Collections

Mustakimur R. Khandaker

Collections

Rust's standard library includes a number of very useful data structures called **collections**.

- collections can contain multiple values.
- Unlike the built-in array and tuple types, collection is stored on the heap.
  - So, amount of data does not need to be known at compile time and can grow or shrink as the program runs.

We will discuss about:

- Vector.
- String (Basics have been covered already).
- Hash Map.

Vectors

Vectors

A vector allows us to store a variable number of values next to each other.

- It can only store values of the same type.
- Vectors are implemented using generics; `vect<T>`

To create a new, empty vector:

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

Note: we added a type annotation here. Because we aren't inserting any values into the vector, so Rust doesn't know what kind of elements we intend to store.

Alternative approach:

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

vec! is a macro provided by Rust if initialized by value.

Vector: Operations

Updating a vector:

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

Dropping a Vector Drops Its Elements (exception vector of references):

```
{
 let v = vec![1, 2, 3, 4];
        // do stuff with v
} // <- v goes out of scope and is freed here
```

Reading Elements of Vectors:

- Get a reference of an item using & (out of bound access cause runtime panic):
-

```
let third: &i32 = &v[2];
```

- Get an Option<&T> using get (out of bound access returns None):

```
match v.get(2) {
        Some(third) => println!("The third element is {}", third),
        None => println!("There is no third element."),
}
```

Question

```rust
fn code2() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

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

This error is due to the way vectors work: 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 currently is.

```
$ 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

error: aborting due to previous error
```

More Operations

Iterating over the Values in a Vector:
To read:

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

To modify (dereference operator (*)):

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

Using an Enum to Store Multiple Types

Vectors can only store values that are the same type.

- However, the variants of an enum are defined under the same enum type.
  - So when we need to store elements of a different type in a vector, we can define and use an enum!

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

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

Note: 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.

String

String

The String type, which is provided by Rust's standard library, is a growable, mutable, owned, UTF-8 encoded string type.

- When we refer to "strings" in Rust, it usually mean the String and the string slice &str types.

Creating new string:

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

Converting &str to String:

```
let data = "initial contents";
let s = data.to_string();

// the method also works on a literal directly:
let s = "initial contents".to_string();
```

UTF-8 encoded:

```
let hello = String::from("Dobrý den");
let hello = String::from("Olá");
let hello = String::from("नमस्ते");
```

Updating a String

Appending a String Slice to a String:

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

To append a char to a String:

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

Concatenation with + operator also works:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // note s1 has been moved
here and can no longer be used
```

Internally, the above call the following function:

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

Basically, s1 is self and &s2 is &str of the function param, and the result returns (Move) a new String (i.e. s3).

More on Concatenation

To concat more than two strings:

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

let s = s1 + "-" + &s2 + "-" + &s3;
```

Alternative to String formatting:

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

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

The format! macro works in the same way as println!, but instead of printing the output to the screen, it returns a String with the contents.

- Similar to sprintf in C/C++.

Indexing String

Accessing String using index is prohibited.

- String holds UTF-8 characters (A String is a wrapper over a Vec<u8>).
  - It could be 1 to 4 bytes long.
- String[0] may or may not point to a partial UTF-8 characters i.e. invalid data.

```
let hello = String::from("Здравствуйте");
let hello = String::from("hello");
let answer = &hello[0];
println!("The first character is {}", answer);
```

However, String slices is possible i.e. create a string literals using a range. But, it could cause runtime panic if detects accessing partial bytes of a characters.

```
let hello = "Здравствуйте";
let s = &hello[0..4];
println!("The first four bytes of the String are {}", s);
```

Note: Above case, each character represents 2 bytes, so first 2 characters will be printed successfully.

String Iterator

Although indexing is prohibited for String, useful String iterators are available.

To iterate over characters or raw bytes:

```
let hello = "Здравствуйте";
for c in hello.chars() {
    print!("{} ", c);
}
println!();

for b in hello.bytes() {
    print!("{} ", b);
}
println!();
```

Hash Map

Hash Map

The type HashMap<K, V> stores a mapping of keys of type K to values of type V.

- It does this via a hashing function, which determines how it places these keys and values into memory.

To create a new HashMap and insert (key, value) pairs:

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

let mut scores = HashMap::new();

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

Alternative approach (create HashMap from other type of Collection):

```
let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];

let mut scores: HashMap<_, _> =
        teams.into_iter().zip(initial_scores.into_iter()).collect();
```

collect() can be used to create a number of other data structure, so explicitly mentioned the type i.e. HashMap<_, _>.

Other Operations

Accessing values in 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);
```

get() returns Option<&V> i.e. when key exists return Some(&V) and when key does not exists return None.

It is possible to iterate over the HashMap:

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

More on Insertion

Inserting a value only if a key does not have a value:

```
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 Key Value

Updating a value usually ends with overwriting a value:

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

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

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

To really update over an old value:

```
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);
```

< Collections />