Rust - Day 10

Strings

- Rust has only one string type in the core language, which is the string slice `str` that is usually seen in its borrowed form `&str`.
- String literals, for example, are stored in the program's binary and are therefore string slices.
- **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.**
- When Rustaceans refer to "strings" in Rust, they might be referring to either the `String` or the string slice `&str` types, not just one of those types.
- Both `String` and string slices are UTF-8 encoded

UTF-8

- UTF-8 is a variable-width encoding system for Unicode characters. This means:
  - Basic ASCII characters (0-127) take up 1 byte
  - Other Unicode characters can take 2, 3, or 4 bytes
- For example, in UTF-8:
  - "hello" takes 5 bytes (1 byte per character)
  - "привет" takes 12 bytes (2 bytes per character)
  - "你好" takes 6 bytes (3 bytes per character)
  - "👋" takes 4 bytes (4 bytes for the emoji)

String (contd)

- `String` is actually implemented as a wrapper around a vector of bytes with some extra guarantees, restrictions, and capabilities.

```
let mut s = String::new();
```

```
let data = "initial contents";

let s = data.to_string();

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

- We can also use the function `String::from` to create a `String` from a string literal.

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

- **`String::from` and `to_string` do the same thing, so which one you choose is a matter of style and readability.**
- Remember that strings are UTF-8 encoded, so we can include any properly encoded data in them

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שלום");
let hello = String::from("नमस्ते");
let hello = String::from("こんにちは");
let hello = String::from("안녕하세요");
let hello = String::from("你好");
let hello = String::from("Olá");
```

```
let hello = String::from("Здравствуйте");
let hello = String::from("Hola");
```

Updating a String

- A `String` can grow in size and its contents can change, just like the contents of a `Vec<T>`, if you push more data into it.
- In addition, you can conveniently use the `+` operator or the `format!` macro to concatenate `String` values.

Appending to a String with pust_str and push

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

- **The `push_str` method takes a string slice because we don't necessarily want to take ownership of the parameter.**

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

- If the `push_str` method took ownership of `s2`, we wouldn't be able to print its value on the last line. However, this code works as we'd expect!
- **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');
```

Using `+` Operator

```
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
```

- **The type of `&s2` is `&String`, not `&str`, as specified in the second parameter to `add`. So why does Listing 8-18 compile?**
- **The reason we're able to use `&s2` in the call to `add` is that the compiler can *coerce* the `&String` argument into a `&str`. When we call the `add` method, Rust uses a *deref coercion*, which here turns `&s2` into `&s2[..]`. We'll discuss deref coercion in more depth in Chapter 15. Because `add` does not take ownership of the `s` parameter, `s2` will still be a valid `String` after this operation.**

Using `format` Macro

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

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

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

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

Rust Strings don't support indexing

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

```
let hello = String::from("Hola");
```

- In this case, len will be 4, which means the vector storing the string "Hola" is 4 bytes long. Each of these letters takes one byte when encoded in UTF-8.

```
let hello = String::from("Здравствуйте");
```

- **If you were asked how long the string is, you might say 12. In fact, Rust's answer is 24: that's the number of bytes it takes to encode "Здравствуйте" in UTF-8, 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.**

Bytes and Scalar Values and Grapheme Clusters! Oh My!

- Another point about UTF-8 is that there are actually three relevant ways to look at strings from Rust's perspective: as bytes, scalar values, and grapheme clusters (the closest thing to what we would call *letters*).
- If we look at the Hindi word "नमस्ते" written in the Devanagari script, it is stored as a vector of `u8` values that looks like this:

```
[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164,
224, 165, 135]
```

- That's 18 bytes and is how computers ultimately store this data. If we look at them as Unicode scalar values, which are what Rust's `char` type is, those bytes look like this:

```
['न', 'म', 'स', '्', 'त', 'े']
```

- There are six `char` values here, but the fourth and sixth are not letters: they're diacritics that don't make sense on their own. Finally, if we look at them as grapheme clusters, we'd get what a person would call the four letters that make up the Hindi word:

```
["न", "म", "स्", "ते"]
```

- Rust provides different ways of interpreting the raw string data that computers store so that each program can choose the interpretation it needs, no matter what human language the data is in.
- **A final reason Rust doesn't allow us to index into a `String` to get a character is that indexing operations are expected to always take constant time (O(1)). But it isn't possible to guarantee that performance with a `String`, because Rust would have to walk through the contents from the beginning to the index to determine how many valid characters there were.**

Slicing Strings

- Indexing into a string is often a bad idea because it's not clear what the return type of the string-indexing operation should be: a byte value, a character, a grapheme cluster, or a string slice. If you really need to use indices to create string slices, therefore, Rust asks you to be more specific.
- Rather than indexing using `[]` with a single number, you can use `[]` with a range to create a string slice containing particular bytes:

```
let hello = "Здравствуйте";
let s = &hello[0..4];
```

- Here, `s` will be a `&str` that contains the first four bytes of the string. Earlier, we mentioned that each of these characters was two bytes, which means `s` will be `Зд`.
- **If we were to try to slice only part of a character's bytes with something like `&hello[0..1]`, Rust would panic at runtime in the same way as if an invalid index were accessed in a vector:**

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/collections`
thread 'main' panicked at src/main.rs:4:19:
byte index 1 is not a char boundary; it is inside '3' (bytes 0..2) of `Здравствуйте`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Methods for iterating over Strings

- The best way to operate on pieces of strings is to be explicit about whether you want characters or bytes.
- For individual Unicode scalar values, use the `chars` method. Calling `chars` on "Зд" separates out and returns two values of type `char`, and you can iterate over the result to access each element:

```
for c in "Зд".chars() {
    println!("{c}");
}
```

```
З
д
```

- Alternatively, the `bytes` method returns each raw byte, which might be appropriate for your domain:

```
for b in "Зд".bytes() {
    println!("{b}");
}
```

```
208
151
208
180
```

- **Getting grapheme clusters from strings, as with the Devanagari script, is complex, so this functionality is not provided by the standard library. Crates are available on crates.io if this is the functionality you need.**

Conclusion

- Rust has chosen to make the correct handling of `String` data the default behavior for all Rust programs, which means programmers have to put more thought into handling UTF-8 data up front.
- This trade-off exposes more of the complexity of strings than is apparent in other programming languages, but it prevents you from having to handle errors involving non-ASCII characters later in your development life cycle.

Hashmaps

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

- **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.**

Creating a New Hash Map

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

let mut scores = HashMap::new();

scores.insert(String::from("Red"), 10);
scores.insert(String::from("Blue"), 20);
```

Accessing Values in a Hash Map

```
let std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Red"), 10);
scores.insert(String::from("Blue"), 20);

let team_name = String::from("Blue");
let score = scores.get(&team_name).copied().unwrap_or(0);
```

- **The `get` method returns an `Option<&V>`; if there's no value for that key in the hash map, `get` will return `None`.**

Iterating over vectors

```
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 {
    println!("{key}: {value}");
}
```

```
Yellow: 50
Blue: 10
```

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**

```
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!
```

- **Rust 🦀 values all follow the ownership model, with two categories of assignment behavior: types implementing Copy (which duplicate their data) and types that don't (which transfer ownership**

without duplication)

- We aren't able to use the variables `field_name` and `field_value` after they've been moved into the hash map with the call to `insert`.
- If we insert references to values into the hash map, the values won't be moved into the hash map.

Updating HashMap

- Althought the number of key and value pairs is growable, each unique key can only have one value associated with it at a time.
- When you want to change the data in a hash map, you have to decide how to handle the case when a key already has a value assigned.

1. You could replace the old value with the new value, completely disregarding the old value.

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

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 20);

println!("{scores:?}"); // Printing in debug mode
```

> This code will print `{"Blue": 25}`. The original value of `10` has been overwritten.

2. You could keep the old value and ignore the new value, only adding the new value if the key *doesn't* already have a value. 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;

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:?}");
```

> 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, it inserts the parameter as the new value for this key and returns a mutable reference to the new value. This technique is much cleaner than writing the logic ourselves and, in addition, plays more nicely with the borrow checker.

3. Or you could combine the old value and the new value.

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

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:?}");
```

> This code will print `{"world": 2, "hello": 1, "wonderful": 1}`.

Hashing Functions

- By default, `HashMap` uses a hashing function called *SipHash* that can provide resistance to denial-of-service (DoS) attacks involving hash tables.
- This is not the fastest hashing algorithm available, but the trade-off for better security that comes with the drop in performance is worth it.
- If you profile your code and find that the default hash function is too slow for your purposes, you can switch to another function by specifying a different hasher.
- A *hasher* is a type that implements the `BuildHasher` trait.
- You don't necessarily have to implement your own hasher from scratch; crates.io has libraries shared by other Rust users that provide hashers implementing many common hashing algorithms.

Practice Questions - To be solved!

1. Given a list of integers, use a vector and return the median (when sorted, the value in the middle position) and mode (the value that occurs most often; a hash map will be helpful here) of the list.
2. Convert strings to pig latin. The first consonant of each word is moved to the end of the word and *ay* is added, so *first* becomes *irst-fay*. Words that start with a vowel have *hay* added to the end instead (*apple* becomes *apple-hay*). Keep in mind the details about UTF-8 encoding!
3. Using a hash map and vectors, create a text interface to allow a user to add employee names to a department in a company; for example, "Add Sally to Engineering" or "Add Amir to Sales." Then let the user retrieve a list of all people in a department or all people in the company by department, sorted alphabetically.