1_Collections - Standard types_en

May 2, 2023

1 Collections - Standard types

Rust's standard library includes a number of very useful data structures called **collections**. These can contain multiple values.Unlike the 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 or shrink as the program runs.

Collections available in Rust:

- A **vector** allows you to store a variable number of values next to each other.
- A **string** is a collection of characters. We've mentioned the String type previously, but in this chapter we'll talk about it in depth.
- A **hash map** allows you to associate a value with a particular key. It's a particular implementation of the more general data structure called a map.

2 Vectors

Vectors allow you to store more than one value in a single data structure that puts all the values next to each other in memory. Vectors can only store values of the same type.

[ ]:
```rust
let v: Vec<i32> = Vec::new(); // creates a new empty vector
```

In more realistic code, Rust can often infer the type of value you want to store once you insert values, so you rarely need to do this type annotation. It's more common to create a `Vec< T >` that has initial values, and Rust provides the **`vec!`** macro for convenience. In the following example, we've given initial `i32` values, so Rust can infer that the type of `v` is `Vec< i32 >`, and the type annotation isn't necessary.

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

2.1 Update vectors

[ ]:
```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
println!("Vector after update: {:?}",v);
```

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

2.2 Reading elements in a vector

[ ]:
```
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let third: &i32 = &v[2]; // first method; index starts at 0
    println!("The third element is {}", third);

    match v.get(2) { // returns an Option<&T>
        Some(third) => println!("The third element is {}", third),
        None => println!("There is no third element."),
    }
}
main();
```

The first `[]` method will cause the program to panic if it references a nonexistent element. This method is best used when you want your program to crash if there's an attempt to access an element past the end of the vector. When the `get` method is passed an index that is outside the vector, it returns `None` without panicking. You would use this method of accessing an element beyond the range of the vector happens occasionally under normal circumstances (e.g. user enters wrong value).

Consider the following example and recall the rule that states you can't have mutable and immutable references in the same scope:

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

    let first = &v[0];

    v.push(6);

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

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. In that case, the reference to the first element would be pointing to deallocated memory.

2.3 Iterating over the values in a vector

[ ]:
```rust
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
for i in v {  // consumes the vector
    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.

[ ]:
```rust
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
for i in &v {
    println!("{}", i);
}
```

Using iterators:

[ ]:
```rust
let v1 = vec![1, 2, 3];

for val in v1.iter() {
    println!("Got: {}", val);
}
```

2.4 Exercises

Exercise 1: Create a vector that stores data coming from spreadsheet cells and having different types: i32, f64 and String. The vector initially contains the elements: `[3, "blue", 10.12]`

[ ]:
```rust
fn main() {
}
main();
```

Tips : > Think about a data structure that allows you to store data of different types. We can define an enum whose variants will hold the different value types, and then all the enum variants will be considered the same type: that of the enum. Then we can create a vector that holds that enum and so, ultimately, holds different types.

Solution:

[ ]:
```rust
fn main() {
    enum SpreadsheetCell {
        Int(i32),
        Float(f64),
        Text(String),
    }

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

Exercise 2: Write a function that calculates the largest value in a vector of i32 elements. > * Don't forget the case where the vector is empty! > * Try to write a version that do not consume the vector.

[ ]:
```rust
// TO DO (1) Define a function that calculates the largest value

fn main() {
  // TO DO (2) Test your implementation here

}
main();
```

Tips: > Use the documentation to investigate the methods implemented by the std::vec::Vec type: https://doc.rust-lang.org/std/vec/struct.Vec.html

Solution:

[ ]:
```rust
/// Gets the largest of the vector and consume the vector
fn into_largest(list: Vec<i32>) -> Option<i32> {
    let mut result = None;
    for value in list.into_iter() {
        result = match result {
            None => Some(value),
            Some(previous) => Some(value.max(previous))
        };
    }
    result
}

/// Gets the largest of the slice, but do not consume it
fn largest_of(list: &[i32]) -> Option<i32> {
    let mut result = None;
    for &value in list.iter() {
        result = match result {
```

```
            None => Some(value),
            Some(previous) => Some(value.max(previous))
        };
    }
    result
}

/// Gets the largest of the slice, but do not consume it
fn largest_of_idiomatic(list: &[i32]) -> Option<i32> {
    list.iter().copied().max()
}

fn main() {
    let input = vec![10,2,3,4];
    let a = largest_of(&input);
    let b = into_largest(input); // consume input

    println!("largest_of   says: {:?}", a);
    println!("into_largest says: {:?}", b);
}

main();
```

Exercise 3: Define a structure called `GeoPolygon` with `ring` as a field representing a list of vertices of type `GeoCoordinate`. Then implement the `bound` method that returns a `GeoBoundingBox` representing the min and max latitudes and longitudes of the polygon. Use the `GeoCoordinate` and `GeoBoundingBox` structures defined earlier (lesson Structures).

Example coordinates for testing:

```
GeoCoordinate {
    latitude: 46.0,
    longitude: 2.0
},
GeoCoordinate {
    latitude: 47.0,
    longitude: 2.0
},
GeoCoordinate {
    latitude: 47.0,
    longitude: 3.0
},
GeoCoordinate {
    latitude: 46.0,
    longitude: 3.0
}
```

[ ]:
```rust
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

#[derive(Debug)]
struct GeoBoundingBox {
    /// The point with the minimal coordinates for this bounding box
    min: GeoCoordinate,
    /// The point with the maximal coordinates for this bounding box
    max: GeoCoordinate
}

struct GeoPolygon {
    /// TO DO (1) Define the rings of the polygon
    ring:
    // TO DO (2) Implement the bounds method that returns a GeoBoundingBox
}
```

Solution:

[ ]:
```rust
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

#[derive(Debug)]
struct GeoBoundingBox {
    /// The point with the minimal coordinates for this bounding box
    min: GeoCoordinate,
    /// The point with the maximal coordinates for this bounding box
    max: GeoCoordinate
}

#[derive(Debug)]
struct GeoPolygon {
    /// The ring of this polygon
    ring: Vec<GeoCoordinate>
}

impl GeoPolygon {
```

```
    // Gets the bounding box for a polygon
    fn bounds(&self) -> GeoBoundingBox {
        let mut min_lat = std::f64::MAX;
        let mut max_lat = std::f64::MIN;
        let mut min_lng = std::f64::MAX;
        let mut max_lng = std::f64::MIN;
        for point in self.ring.iter() {
            if point.latitude < min_lat {
                min_lat = point.latitude;
            }
            if point.latitude > max_lat {
                max_lat = point.latitude;
            }
            if point.longitude < min_lng {
                min_lng = point.longitude;
            }
            if point.longitude > max_lng {
                max_lng = point.longitude;
            }
        }
        GeoBoundingBox {
            min: GeoCoordinate {
                latitude: min_lat,
                longitude: min_lng
            },
            max: GeoCoordinate {
                latitude: max_lat,
                longitude: max_lng
            }
        }
    }
}

fn main() {
    let ring = vec![GeoCoordinate {
        latitude: 46.0,
        longitude: 2.0
    },
    GeoCoordinate {
        latitude: 47.0,
        longitude: 2.0
    },
    GeoCoordinate {
        latitude: 47.0,
        longitude: 3.0
    },
    GeoCoordinate {
```

```
        latitude: 46.0,
        longitude: 3.0
    }];
    let polygon = GeoPolygon { ring };
    let bounds = polygon.bounds();
    println!("{:#?}", &bounds);
}
main();
```

3 Strings

The String type, which is provided by Rust's standard library rather than coded into the core language, is a growable, mutable, owned, UTF-8 encoded string type.

[ ]:
```
let mut s = String::new(); // create a new String
```

[ ]:
```
let data = "initial contents";

let s = data.to_string();

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

[ ]:
```
let s = String::from("initial contents");
```

> **Note:** Rust's standard library also includes a number of other string types, such as `OsString`, `OsStr`, `CString`, and `CStr`. Library crates can provide even more options for storing string data. See how those names all end in String or Str? They refer to owned and borrowed variants, just like the `String` and `str` types you've seen previously. These string types can store text in different encodings or be represented in memory in a different way, for example.

[ ]:
```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}, s1 is {}", s2, s1);
```

The push method takes a single character as a parameter and adds it to the String.

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

3.1 String concatenation

[ ]:
```
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
```

For more complicated string combining, we can use the **format!** macro:

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

let s = format!("{}-{}-{}", s1, s2, s3); // does not take ownership of any of␣
 ↪its parameters
println!("{}", s);

s1.push_str("xx");
println!("{}", s1);
```

3.2 Internal representation

A String is a wrapper over a Vec< u8 >.

[ ]:
```
let len = String::from("Hola").len();
println!("{}",len);
let len = String::from("      ").len();
println!("{}",len);
```

The number of bytes it takes to encode “ ” in UTF-8 is 24 because each Unicode scalar value in that string takes 2 bytes of storage. Therefore, an index into the string's bytes will not always correlate to a valid Unicode scalar value. For this reason, indexing into strings is not a valid operation.

[ ]:
```
let hello = "      ";
let answer = &hello[0]; // error
```

3.3 Slicing strings

To be more specific in your indexing and indicate that you want a string slice, rather than indexing using [] with a single number, you can use [] with a range to create a string slice containing particular bytes:

[ ]:
```
let hello = "      ";

let s = &hello[0..4];
println!("s: {}", s);
```

3.4 Iterating over Strings

[ ]:
```rust
for c in "  ".chars() { // on each iteration returns a value of type char
    println!("{}", c);
}
let text = "  ";
println!("First char {:?}", text.chars().nth(0));
```

[ ]:
```rust
for b in "  ".bytes() { // on each iteration returns a raw byte
    println!("{}", b);
}
let bb: std::str::Bytes = "  ".bytes();
```

3.5 Exercises

Exercise 1: Write a small program that checks if a word is a palindrom.

[ ]:
```rust
fn main() {

}
main();
```

> **Tips:** To solve this problem, checkout the manual pages to understand how the iterator and its nth method works (https://doc.rust-lang.org/std/iter/trait.Iterator.html); use the bytes or chars iterator.

Solution:

[ ]:
```rust
fn palindrome(s: String) -> bool {
    let len = s.len();
    if len == 0 {return true;}
    let mut i = 0;
    for c in s.chars() {
        match s.chars().nth(len-i-1) {
            Some(value) => {if value != c { return false;}},
            None => break,
        }
        i += 1;
    }
    return true
}

fn palindrome_idiomatic(text: &str) -> bool {
    text
        .chars()
        .zip(text.chars().rev())
        .all(|(a, b)| a == b)
}
```

```
fn main() {
    let word = String::from("abccba");
    let another_word = String::from("abcccca");
    let empty_word = String::from("");
    match palindrome_idiomatic(&another_word) {
        true => println!("The word is a palindrome"),
        false => println!("The word is not a palindrome"),
    }
}
main();
```

4 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. 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.

4.1 Creating a new hash map

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

Another way of constructing a hash map is by using the **`collect`** method on a vector of tuples, where each tuple consists of a key and its value.

[ ]:
```
use std::collections::HashMap;
fn main()
{
    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: {:?}", scores);
}
main();
```

4.2 Hash map 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:

[ ]:
```
use std::collections::HashMap;

fn main() {
    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: {}", field_name);
}
main();
```

Conclusion: If we insert references to values into the hash map, the values won't be moved into the hash map. The values that the references point to must be valid for at least as long as the hash map is valid.

4.3 Accessing values

We can get a value out of the hash map by providing its key to the `get` method. Check out the documentation to see how get if defined: https://doc.rust-lang.org/std/collections/struct.HashMap.html#method.get

[ ]:
```
use std::collections::HashMap;

fn main() {
    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);
    match score {
        Some(&value) =>  println!("The score for the {} team is {}", team_name,␣
 ↪value),
        None => (),
    }
}
main();
```

We can iterate over each key/value pair in a hash map in a similar manner as we do with vectors,

using a for loop:

[ ]:
```
use std::collections::HashMap;

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

4.4 Updating a Hash Map

If we insert a key and a value into a hash map and then insert that same key with a different value, the value associated with that key will be replaced.

[ ]:
```
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);
}
main();
```

It's common to check whether a particular key has a value and, if it doesn't, insert a value for it. Hash maps have a special API for this called **`entry`** that takes the key you want to check as a parameter. The return value of the entry method is an enum called **`Entry`** that represents a value that might or might not exist.

[ ]:
```
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);
```

```
}
main();
```

The `or_insert` method on `Entry` is defined to return a mutable reference to the value for the corresponding `Entry` key if that key exists, and if not, inserts the parameter as the new value for this key and returns a mutable reference to the new value.

Another common use case for hash maps is to look up a key's value and then update it based on the old value.

4.5 Exercices

Exercise 1: Write the missing code that counts how many times each word appears in some text. *

> **Tips:** Look out for the method *split_whitespace* in the documentation pages. https://doc.rust-lang.org/std/primitive.str.html#method.split_whitespace

[ ]:
```
use std::collections::HashMap;

fn main() {
    let text = "hello world wonderful world";
    // TO DO (1) Create a new hashmap called "map"
    // TO DO (1) Iterate on words
    // TO DO (2) Count the words
    // TO DO (3) Display the map with counts for each word
}
main();
```

Solution:

[ ]:
```
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);
}
main();
```