Lecture 3

Compound Data Types

Overview

Compound Data Types

- Primitives: Tuples, Arrays
- Collections: Vectors
- Special Enums: Option, Result

Optional Reading:

The Rust Book

- Chapter 3 – Common Programming Concepts
- Chapter 8 – Common Collections

Recap

Rust has 4 primary scalar types:

- Integers
- Floating points
- Booleans
- Characters

Today we'll talk about compound types:

- Primitives: Tuples, Arrays
- Collections: Vectors
- Special Enums: Option, Tuple

Tuples

```
let tuple1 = (5, 3.0, "chicken");
let tuple2 = (true, 700, false);
// With type annotation
let tuple3: (u8, bool, i32) = (8, true, 20);
```

Tuples are groupings of values

- Values can be of different types!
- Fixed length, once declared, we cannot expand or shrink

Accessing tuple values

```
let tuple1 = (5, 3.0, "chicken");
let (x, y, z) = tuple1; // destructuring

println!("Y is {y}");

let number = tuple1.0; // indexing
let word = tuple1.2; // indexing

println!("Number is {number} and word is {word}");
```

Unit tuple

```
let unit_tuple = ();
```

There is a special tuple called the **unit tuple**

- Tuple without any values
- Represent empty value / empty return type

Arrays

```
let array1 = [1,2,3,4,5];
let array2: [char; 3] = ['a', 'b', 'c']; // type annotation

// Using repetition
let array3 = [true; 4]; // [true, true, true, true]
```

Arrays are lists of values of the same type:

- Every element must have the same type!
- Arrays have a fixed length once they are declared

Accessing array values

```
let array = [1,2,3,4,5];

let number1 = array[0];
let number2 = array[1];

println!("Number 1 is {number1}, Number 2 is {number2}");
```

Array indexing is similar to how it looks in other programming languages

Vector

- A collection type in Rust
- Like an array that can be resized
  - Because the vector is stored on the heap instead of the stack
- All elements of the vector have to be the same type

Initializing a vector

```
let myvector: Vec<i32> = Vec::new();
let yourvector = Vec::from(['a','b','c'])
let ourvector = vec![1,3,5,7]; // using a macro
```

3 main ways to initialize a vector:

- Using the Vec::new() function, need to annotate type
- Using the Vec::from() function, pass in an array
- Using the vec! macro, compiler can infer the type

Inserting into a vector

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

myvector.push(5); // Compiler infers type
myvector.push(7);

println!("{:?}", myvector); // [5, 7]
```

Using the push() method

Accessing a vector - Indexing

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

myvector.push(5);
myvector.push(6);

let a = myvector[0]; // indexing
println!("{}", a); // 5
```

Using traditional indexing, code will panic if we index past the length of the array

A detour … Option

```
enum Option<T> {
    None,
    Some(T),
}
```

Option is a special type in Rust that can be one of 2 variants

- None means that there is no value (almost like a null)
- Some has a value associated with it (think of it like wrapping a value)
- E.g. an Option<i32> can be a None or a Some(5)
- The .unwrap() method returns x from Some(x) or crashes from None

Accessing a vector - The get() & pop() methods

```
let mut myvector = Vec::new();
myvector.push(5); myvector.push(6);

let a = myvector.get(0); // returns an Option type
println!("{:?}", a); // Some(5)
let b = myvector.pop();
println!("{:?}", b); // Some(6)
```

Using the get() method, the code will not panic when we try to read an invalid index, instead it will only return None.
The pop() method removes the last element from the vector and returns it.

Iterating through a vector

```
for i in 0..(vector.len()) {
    // use index here
}
// foreach
for elem in vector.iter() {
    // use element here
}
```

Two main ways to iterate through a vector:

- For loop over the index. Access vector elements using get().
- Foreach. Iterates over references to each element.

Another enum: Result

```
enum Result<T,E> {
    Ok(T),
    Err(E),
}
```

Result is a special type in Rust that can be one of 2 variants

- Ok means that the function returns successfully (holds return value)
- Err means something went wrong (holds error value)

Result is typically used for error handling in functions

Recap

Compound data types — Tuples, arrays

Vectors:

- Initialization: Vec::new(), vec![]
- Adding elements: push()
- Accessing elements: Indexing, get() and pop() methods

Option and Result:

- None Types (nulls) and Error Values

Announcements

HW1 is already released

Due on Friday 02/7 23:59