 쓸 필드를 키(Key)로 추출한다.
(음수를 반환하는 이유는 내림차순으로 정렬하고 싶기 때문이다.)
 - `sort_by_key` 메소드는 이 키 함수를 매개 변수로 받는다.

```
● ● ●

struct City {
    name: String,
    population: i64,
    country: String,
}

fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(city_population_descending);
}
```

- 하지만 익명의 함수 표현식인 **클로저(Closure)**를 사용하면 코드를 간결하게 만들 수 있다.
 - 이 클로저는 인수 city를 받아서 -city.population을 반환한다.
Rust는 이 클로저의 쓰임새에서 인수 타입과 리턴 타입을 추론한다.

```
● ● ●

struct City {
    name: String,
    population: i64,
    country: String,
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(|city| -city.population);
}
```

- 표준 라이브러리에서 클로저를 받는 예는 다음과 같다.
 - 순차 데이터를 다루기 위한 `map`과 `filter`와 같은 `Iterator` 메소드
 - 새 시스템 스레드를 시작하는 `thread::spawn`과 같은 스레드 API
 - `HashMap` 항목의 `or_insert_with` 메소드와 같이 조건부로 기본값을 계산해야 하는 일부 메소드

변수 캡처하기

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 클로저는 바깥쪽 함수에 속한 데이터를 사용할 수 있다.



```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {  
    cities.sort_by_key(|city| -city.get_statistic(stat));  
}
```

- 예제에서 클로저는 `sort_by_statistic`이 소유한 `stat`을 사용한다.
- 이를 가리켜 클로저가 `stat`을 **캡처(Capture)**한다고 말한다. 그러나 Rust에서는 이 기능에 조건이 붙는다.

변수 캡처하기

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 클로저를 갖춘 대부분의 언어에서는 가비지 컬렉션이 중요한 역할을 한다.

```
function startSortingAnimation(cities, stat) {
    function keyfn(city) {
        return city.get_statistic(stat);
    }

    if (pendingSort)
        pendingSort.cancel();

    pendingSort = new SortingAnimation(cities, keyfn);
}
```

- 클로저 keyfn은 새 SortingAnimation 오브젝트에 저장되며, 아마 startSortingAnimation이 복귀한 이후에 호출될 것이다. 함수가 복귀하면 보통은 그의 모든 변수와 인수가 범위를 벗어나 버려진다. 그러나 여기서는 클로저가 stat을 사용하므로 JavaScript 엔진은 이를 어떻게든 유지해 두어야 한다. 대부분은 stat을 힙에 할당하고 가비지 컬렉터가 나중에 이를 회수하는 방법으로 이 문제를 해결한다.
- 하지만 Rust는 가비지 컬렉션이 없는데, 이 문제를 어떻게 해결할까?

- 앞에서 봤던 함수 `sort_by_statistic`을 다시 보자.



```
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {  
    cities.sort_by_key(|city| -city.get_statistic(stat));  
}
```

- Rust가 클로저를 생성할 때 `stat`의 레퍼런스를 자동으로 빌린다.
클로저가 `stat`을 참조하고 있으므로 그의 레퍼런스를 가지고 있어야 하는 건 당연하다.
- 클로저는 빌림과 수명에 관한 규칙을 적용받는다. `stat`의 레퍼런스를 갖고 있으므로 `stat`보다 더 오래 살아있을 수 없다.
여기서는 클로저가 정렬하는 동안에만 쓰이므로 문제될 게 없다.
- Rust는 가비지 컬렉터 대신 수명을 써서 안전성을 보장한다.

훔치는 클로저

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 이번 예제는 좀 더 까다롭다.



```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>> {
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- 앞에서 살펴본 JavaScript 함수와 비슷한 일을 한다.
`thread::spawn`은 주어진 클로저를 새 시스템 스레드에서 호출한다.
- 새 스레드는 호출부와 병렬로 실행되며, 클로저가 복귀하면 종료된다.
- 이번 예제에서도 클로저 `key_fn`은 `stat`의 레퍼런스를 갖는다.

- 컴파일 오류 원인 분석

- 하지만 이번에는 Rust가 레퍼런스의 안전한 사용을 보장할 수 없어서 컴파일 오류가 발생한다.

```
error[E0373]: closure may outlive the current function, but it borrows `stat`,  
          which is owned by the current function  
--> closures_sort_thread.rs:33:18  
33 |     let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };  
   |     ^^^^^^^^^^^^^^^^^^^^^^ ^^^^  
   |     | `stat` is borrowed here  
   |     | may outlive borrowed value `stat`
```

- 컴파일 오류가 발생하는 이유는, `thread::spawn()` 생성한 새 스레드가 앞서 나온 함수의 끝에서 `cities`와 `stat`이 소멸되기 전에 자신의 일을 마치리라고 기대할 수 없기 때문이다.
 - 그렇다면 어떻게 해결해야 할까?

훔치는 클로저

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 해결 방법

- Rust에게 `cities`와 `stat`의 레퍼런스를 빌리지 말고 대신 이 둘을 앞서 나온 클로저 안으로 옮겨달라고 말하는 것이다.
- `move` 키워드는 Rust에게 클로저가 자신이 사용하는 변수를 빌리지 않고 훔친다고 말한다.

```
● ● ●

use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>> {
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

- 클로저에서 데이터를 가져오기 위한 방법 : 이동과 빌림
 - `i32`처럼 값을 복사할 수 있는 타입의 값을 `move`할 경우에는 이동이 아니라 복사가 일어난다. 따라서 `Statistic`이 복사할 수 있는 타입일 경우에는 `move` 클로저를 생성한 이후에도 `stat`을 계속 사용할 수 있다.
 - `Vec<City>`처럼 복사할 수 없는 타입의 값은 실제로 이동된다. 앞의 코드는 `move` 클로저를 통해서 `cities`를 새 스레드로 옮긴다. 따라서 클로저를 생성한 이후에는 `cities`에 접근할 수 없다.
 - `cities`를 다시 사용해야 한다면 복제해서 그 사본을 다른 변수에 담아 두는 간단한 우회책을 사용할 수 있다. 그러면 클로저가 여러 사본 중에서 자신이 참조하고 있는 것만 훔치게 될 것이다.

함수와 클로저 타입

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 함수와 클로저에도 타입이 있다.
 - 다음 함수는 &City 하나를 인수로 받아 i64를 반환하며, 타입은 fn(&City) -> i64다.

```
● ● ●  
  
fn city_population_descending(city: &City) -> i64 {  
    -city.population  
}
```

- 함수는 다른 값들과 똑같이 취급해 사용할 수 있다.
함수를 변수에 저장할 수도 있고, 함수값을 계산할 때도 Rust 문법을 모두 사용할 수 있다.

```
● ● ●  
  
let my_key_fn: fn(&City) -> i64 =  
    if user.prefs.by_population {  
        city_population_descending  
    } else {  
        city_monster_attack_risk_descending  
    };  
  
cities.sort_by_key(my_key_fn);
```

함수와 클로저 타입

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 함수와 클로저에도 타입이 있다.
 - 함수는 다른 함수를 인수로 받을 수 있다.

```
● ● ●

fn count_selected_cities(cities: &Vec<City>, test_fn: fn(&City) -> bool) -> usize {
    let mut count = 0;

    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }

    count
}

fn has_monster_attacks(city: &City) -> bool {
    city.monster_attack_risk > 0.0
}

let n = count_seleccted_cities(&my_cities, has_monster_attacks);
```

함수와 클로저 타입

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 하지만 클로저의 타입은 함수의 타입과 다르다.

```
● ● ●  
let limit = preferences.acceptable_monster_risk();  
let n = count_selected_cities(&my_cities, |city| city.monster_attack_risk > limit);
```

- `count_selected_cities` 함수가 클로저를 지원하기 위해서는 타입 시그니처를 바꿔야 한다.

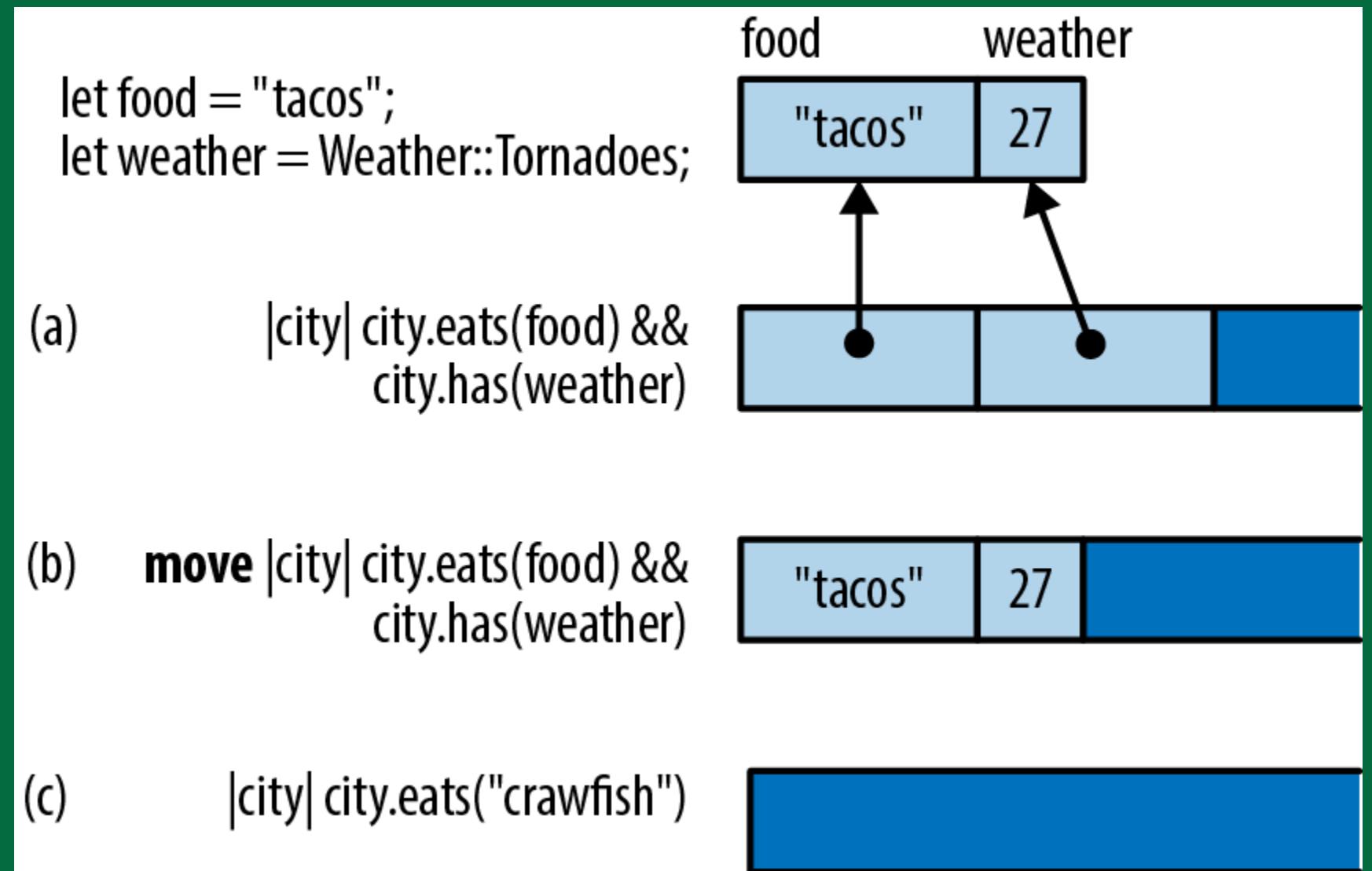
```
● ● ●  
fn count_selected_cities(cities: &Vec<City>, test_fn: F) -> usize  
where F: Fn(&City) -> bool  
{  
    let mut count = 0;  
  
    for city in cities {  
        if test_fn(city) {  
            count += 1;  
        }  
    }  
    count  
}
```

- 원인 분석
 - 클로저는 호출할 수 있지만 fn은 아니다.
클로저 |city| city.monster_attack_risk > limit은 fn 타입이 아니라 자기만의 고유한 타입을 갖는다.
 - 사실, 모든 클로저는 자기만의 고유한 타입을 갖는다. 왜냐하면 클로저가 바깥쪽 범위에서 빌리거나 훔친 값을 데이터로 가질 수도 있고, 또 가지고 있는 변수의 수와 타입의 조합이 전부 제각각일 수 있기 때문이다.
따라서 모든 클로저는 컴파일러로부터 자신의 데이터가 전부 들어갈 만한 크기를 가진 임시 타입을 부여받는다.
 - 똑같은 타입을 가진 클로저란 있을 수 없다. 하지만 모든 클로저는 Fn 트레잇을 구현한다.
앞서 나온 예제에 있는 클로저는 Fn(&City) -> i64를 구현한다.

- Rust 클로저는 다른 언어와 어떻게 다른가
 - 대부분의 언어에서는 클로저가 힙에 할당되고, 동적으로 디스패치되고, 가비지 컬렉션된다. 따라서 클로저를 생성하고, 호출하고, 컬렉션할 때마다 약간의 추가 CPU 시간이 소모된다. 설상가상으로 컴파일러가 함수 호출 비용을 없애고 다양한 최적화를 적용하기 위해 사용하는 핵심 기법인 **인라인(Inline)** 처리의 대상이 되지 못하는 경우가 많다.
 - Rust의 클로저는 가비지 컬렉션되지 않으며 box나 Vec 등의 컨테이너에 집어넣지 않는 이상 힙에 할당되지 않는다. 또한 각 클로저가 서로 다른 타입을 갖기 때문에, Rust 컴파일러가 여러분이 호출하고 있는 클로저의 타입을 인식할 때마다 해당 클로저의 코드를 인라인 처리할 수 있다. 그 덕분에 클로저를 짧고 빠른 주기로 도는 반복문에서 써도 문제가 없다.

클로저 성능

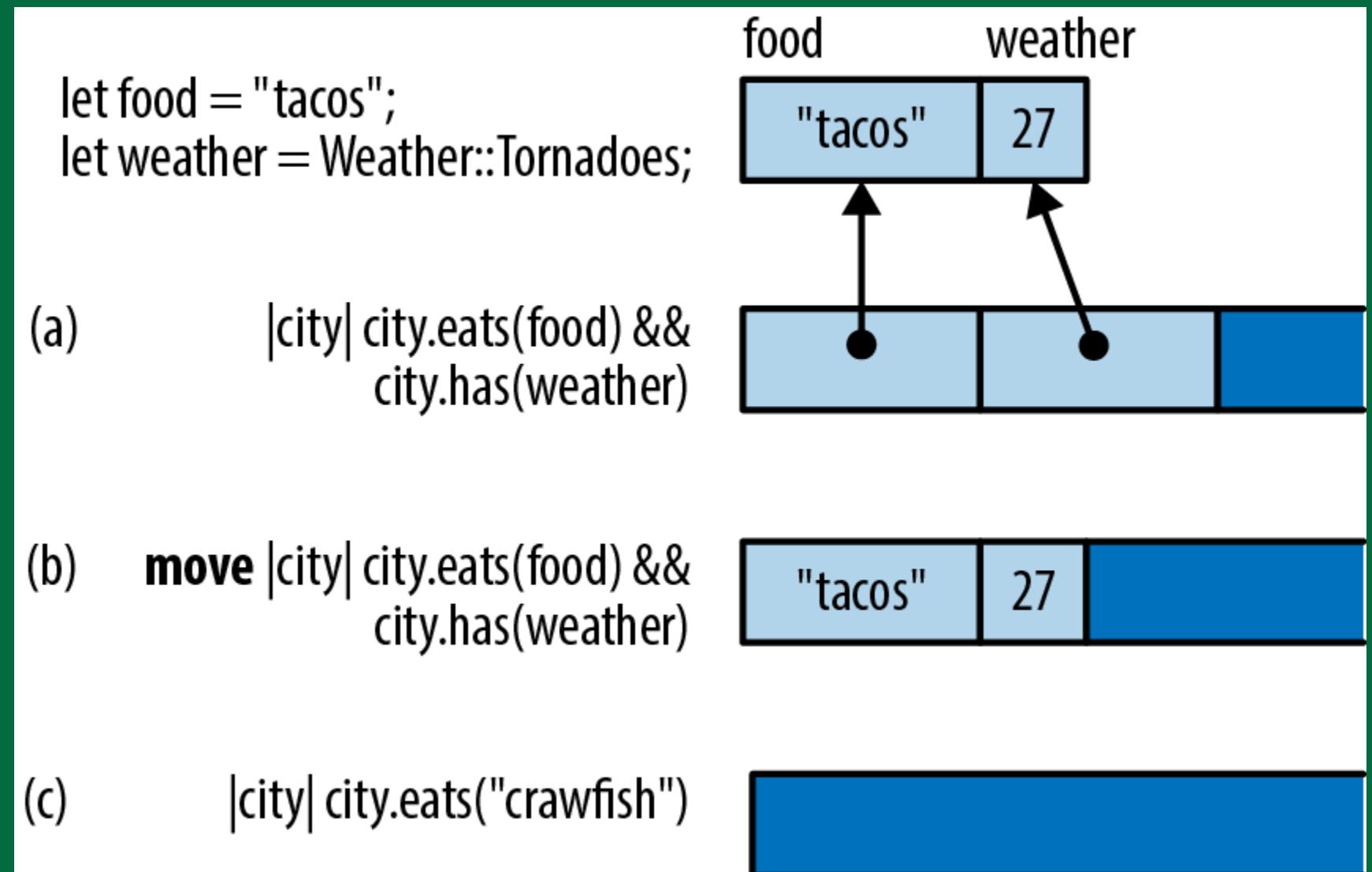
- Rust 클로저는 메모리에 어떻게 배치되는가



- 클로저 (a)는 두 변수를 모두 사용한다. 이 클로저의 메모리 구조는 두 변수의 레퍼런스를 가진 작은 구조체를 담았다.
- 코드를 가리키는 포인터가 포함되어 있지 않다는 점에 주목하자! Rust는 클로저의 타입을 알고 있어서 호출이 들어올 때 어떤 코드를 실행해야 하는지 알기 때문에 그럴 필요가 없다.

클로저 성능

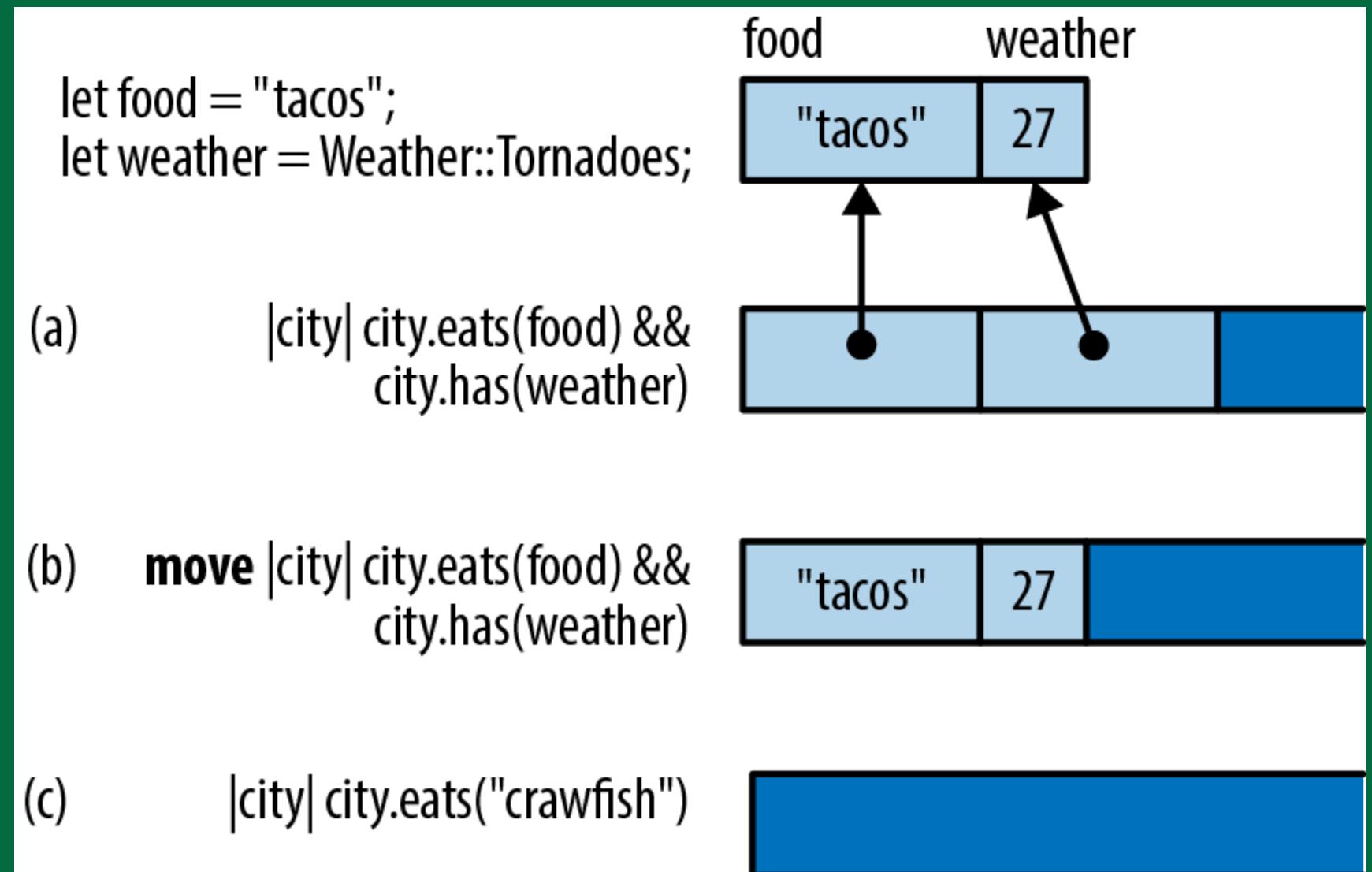
- Rust 클로저는 메모리에 어떻게 배치되는가



- 클로저 (b)는 `move` 클로저라서 레퍼런스 대신 값을 갖는다는 점만 제외하면 똑같다.
- 클로저 (c)는 자기 환경에 있는 변수를 하나도 사용하지 않는다.
구조체가 비어 있으므로 이 클로저는 메모리를 전혀 차지하지 않는다.

클로저 성능

- Rust 클로저는 메모리에 어떻게 배치되는가



- 클로저는 그리 많은 공간을 차지하지 않는다. 심지어 몇 바이트 안되는 공간마저도 실제로 항상 필요로 하는 건 아니다. 컴파일러가 클로저의 모든 호출을 인라인 처리할 수 있는 경우가 많으므로 위 그림에 표시된 작은 구조체들도 최적화될 가능성이 있다.

클로저의 안전성

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- 지금까지는...
 - 클로저가 주변 코드로부터 변수를 빌리거나 옮길 때 Rust가 어떤 식으로 언어의 안전 규칙을 준수하게 만드는지에 대해서 이야기했다.
 - 이번에는 클로저가 캡처된 값을 드롭하거나 수정할 때 벌어지는 일들에 대해서 좀 더 설명한다.

드롭하는 클로저

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- Rust에서 값을 드롭하는 가장 간단한 방법은 drop()을 호출하는 것이다.

```
● ● ●  
let my_str = "hello".to_string();  
let f = || drop(my_str);
```

- f를 호출하면 my_str이 드롭된다. 이제 f를 두 번 호출하면 무슨 일이 벌어질까?
 - f를 처음 호출하면 my_str이 드롭된다. 문자열이 저장되어 있던 메모리가 해제되어 시스템에 반환된다는 의미다.
 - 문제는 f를 다시 호출했을 때도 똑같은 일이 벌어진다는 거다. 이를 바로 중복 해제(Double Free)라고 한다.
- Rust는 위의 클로저가 두 번 호출될 수 없다는 걸 알고 있다.
 - 값이 소모된다는(즉, 이동된다는) 발상은 러스트의 핵심 개념 중 하나다.
 - 클로저에도 이 개념이 적용된다.

- Rust를 다시 한 번 속여서 String이 두 번 드롭되게 해보자.

```
● ● ●  
  
fn call_twice<F>(closure: F) where F: Fn() {  
    closure();  
    closure();  
}
```

- 이 제네릭 함수에는 Fn() 트레이트를 구현하고 있는 모든 클로저,
즉 아무 인수도 받지 않고 ()을 반환하는 클로저를 전달할 수 있다.
- 그런데 앞서 나온 제네릭 함수에 안전하지 않은 클로저를 전달하면 무슨 일이 벌어질까?

```
● ● ●  
  
let my_str = "hello".to_string();  
let f = || drop(my_str);  
call_twice(f);
```

- 이번에도 이 클로저를 호출하면 `my_str`이 드롭된다.
따라서 두 번 호출하면 중복 해제가 발생한다. 그러나 Rust는 속지 않는다.

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> src\main.rs:11:13
|
11 |     let f = || drop(my_str);
|          ^----- closure is `FnOnce` because it moves the variable `my_str` out of its environment
|          |
|          this closure implements `FnOnce`, not `Fn`
12 |     call_twice(f);
|----- - the requirement to implement `Fn` derives from here
|          |
|          required by a bound introduced by this call
```

- 이 오류 메시지는 Rust가 '죽이는 클로저'를 다루는 법에 대해 자세히 알려준다.
- 죽이는 클로저를 언어 차원에서 아예 금지할 수도 있지만,
경우에 따라서는 뒷정리를 하는 클로저가 유용할 때도 있기 때문에 Rust는 금지하는 대신 용도를 제한하고 있다.
- `f`처럼 값을 드롭하는 클로저는 `Fn`을 가질 수 없다. 말 그대로 `Fn`이 아닌 것이다.
그 대신 덜 강력한 트레이트인 `FnOnce`를 구현하는데, 이는 한번만 호출될 수 있는 클로저의 트레이트이다.

- FnOnce 클로저는 처음 호출될 때 클로저 자체가 소모된다.
Fn과 FnOnce는 다음처럼 정의되어 있다고 보면 된다.

```
● ● ●

trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

- Rust는 closure()를 위에 나와 있는 두 트레이트 메소드 중 하나의 축약 표기로 취급한다.
- Fn 클로저의 경우에는 closure()가 closure.call()로 확장된다.
이 메소드는 self를 레퍼런스로 받으므로 클로저가 이동되지 않는다.
- 그러나 클로저가 처음 호출될 때만 안전한 경우에는 closure()가 closure.call_once()로 확장된다.
이 메서드는 self를 값으로 받으므로 클로저가 소모된다.

- 앞에서는 고의로 drop()를 써서 문제를 일으켰지만, 실제로는 우연히 이런 상황에 빠진다.
어쩌다 한 번쯤은 나도 모르게 값을 소모하는 클로저 코드를 작성할 수도 있다.

```
● ● ●  
  
let dict = produce_glossary();  
let debug_dump_dict = || {  
    for (key, value) in dict {  
        println!("{} - {}", key, value);  
    }  
};
```

- `debug_dump_dict()`를 두 번 이상 호출하면, 다음과 같은 오류 메시지가 표시된다.

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> src\main.rs:11:13
|
11 |     let f = || drop(my_str);
|           ^          ----- closure is `FnOnce` because it moves the variable `my_str` out of its environment
|           |
|           this closure implements `FnOnce`, not `Fn`
12 |     call_twice(f);
|           ----- the requirement to implement `Fn` derives from here
|           |
|           required by a bound introduced by this call
|
17 |     debug_dump_dict();
|           ----- `debug_dump_dict` moved due to this call
18 |     debug_dump_dict();
|           ^^^^^^^^^^^^^ value used here after move
|
note: closure cannot be invoked more than once because it moves the variable `dict` out of its environment
--> src\main.rs:12:29
|
12 |         for (key, value) in dict {
|             ^
|
note: this value implements `FnOnce`, which causes it to be moved when called
--> src\main.rs:17:5
|
17 |     debug_dump_dict();
|           ^^^^^^^^^^^^^
```

- 원인 분석

- 이를 디버깅하려면 클로저가 FnOnce인 이유를 알아내야 한다.
컴파일러는 앞의 코드에서 참조하고 있는 유일한 값인 `dict`를 유력한 용의자로 지목하고 있다.
- 바로 여기에 버그가 있다. `dict`를 직접 반복 처리하는 바람에 값이 소모되고 있는 것이다.
따라서 그냥 `dict` 대신 `&dict`를 반복 처리하게 고쳐서 레퍼런스로 값을 접근하도록 만들어야 한다.

```
● ● ●  
  
let dict = produce_glossary();  
let debug_dump_dict = || {  
    for (key, value) in &dict {  
        println!("{} - {}", key, value);  
    }  
};
```

- 이렇게 하면 문제가 해결된다. 이제 이 함수는 Fn이므로 여러 번 호출할 수 있다.

- 클로저의 종류에는 변경할 수 있는 데이터나 `mut` 레퍼런스를 갖고 있는 클로저도 있다.
- Rust의 안전 판단 기준
 - 클로저가 `mut` 데이터를 갖지 않는 경우 여러 스레드에 공유해도 안전하다.
 - 하지만 `mut`가 아닌 클로저가 `mut` 데이터를 갖는 경우 여러 스레드에 공유하는 건 안전하지 않다.
 - 이런 클로저를 여러 스레드에서 호출하게 되면 스레드들이 동시에 같은 데이터를 읽고 쓰려 하기 때문에 온갖 종류의 경합 상태가 발생할 수 있기 때문이다.

- 따라서 Rust는 클로저의 범주를 하나 더 마련해 뒀는데, 쓰는 클로저를 위한 FnMut다.
- FnMut 클로저는 mut 레퍼런스를 통해 호출되며, 다음과 같이 정의되어 있다.

```
● ● ●

trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

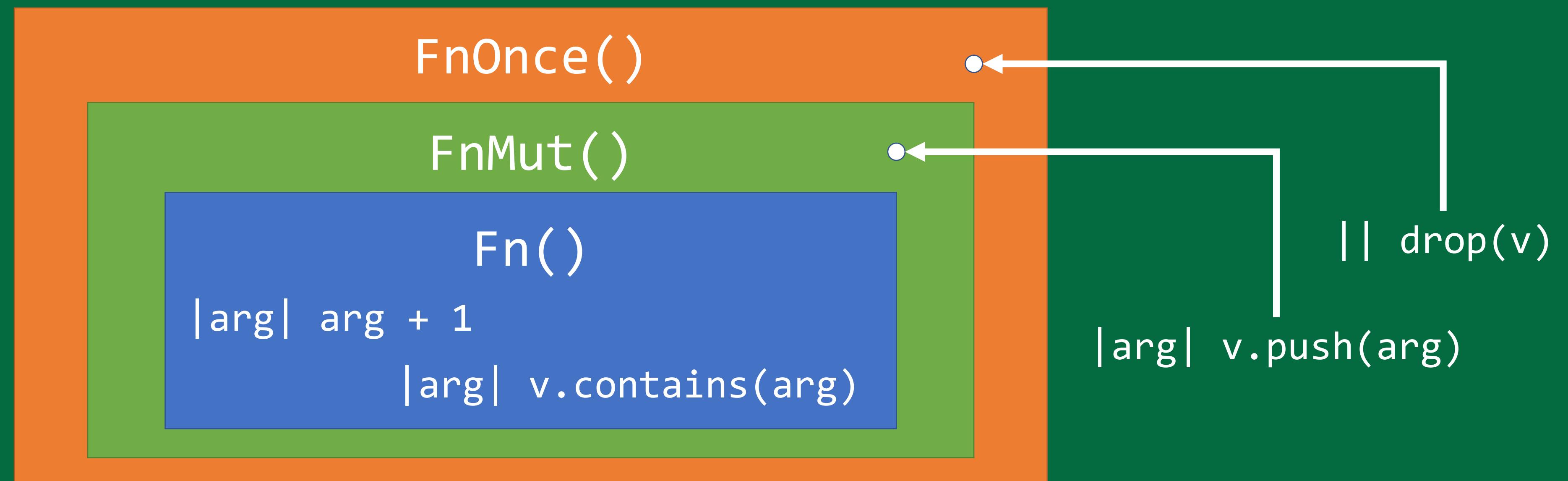
- FnMut 클로저는 값을 mut 접근 권한을 필요로 하지만 아무런 값도 드롭하지 않는 클로저다.

```
● ● ●

let mut i = 0;
let incr = || {
    i += 1;
    println!("Ding! i is now: {}", i);
}
call_twice(incr);
```

- 앞에서 봤던 call_twice는 Fn을 요구한다. 그러나 incr은 Fn이 아니라 FnMut이므로 앞의 코드는 컴파일되지 않는다.
- 이 문제의 해결책을 이해하려면 지금까지 살펴본 Rust 클로저의 세 가지 범주에 관한 내용을 살펴봐야 한다.

- Rust 클로저의 세 가지 범주
 - Fn은 제한 없이 여러 번 호출할 수 있는 클로저와 함수군이다. 모든 fn 함수 역시 이 최상위 범주에 포함된다.
 - FnMut는 클로저 자체를 mut로 선언한 경우에 여러 번 호출할 수 있는 클로저군이다.
 - FnOnce는 호출부가 클로저를 소유한 경우에 한 번만 호출할 수 있는 클로저군이다.
- 모든 Fn은 FnMut의 요건을 충족하고, 모든 FnMut는 FnOnce의 요건을 충족한다.



- Fn()은 FnMut()의 서브트레잇이고 FnMut()는 FnOnce()의 서브트레잇이다.
 - 따라서 Fn은 가장 배타적인 범주이자 가장 강력한 범주다.
 - FnMut와 FnOnce는 용도에 제한이 있는 클로저를 포함하는 더 넓은 범주다.
- 해결책
 - call_twice 함수가 모든 FnMut 클로저를 받도록 고쳐서 수용할 수 있는 클로저의 범주를 넓히면 된다.

```
● ● ●  
  
fn call_twice<F>(mut closure: F) where F: FnMut() {  
    closure();  
    closure();  
}
```

- 첫 번째 줄에 있는 제약이 F: Fn()에서 F: FnMut()로 바뀌었다. 이렇게 하면 여전히 모든 Fn 클로저를 받을 수 있는 데다, 데이터를 변경하는 클로저에 대해서도 call_twice를 쓸 수 있게 된다.

클로저를 위한 Copy와 Clone

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

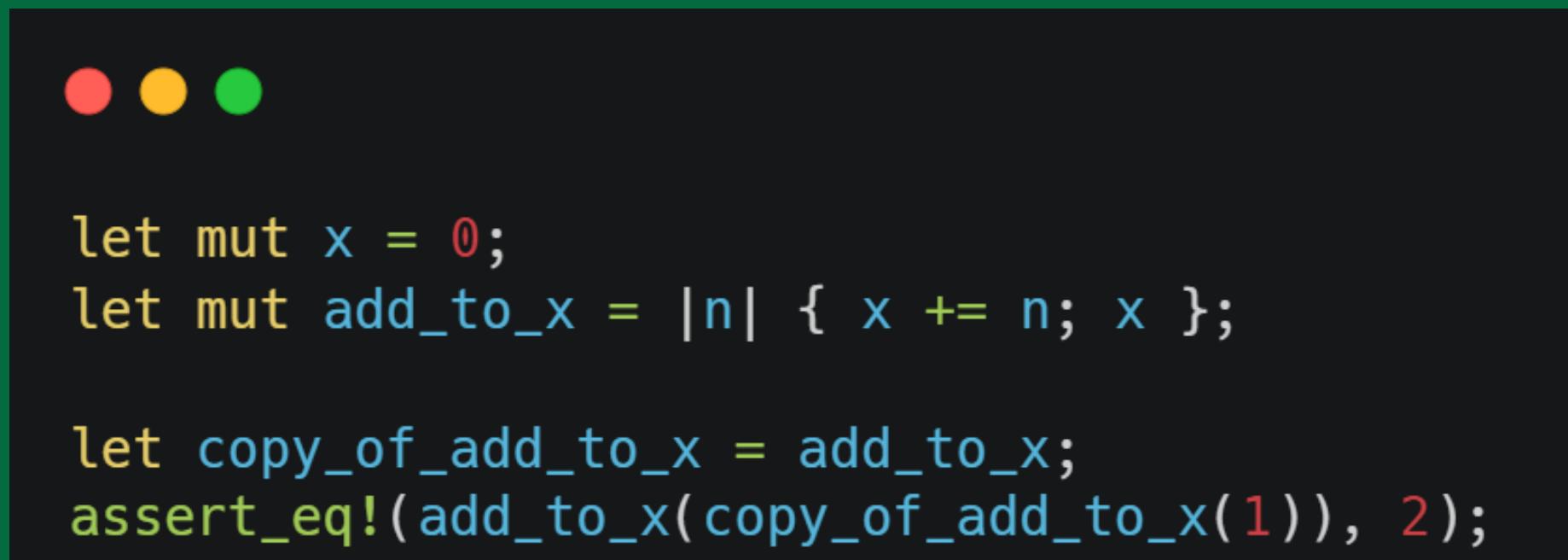
- Rust는 한 번만 호출될 수 있는 클로저를 알아서 파악했던 것처럼, Copy와 Clone을 구현할 수 있는 클로저와 그럴 수 없는 클로저를 파악할 수 있다.
- 클로저는 (move 클로저일 경우) 자신이 캡처하는 변수의 값이나 (move가 아닌 클로저일 경우) 그 값의 레퍼런스를 갖는 구조체로 표현된다.
- 클로저를 위한 Copy와 Clone 규칙은 일반적인 구조체를 위한 Copy와 Clone 규칙과 같다.
- 변수를 변경하지 않는 move가 아닌 클로저는 Clone이면서 Copy인 공유된 레퍼런스만 취하므로, 클로저 역시 Clone이면서 Copy다.

```
● ● ●  
  
let y = 10;  
let add_y = |x| x + y;  
let copy_of_add_y = add_y;  
assert_eq!(add_y(copy_of_add_y(22)), 42);
```

클로저를 위한 Copy와 Clone

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- Rust는 한 번만 호출될 수 있는 클로저를 알아서 파악했던 것처럼, Copy와 Clone을 구현할 수 있는 클로저와 그럴 수 없는 클로저를 파악할 수 있다.
 - 반면 값을 변경하는 move가 아닌 클로저는 내부 표현 안에 변경할 수 있는 레퍼런스를 갖는다.
 - 변경할 수 있는 레퍼런스는 Clone도 아니고 Copy도 아니므로 이를 사용하는 클로저 역시 둘 다 아닌 게 된다.



A terminal window showing the execution of a Rust program. The window has three colored dots (red, yellow, green) at the top left. The code is as follows:

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x;
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2);
```

클로저를 위한 Copy와 Clone

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- Rust는 한 번만 호출될 수 있는 클로저를 알아서 파악했던 것처럼, Copy와 Clone을 구현할 수 있는 클로저와 그럴 수 없는 클로저를 파악할 수 있다.
- move 클로저의 경우에는 규칙이 훨씬 더 단순하다.
- move 클로저가 캡처하는 게 전부 Copy면 클로저도 Copy고, 전부 Clone이면 클로저도 Clone이다.

```
● ● ●

let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{}", greeting);
};

greet.clone()( "Alfred" );
greet.clone()( "Bruce" );
```

- 많은 라이브러리가 **콜백(Callback)**을 자신이 가진 API의 일부로 사용한다.
 - 콜백은 라이브러리가 나중에 호출할 수 있도록 사용자가 제공하는 함수다.
 - actix-web 프레임워크를 사용해 간단한 웹 서버 프로그램을 만든다고 가정하자. 여기서 라우터가 중요한 역할을 한다.



```
● ● ●

App::new()
    .route("/", web::get().to(get_index))
    .route("/gcd", web::post().to(post_gcd))
```

- 라우터의 목적은 인터넷에서 들어오는 요청을 가져다가 해당 유형의 요청을 처리하는 Rust 코드로 보내는 것이다. 위 코드에서 get_index와 post_gcd는 프로그램 어딘가에 fn 키워드로 선언된 함수의 이름이다.

- 그러나 그 자리에 다음처럼 클로저를 전달할 수도 있다.
 - 이렇게 할 수 있는 이유는 actix-web이 스레드 안전성을 가진 Fn을 인수로 받도록 작성되었기 때문이다.



```
App::new()
    .route(
        "/",
        web::get().to(|| {
            HttpResponse::Ok()
                .content_type("text/html")
                .body("<title>GCD Calculator</title>...")
        }),
    )
    .route(
        "/gcd",
        web::post().to(|form: web::Form<GcdParameters>| {
            HttpResponse::Ok().content_type("text/html").body(format!(
                "The GCD of {} and {} is {}.",
                form.n,
                form.m,
                gcd(form.n, form.m)
            ))
        }),
    )
}
```

- 우리 프로그램에서 이렇게 할 수 있으려면 어떻게 해야 할까?
아주 간단한 우리만의 라우터를 actix-web의 도움 없이 처음부터 작성해 보자.
- 먼저 HTTP 요청과 응답을 나타내는 타입 몇 가지를 선언하는 것으로 시작하자.

```
● ● ●  
  
struct Request {  
    method: String,  
    url: String,  
    headers: HashMap<String, String>,  
    body: Vec<u8>,  
}  
  
struct Response {  
    code: u32,  
    headers: HashMap<String, String>,  
    body: Vec<u8>,  
}
```

- 우리만의 라우터 만들기

- 이제 라우터는 URL과 콜백의 매핑 테이블을 저장해 두었다가 요구가 있을 때 올바른 콜백을 호출하기만 하면 된다.
- 안타깝지만 이 코드에는 잘못된 부분이 하나 있다. 어떤 부분이 잘못됐을까?

```
● ● ●

struct BasicRouter<C>
where
    C: Fn(&Request) -> Response,
{
    routes: HashMap<String, C>,
}

impl<C> BasicRouter<C>
where
    C: Fn(&Request) -> Response,
{
    /// Create an empty router.
    fn new() -> BasicRouter<C> {
        BasicRouter {
            routes: HashMap::new(),
        }
    }

    /// Add a route to the router.
    fn add_route(&mut self, url: &str, callback: C) {
        self.routes.insert(url.to_string(), callback);
    }
}
```

- 원인 분석
 - 문제는 BasicRouter 타입을 정의한 방식에 있다.



```
struct BasicRouter<C> where C: Fn(&Request) -> Response {  
    routes: HashMap<String, C>  
}
```

- 무의식적으로 각 BasicRouter가 단일 콜백 타입 C를 가지며, HashMap에 있는 모든 콜백이 해당 타입으로 되어 있다고 선언해버린 것이다.

- 해결 방법
 - 다양한 타입을 지원하고자 하는 경우이므로 박스와 트레잇 오브젝트를 써야 한다.



```
type BoxedCallback = Box<dyn Fn(&Request) -> Response>;  
  
struct BasicRouter {  
    routes: HashMap<String, BoxedCallback>  
}
```

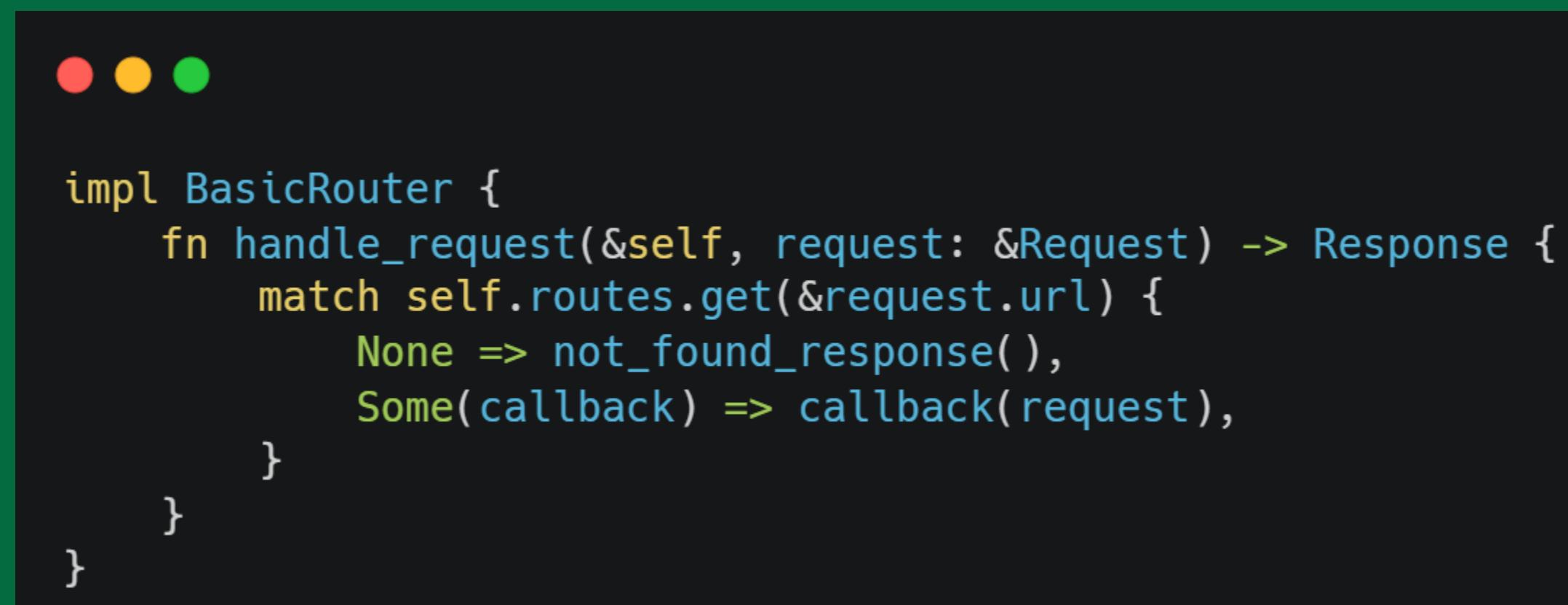
- 각 박스는 서로 다른 타입의 클로저를 가질 수 있으므로 하나의 `HashMap` 안에 모든 종류의 콜백을 담을 수 있다.
타입 매개 변수 `C`가 사라졌다는 점에 주목하자.

- 해결 방법
 - 메소드도 조금 고쳐야 한다.

```
impl BasicRouter {  
    // Create an empty router.  
    fn new() -> BasicRouter {  
        BasicRouter {  
            routes: HashMap::new(),  
        }  
    }  
  
    // Add a route to the router.  
    fn add_route<C>(&mut self, url: &str, callback: C)  
    where  
        C: Fn(&Request) -> Response + 'static,  
    {  
        self.routes.insert(url.to_string(), Box::new(callback));  
    }  
}
```

- `add_route`의 타입 시그니처를 보면 `C`의 제약이 2개인데, 특정 형식으로 된 Fn 트레잇과 ‘`static`’ 수명이 바로 그것이다. 여기서 ‘`static`’ 제약이 없으면 `Box::new(callback)`을 호출할 때 오류가 발생할 수 있다. 왜냐하면 곧 범위에서 사라질 변수의 차용된 레퍼런스를 가지고 있는 클로저의 경우에는 저장해 두는 것이 안전하지 않기 때문이다.

- 이제 이 라우터를 가지고 들어오는 요청을 처리할 준비가 끝났다.



```
impl BasicRouter {
    fn handle_request(&self, request: &Request) -> Response {
        match self.routes.get(&request.url) {
            None => not_found_response(),
            Some(callback) => callback(request),
        }
    }
}
```

- 좀 더 생각해보기

- 트레이트 오브젝트를 저장하지 말고 **함수 포인터(Function Pointer)**나 fn 타입을 쓰면 유연성은 조금 떨어져도 공간 효율이 좀 더 좋은 버전의 라우터를 작성할 수 있다.
`fn(u32) -> u32`와 같은 타입은 클로저와 매우 비슷하게 동작한다.



```
fn add_ten(x: u32) -> u32 {
    x + 10
}

let fn_ptr: fn(u32) -> u32 = add_ten;
let eleven = fn_ptr(1); // 11
```

- 좀 더 생각해보기

- 사실 자신의 환경에서 아무것도 캡처하지 않는 클로저는 캡처된 변수에 관한 추가 정보를 줄 필요가 없기 때문에 함수 포인터와 동일하다. 따라서 바인딩이나 함수 시그니처로 적절한 fn 타입을 지정해 두면, 컴파일러가 이를 찰떡같이 알아 듣고는 다음과 같은 식으로 쓸 수 있게 해준다.



```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;
let two = closure_ptr(1); // 2
```

- 캡처하는 클로저와 달리 이런 함수 포인터는 usize 하나 크기의 공간만 차지한다.

- 좀 더 생각해보기
 - 함수 포인터를 줘고 있는 라우팅 테이블의 모습은 다음과 같다.

```
● ● ●  
  
struct FnPointerRouter {  
    routes: HashMap<String, fn(&Request) -> Response>  
}
```

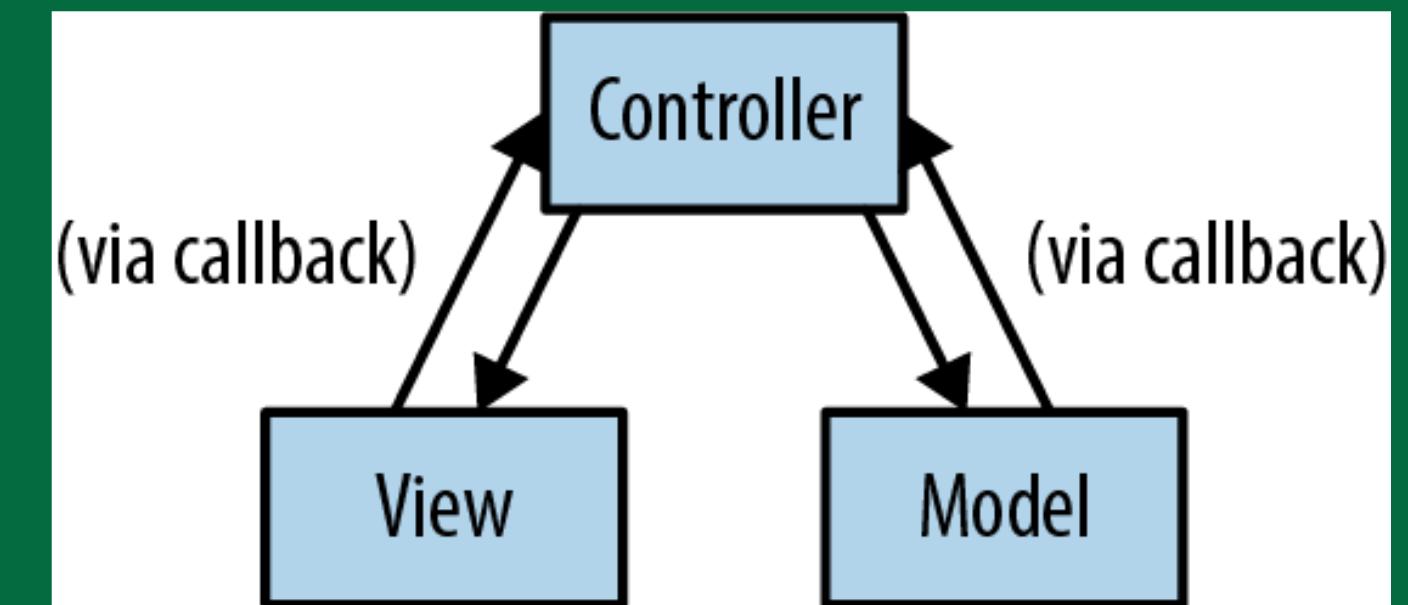
- 여기서 HashMap은 String 하나당 usize 하나만 저장하며, 결정적으로 Box를 쓰지 않는다. HashMap 자체를 제외하면 동적 할당이 전혀 없다. 물론, 메소드도 여기에 맞게 조금 손봐야 한다.

```
● ● ●  
  
impl FnPointerRouter {  
  
    // Create an empty router.  
    fn new() -> FnPointerRouter {  
        FnPointerRouter {  
            routes: HashMap::new(),  
        }  
    }  
  
    // Add a route to the router.  
    fn add_route(&mut self, url: &str, callback: fn(&Request) -> Response) {  
        self.routes.insert(url.to_string(), callback);  
    }  
}
```

효율적인 클로저 사용법

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

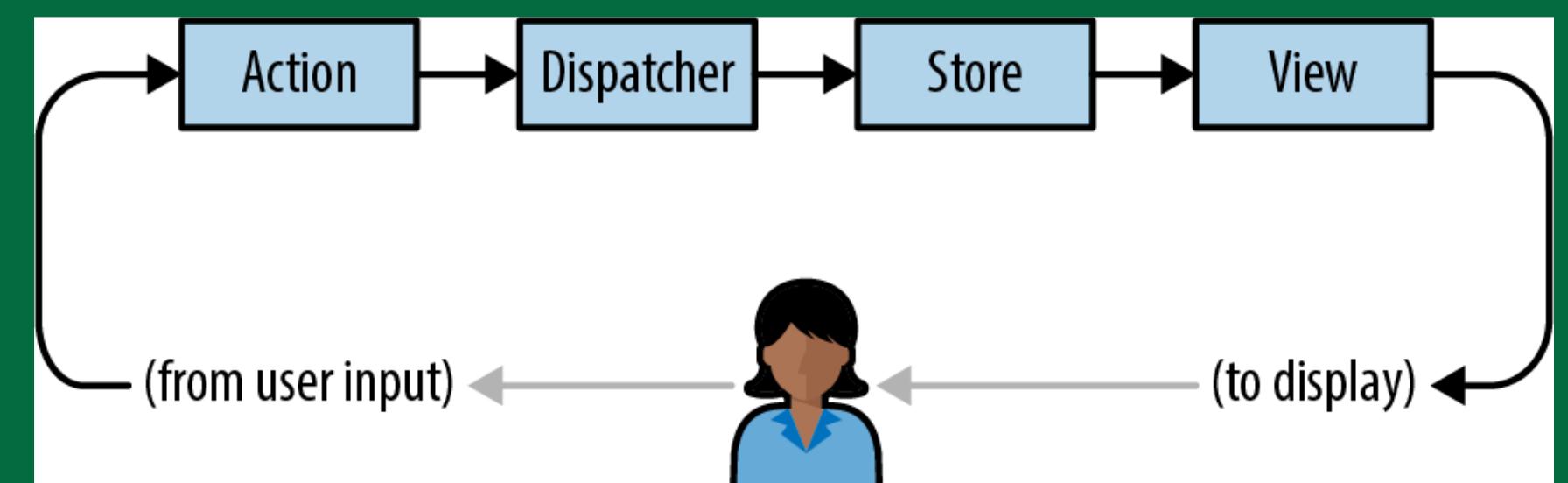
- Rust의 클로저는 대부분의 다른 언어에 있는 클로저와 다르다.
 - 가장 큰 차이점은 가비지 컬렉션이 있는 언어의 경우는 클로저에서 지역 변수를 사용할 때 수명이나 소유권에 대해 걱정할 필요가 없다는 것이다. 하지만 가비지 컬렉션이 없다면 이야기가 달라진다. Java, C#, JavaScript에서 흔히 볼 수 있는 디자인 패턴 중 일부는 그대로 가져다 써봤자 Rust에서 제대로 동작하지 않는다.
 - 예를 들어, **모델-뷰-컨트롤러(Model-View-Controller; MVC)** 디자인 패턴을 살펴 보자. MVC 프레임워크는 사용자 인터페이스의 모든 요소에 대해서 각각 세 가지 오브젝트를 생성한다.
 - UI 요소의 상태를 나타내는 **모델(Model)**
 - 그의 겉모양을 책임지는 **뷰(View)**
 - 사용자 상호 작용을 처리하는 **컨트롤러(Controller)**
 - 보통 각 오브젝트는 다른 오브젝트를 직접 또는 콜백을 통해 간접 참조한다. 한 오브젝트에서 무슨 일이 생길 때마다 이를 나머지 오브젝트들에게 알려서 모두를 즉시 업데이트하기 위해서다.
 - 그런데 문제는 여기서 누가 누굴 “소유”하는지 전혀 알 수 없다는 것이다.



효율적인 클로저 사용법

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- Rust에서는 이 패턴을 그대로 가져다 구현할 수 없다.
 - 고칠 부분이 조금 있는데, 소유권을 명시해야 하고 참조 순환을 제거해야 한다. 또 모델과 컨트롤러는 서로를 직접 참조할 수 없다.
- Rust는 좋은 대안 설계를 마련하는 쪽에 과감히 판돈을 걸었다.
 - 이를테면 각 클로저가 필요로 하는 레퍼런스를 인수로 받도록 만들어서 클로저의 소유권과 수명에 관한 문제를 해결하는 방법이 있다.
 - 또한, 시스템 안에 있는 모든 것에 번호를 부여해서 레퍼런스 대신 이 번호를 돌리는 방법도 생각해 볼 수 있다.
 - 오브젝트들이 서로를 아예 참조하지 않는 MVC의 여러 변형 중 하나를 구현해 볼 수도 있다.
 - 아니면 메타(Meta)의 플럭스(Flux) 아키텍처처럼 데이터가 한 방향으로 흐르는 비MVC 시스템을 본떠서 우리만의 툴킷을 만들 수도 있다.



참고 자료

Konkuk University GDSC + EDGE 스터디
Week 6: Closures

- <https://www.rust-lang.org/>
- <https://doc.rust-lang.org/book/>
- <https://tourofrust.com/>
- The Rust Programming Language (No Starch Press, 2019)
- Programming Rust, 2nd Edition (O'Reilly, 2021)

Thank you!