Rust - Day 9

Common Collections

- Rust's stantard library contains some useful DS called **Collections** that can contain multiple values.
- Unlike built-in array and tuple types, the data these collections point to is stored on the heap which means the amount of data does not need to be known at compile time and can grow and shrink.
- These collections include:
  - Vector: Store variable number of values (same type) next to each other
  - String: Collection of characters
  - Hashmap: Key-Value storage

Vectors

- Vectors allow you to store multiple values in a single data structure next to each other.
- Vectors can only store values of the same type.

Creating a vector

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

- We added a type annoation to define the type of elements to be stored.
- Vectors are implemented using Generics, allowing to hold any type of data.
- Rust also infers the type of value based off the first value entered
- Rust conveniently provides the `vec!` macro, which will create a new vector that holds the values you give it.

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

- Here, the type is inferred as `i32`.

Updating a vector

- We can use `push` method on an mutable vector.

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

v.push(1);
v.push(2);
v.push(3);
v.push(4);
```

Reading Elements of Vectors

- To refer to a value stored in a vector, you can use
  - Indexing
  - `get` method

```
let v: Vec<i32> = vec![1, 2, 3, 4, 5];

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

let fifth: Option<&i32> = v.get(2);
match fifth {
```

```
        Some(fifth) => println!("The fifth element is {fifth}"),
        None => println!("There is no fifth element."),
    }
```

- Vectors are indexed starting at zero.
- Using `&` and `[]` gives us a reference to the element at the index value.
- **When we use the `get` method with the index passed as an argument, we get an `Option<&T>` that we can use with `match`.** This handles the edge case where the index value might not exist.

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

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

- Using `[]` would crash the program when out-of-bound value is accessed while `get` would return the `None` type.
- When we hold an immutable reference to the first element in a vector and try to add an element to the end, then the program would crash.

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6);
```

```
$ cargo run
  Compiling collections v0.1.0 (file:///projects/collections)
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
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

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` (bin "collections") due to 1 previous error
```

> Why should a reference to the first element care about changes at the end of the vector? This error is due to the way vectors work: because vectors put the values next to each other in memory, adding a new element onto the end of the vector might require allocating new memory and copying the old elements to the new space, if there isn't enough room to put all the elements next to each other where the vector is currently stored. In that case, the reference to the first element would be pointing to deallocated memory. The borrowing rules prevent programs from ending up in that situation.

Iterating Over the Values in a Vector

- To iterate over a vector, use `for` loop

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

- We can also iterate over mutable references to each element in a mutable vector in order to make changes to all the elements.

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

- **To change the value that the mutable reference refers to, we have to use the `*` dereference operator to get to the value in `i` before we can use the `+=` operator.**
- Iterating over a vector, whether immutably or mutably, is safe because of the borrow checker's rules.

Using Enum to Store Multiple Types

- We can use enum to be able to store values of different types in an vector.

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

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

- **Rust needs to know what types will be in the vector at compile time so it knows exactly how much memory on the heap will be needed to store each element.
- **We must also be explicit about what types are allowed in this vector. If Rust allowed a vector to hold any type, there would be a chance that one or more of the types would cause errors with the operations performed on the elements of the vector. Using an enum plus a `match` expression means that Rust will ensure at compile time that every possible case is handled.**
- If you don't know the exhaustive set of types a program will get at runtime to store in a vector, the enum technique won't work. Instead, you can use a trait object, which we'll cover in Chapter 17.

Dropping a Vector drops its Elements

- Like any other `struct`, a vector is freed when it goes out of scope

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

    // do stuff with v
} // ← v goes out of scope and is freed here
```

- When the vector gets dropped, all of its contents are also dropped, meaning the integers it holds will be cleaned up. The borrow checker ensures that any references to contents of a vector are only used while the vector itself is valid.