8. Common Collections

Collections

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

Unlike the built-in array and tuples types, collection data is heap-allocated.

Different collections have different pros/cons.

"To get this out of the way: you should probably just use `Vec` or `HashMap`"

Built-in Collections

Rust's collections can be grouped into four major categories:

- Sequences: `Vec`, `VecDeque`, `LinkedList`
- Maps: `HashMap`, `BTreeMap`
- Sets: `HashSet`, `BTreeSet`
- Misc: `BinaryHeap`

Not a "collection" per se: `String`.

Vec<T>

Vectors

Vectors let you store values of the same type in contiguous memory.

In other languages, they are sometimes known as `List` or `Array`.

(Rust already uses the term `array` for fixed-size primitive arrays.)

Creating a Vector

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

Note the type annotation. Rust is statically typed so it needs to know what is going into the `Vec`.

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

For initialising with default values, there is a `vec!` macro (warning: advanced feature). Rust infers that `v` is a `Vec<i32>`.

Updating a Vector

Add elements using `push()`.

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

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

Note: `mut` to make it mutable, and no type annotation because Rust can infer it in this case.

Dropping a Vector

Because there is no `free()` in Rust, vectors are freed when they go out of scope.

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

    // do stuff with v

} // <- v goes out of scope and is freed here
```

Reading from a Vector

There are two ways to read from a `Vector`.

1. Indexing syntax: get a reference
2. `get()` method: get an `Option<&T>`

Reading from a Vector

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

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

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

This is an

awesome

feature

Why this is 🎉

Rust has two ways so you can choose how your program behaves when you index a non-existent element.

1. Indexing syntax: dereference failure, panic
2. `get()` method: returns `None` so you can handle it

Ownership of Vector data

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

let first = &v[0];
v.push(6);

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

Compile error.

Why? Pushing to the vector might require alloc/copy, making `first` invalid.

Iteration

Rust makes you choose between a non-mutating **and a** mutating **loop.**

Iterating over a Vector

Non-mutating:

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

Iterating over a Vector

Mutating:

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

Storing values of multiple types

Vecs can only store items of the same type. Sometimes this is inconvenient. Use an `enum`!

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

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

String

String representations in Rust

— String slice `str`.

Defined in the language, used for string literals.

— `String`.

Defined in the standard library. Growable heap-allocated UTF-8.

— `OsString`, `OsStr`, `CString`, `CStr`.

Platform and C interop strings. Ignore until needed.

Creating a new String

```rust
let mut s1 = String::new();

let data = "initial contents"; // &'static str
let s2 = data.to_string();

// the method also works on a literal directly:
let s3 = "more contents".to_string();

let s4 = String::from("extra contents");
```

UTF-8 FTW

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hello");
let hello = String::from("שָׁלוֹם");
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");
```

Appending to a String

```rust
let mut s = String::from("foo");
s.push_str("bar"); // takes a string slice

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

Q:
What's a
character?

A:

Unicode Scalar Value

(four bytes)

(Not a UTF-8 codepoint)

String concatenation

```rust
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 string s3 will contain `"Hello, world!"` as a result of this code. The reason `s1` is no longer valid after the addition and the reason we used a reference to `s2` has to do with the signature of the method that gets called when we use the + operator..."

String concatenation (cont.)

"The + operator uses the `add()` method, whose signature looks something like this (generics removed):"

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

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

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
```

"First, `s2` has an &, meaning that we’re adding a reference of the second string to the first string because of the `s` parameter in the `add` function: we can only add a `&str` to a `String`; we can’t add two `String` values together. But wait — the type of `&s2` is `&String`, not `&str`, as specified in the second parameter to `add`. So why does it compile?"

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

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
```

"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."

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

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
```

"Second, we can see in the signature that add takes ownership of `self`, because `self` does not have an &. This means `s1` will be moved into the `add()` call and no longer be valid after that."

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

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
```

"So although `let s3 = s1 + &s2;` looks like it will copy both strings and create a new one, this statement actually takes ownership of `s1`, appends a copy of the contents of `s2`, and then returns ownership of the result."

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

let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2;
```

"In other words, it looks like it’s making a lot of copies but isn’t; the implementation is more efficient than copying."

Wow.

More concatenation

Use the `format!` macro.

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

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

Indexing into Strings

You cannot index like a `Vec`.

```
let s1 = String::from("hello");
let h = s1[0];
```

Outputs:

```
error[E0277]: the trait bound `std::string::String: std::ops::Index<{integer}>` is not satisfied
 -->
  |
3 |     let h = s1[0];
  |             ^^^^^ the type `std::string::String` cannot be indexed by `{integer}`
  |
  = help: the trait `std::ops::Index<{integer}>` is not implemented for `std::string::String`
```

Internal representation

`String` is a wrapper around a `Vec<u8>` of UTF-8 bytes.

```
let l1 = String::from("Hola").len(); // 4
let l2 = String::from("Здравствуйте").len(); // 24, not 12
```

Consider this broken code:

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

What should the value of answer be? The first byte of "З" in UTF-8 is `208`. Probably not what a user wants/expects.

Hindi word “नमस्ते”.

— UTF-8 `u8` byte vector:

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

— Unicode scalar values:

```
['न', 'म', 'स', X, 'त', X]
```

— Extended grapheme clusters:

```
["न", "म", "स्", "ते"]
```

The O(Kicker)

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 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. However, you can use [] with a range to create a string slice of bytes:

```
let hello = "Здравствуйте";

let s1 = &hello[0..4];
// Зд

let s2 = &hello[0..1];
// panicked at 'byte index 1 is not a char boundary;
// it is inside 'З' (bytes 0..2) of `Здравствуйте`'
```

Iterating Strings

```
for c in "नमस्ते".chars() {
    println!("{}", c); // 6 Unicode scalar values
}

for b in "नमस्ते".bytes() {
    println!("{}", b); // 18 u8 bytes
}
```

Extended grapheme clusters are not in the standard library.

Strings == complicated

"To summarize, strings are complicated. Different programming languages make different choices about how to present this complexity to the programmer. 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 upfront. 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."

HashMap<K, V>

HashMap basics

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

AKA: hash, map, object, hash table, dictionary, associative array...

`HashMap` in Rust uses the SipHash algorithm.

Creating HashMaps

```rust
use std::collections::HashMap; // not in the 'prelude' automatically

let mut scores = HashMap::new();

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

Strongly typed.

Keys all the same type. Values all the same type.

Creating HashMaps (cont.)

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

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();
```

Use the `zip()` method to create a vector of tuples. Then use the `collect()` method to turn that vector of tuples into a hash map.

HashMap ownership

Copy types like `i32` are copied into the map. Owned values like `String` are moved and the map becomes the owner:

```rust
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
```

HashMap lifetimes

"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. We’ll talk more about these issues in the “Validating References with Lifetimes” section in Chapter 10."

😨

Accessing values in HashMaps

```rust
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); // Some(&10)
```

Iterating HashMaps

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

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

for (key, value) in &scores { // non-mutable reference
    println!("{}: {}", key, value);
}
```

Overwriting a value

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

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

println!("{:?}", scores); // {"Blue": 25}
```

Only inserting if a key has no value

```rust
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); // {"Blue": 10, "Yellow": 50}
```

The return value of the `entry` method is an enum called `Entry` that represents a value that might or might not exist.

Q: What's wrong with `Option`?

Updating a value based on the old value

```rust
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); // {"wonderful": 1, "world": 2, "hello": 1}
```

`or_insert()` returns a `&mut V`, so we dereference with `*`.

Thank you

🙇