Hash Maps

Getting Started

Hash maps are useful when you want to look up data not by using an index, as you can with vectors, but by using a key that can be of any type. For example, in a game, you could keep track of each team's score in a hash map in which each key is a team's name and the values are each team's score. Given a team name, you can retrieve its score.

Program 1

```
fn main() {
        use std::collections::HashMap;
        let mut scores = HashMap::new();
        scores.insert(String::from("Blue"), 10);
        scores.insert(String::from("Yellow"), 50);
        println!("{:?}",scores);
}
```

Just like vectors, hash maps store their data on the heap. This HashMap has keys of type String and values of type i32. Like vectors, hash maps are homogeneous: all of the keys must have the same type, and all of the values must have the same type.

Program 2

```
fn main() {
        use std::collections::HashMap;
        let teams = vec![String::from("Blue"), String::from("Yellow")];
        let initial_scores = vec![10, 50];
        let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
        println!("{:?}",scores);
}
```

For example, if we had the team names and initial scores in two separate vectors, we could use the zip method to create a vector of tuples where "Blue" is paired with 10, and so forth. Then we could use the collect method to turn that vector of tuples into a hash map.

The type annotation HashMap<_, _> is needed here because it's possible to collect into many different data structures and Rust doesn't know which you want unless you specify. For the parameters for the key and value types, however, we use underscores, and Rust can infer the types that the hash map contains based on the types of the data in the vectors.

Hash Maps and Ownership

For types that implement the Copy trait, like i32, the values are copied into the hash map. For owned values like String, the values will be moved and the hash map will be the owner of those values.

Program 3

```
fn main() {
        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_name and field_value are invalid at this point, try using them and
        // see what compiler error you get!
        println!("{}",field_name);
}
```

Program 4

```
fn main() {
        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);
}
```

Program 5

```
fn main() {
        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); // Do you think team_name ownership will move and this program will be executed?
}
```

Accessing Values in a Hash Map

Program 6

```
fn main() {
    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); // This method only accepts the reference
    println!("{:?}",score);
}
```

Assignment

Using match operator, print the value of score

Program 7

```
fn main() {
    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 { // scores will move
    println!("{}: {}", key, value);
    }
}
```

Program 7

```
fn main() {
        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 { // Only reference will pass
        println!("{}: {}", key, value);
        }
}
```

Updating a Hash Map

Overwriting a value

Program 8

```
use std::collections::HashMap;
fn main() {
        let mut scores = HashMap::new();
        scores.insert(String::from("Blue"), 10);
        scores.insert(String::from("Blue"), 25);
        println!("{:?}", scores);
}
```

Only Inserting a Value If the Key Has No Value

Program 9

```
use std::collections::HashMap;
fn main() {
        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);
}
```

Updating a Value Based on the Old Value

Program 10

```
use std::collections::HashMap;
fn main() {
	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);
}
```

The or_insert method actually returns a mutable reference (&mut V) to the value for this key. Here we store that mutable reference in the count variable, so in order to assign to that value, we must first dereference count using the asterisk (*).