Vectors and Hash Maps

Performant Software Systems with Rust — Lecture 7

Baochun Li, Professor

Department of Electrical and Computer Engineering

University of Toronto

Vectors: `Vec<T>`

- Growable stores on the heap that contain multiple values of the **same type**
  - such as characters in a `String`

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

```
1  let v: Vec<i32> = Vec::new(); // type annotation is required
```

```
1  let mut v = Vec::new(); // no type annotation needed
2  v.push(5); // updating a vector by pushing a value into it
```

Reading Elements of a Vector

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

    let third: &i32 = &v[2]; // panics if the index is out of bounds
    println!("The third element is {third}");

    let third: Option<&i32> = v.get(2); // returns None, no panic!
    match third {
        Some(third) => println!("The third element is {third}"),
        None => println!("There is no third element."),
    }
} // v goes out of scope and is freed here
```

Enforcing Ownership Rules

```
1 let mut v = vec![1, 2, 3, 4, 5];
2
3 let first = &v[0]; // immutable borrow
4 v.push(6); // mutable borrow
5 println!("The first element is: {first}");
```

```
cannot borrow `v` as mutable because it is also borrowed as immutable
```

Iterating Over the Values in a Vector

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

for i in &v {
    println!("{i}");
}

for i in &mut v {
    *i += 50; // dereference the mutable reference
}
```

Live Demo

Using an Enum to Store Multiple Types in a Vector

```rust
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

Hash Maps: `HashMap<K, V>` Are Key-Value Stores

- Stores a mapping of keys of type `K` (same type) to values of type `V` (same type) on the heap
  - Using a hash function
  - Useful when you wish to look up data using keys rather than indices
  - Similar to dictionaries in Python

Inserting Key-Value Pairs into Hash Maps

- Moves ownership into the hash map, except references

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

let mut scores = HashMap::new();

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

for (key, value) in &scores {
    println!("{key}: {value}");
}
```

Updating a Hash Map by Replacing a Value

```
let mut scores = HashMap::new();

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

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

```
{"Blue": 25}
```

Adding a Key and Value Only If a Key Is Not Present

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

// entry() returns a mutable reference
scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

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

```
{"Yellow": 50, "Blue": 10}
```

Updating a Value Based on the Old Value

```rust
let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{map:?}");
```

```
{"world": 2, "wonderful": 1, "hello": 1}
```

Required Additional Reading

The Rust Programming Language, Chapter 8