RUST 컬렉션

송민호

유튜브: https://youtu.be/fealeeauAxk

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

컬렉션

- 러스트의 표준 라이브러리에 포함되는 데이터 구조들
  - 대부분의 다른 데이터 타입은 단일한 특정 값을 나타냄
  - 컬렉션은 다수의 값을 담을 수 있음

- 컬렉션이 가리키는 데이터들은 힙에 저장됨
  - 데이터의 양이 컴파일 타임에 결정되지 않아도 되며 프로그램 실행 중에 늘어나거나 줄어들 수 있음

- 러스트에서 자주 사용되는 세가지 컬렉션
  - 벡터(vector), 문자열(string), 해시맵(hash map)

벡터

- 메모리에서 모든 값을 서로 이웃하도록 배치하는 단일 데이터 구조에 하나 이상의 값을 저장할 수 있음
  - 같은 타입의 값만을 저장할 수 있음

- 벡터 생성

```rust
let v: Vec<i32> = Vec::new();
```

- Vec::new 함수를 호출
- <> 안에 해당 타입 지정

```rust
let v = vec![1, 2, 3];
```

- vec! 매크로를 통해 생성 가능
- 타입 미지정 시 기본 타입은 i32

벡터 업데이트

- 벡터에 값을 추가하기 위해서는 가변으로 만들어줘야 함

- mut 키워드를 사용하여 해당 변수를 가변으로 만듦

- Push 메서드를 사용하여 벡터에 값을 추가할 수 있음

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

벡터 요소 읽기

- 벡터에 저장된 값을 참조하는 방법 – 인덱싱, get 메서드
- &와 []를 사용하면 인덱스 값에 위치한 요소의 참조를 얻게됨
- Get 함수에 인덱스를 매개변수로 넘기면 match를 통해 처리할 수 있는 Option<&T>를 얻게 됨

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
println!("The third element is {third}");

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

벡터 소유권

- 소유권 및 대여 규칙 적용
  - 유효한 참조자가 있다면 이 참조자와 벡터의 내용물로부터 얻은 다른 참조자들이 계속 유효하게 남아있도록 보장함

- 벡터는 모든 요소가 서로 붙어서 메모리에 저장
  - 벡터 메모리 위치에 새로운 요소를 추가할 공간이 없다면, 다른 곳에 메모리 할당
  - 이후 기존 요소를 새로 할당한 공간에 복사

  - **기존 요소의 참조자가 해제된 메모리를 가리키게 되므로 불가**

```
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);

println!("The first element is: {first}");
```

```
  |
4 |     let first = &v[0];
  |                 - immutable borrow occurs here
5 |
6 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("The first element is: {first}");
  |                                      ----- immutable borrow later used here
```

벡터 값 반복

- 벡터 내의 각 요소에 차례대로 접근
  - 인덱스를 사용하여 하나씩 값에 접근하기보다는 모든 요소에 대한 반복 처리

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{i}");
}
```

```
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

- 가변 참조자가 가리키는 값 수정
  - += 연산자 이전에 * 역참조 연산자로 i의 값을 얻어야함

문자열

- String 타입은 언어의 핵심 기능에 구현된 것이 아니고 러스트의 표준 라이브러리를 통해 제공됨
  - 커질 수 있고, 가변적이며, 소유권을 갖고 있고, UTF-8으로 인코딩된 문자열

- 문자열 생성
  - String::new 함수 통해 생성

```
let mut s = String::new();
```

```
let s = String::from("initial contents");
```

```
let data = "initial contents";

let s = data.to_string();

// 이 메서드는 리터럴에서도 바로 작동합니다:
let s = "initial contents".to_string();
```

문자열 업데이트

- Push_str과 push를 이용

- Push_str 메서드는 문자열 슬라이스를 매개변수로 갖음
  - 이는 매개변수의 소유권을 가져올 필요가 없기 때문
  - Push_str 함수가 s2의 소유권을 가져갔다면 마지막 줄에서 이 값을 출력할 수 없었을 것

```
let mut s = String::from("foo");
s.push_str("bar");
```

```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {s2}");
```

- Push 메서드는 한 개의 글자를 매개변수로 받음

```
let mut s = String::from("lo");
s.push('l');
```

문자열 업데이트

- 연산자나 format! 매크로를 이용한 접합

- + 연산자는 add 메서드를 사용

```
fn add(self, s: &str) -> String {
```

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // s1은 여기로 이동되어 더 이상 사용할 수 없음을 주의하세요
```

- S2에는 &가 있는데, 즉 첫 번째 문자열에 두 번째 문자열의 참조자를 더하고 있음
- String에는 &str만 더할 수 있고, 두 String 끼리는 더하지 못함
- 하지만 &s2의 타입은 &String이며 add의 두 번째 매개변수에 지정된 &str이 아님

- Add 호출에서 &String 인수가 &str로 강제될 수 있기 때문
- Add 함수가 호출되면, 러스트는 역참조 강제를 사용 - &s2를 &s2[..]로 바꿈

- Add가 매개변수의 소유권을 가져가지 않으므로 s2는 이 연산 후에도 계속 유효한 String

문자열 업데이트

```
fn add(self, s: &str) -> String {
```

```
    let s1 = String::from("Hello, ");
    let s2 = String::from("world!");
    let s3 = s1 + &s2; // s1은 여기로 이동되어 더 이상 사용할 수 없음을 주의하세요
```

- add가 self의 소유권을 가져감
  - 이는 self가 &를 안 가지고 있기 때문
  - S1 add 호출로 이동되어 이후에는 더 이상 유효하지 않음

문자열 업데이트

- Format! 매크로

- Format! 매크로는 println!처럼 작동
  - 화면에 결과를 출력하는 대신 결과가 담긴 String을 반환

- Format! 매크로로 만들어진 코드는 참조자를 이용
  - 이 호출은 아무 매개변수의 소유권도 가져가지 않음

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{s1}-{s2}-{s3}");
```

해시맵

- HashMap<K, V> - K 타입의 키, V 타입의 값에 대해 해시함수를 사용하여 매핑한 것을 저장
  - 이 해시 함수는 이 키와 값을 메모리 어디에 저장할지 결정
  - 임의의 타입으로 된 키를 이용하여 데이터를 찾고 싶을 때 유용

- 해시맵 생성
  - 표준 라이브러리 컬렉션 부분으로부터 HashMap을 use로 가져와야함
  - HashMap::new 함수 호출

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

해시맵

- Get 메서드에 키를 제공하여 해시맵으로부터 값을 얻어옴

- Get 메서드는 Option<&V>를 반환
  - 해시맵에 해당 키에 대한 값이 없다면 None을 반환

- Copied를 호출하여 Option<&i32>가 아닌 Option<i32>를 얻어옴
  - Unwarp_or을 써서 scores가 해당 키에 대한 아이템을 가지고 있지 않을 경우 score에 0을 설정

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name).copied().unwrap_or(0);
```

해시맵과 소유권

- 키와 값이 삽입되는 순간 해시맵의 소유가 됨

- Insert를 호출하여 field_name, field_value를 해시맵으로 이동시킨 후에는 더 이상 이 둘을 사용할 수 없음

```
use std::collections::HashMap;

let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name과 field_value는 이 시점부터 유효하지 않습니다.
```

해시맵 업데이트

- 이전에 저장된 값을 덮어씀
  - 똑같은 키에 다른 값을 삽입하면 해당 키에 연관된 값은 새 값으로 대신함

```
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{:?}", scores);
```

- 키가 없을 때 키, 값 추가
  - Entry 메서드 사용, Entry는 검사하려는 키를 매개변수로 받음
  - Or_insert 메서드는 해당 키가 존재하지 않을 경우 매개변수로 제공된 값을 새 값으로 삽입

```
use std::collections::HashMap;

let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{:?}", scores);
```

Q & A