Fundamental Collections

Rust’s standard library includes a number of really useful data structures called collections. Most other data types represent one specific value, but collections 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. Each kind of collection has different capabilities and costs, and choosing an appropriate one for the situation you’re in is a skill you’ll develop over time. In this chapter, we’ll go over three collections which are used very often in Rust programs:

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

A string is a collection of characters. We’ve seen the String type before, but we’ll talk about it in depth now.

A hash map allows us to associate a value with a particular key.

There are more specialized variants of each of these data structures for particular situations, but these are the most fundamental and common. We’re going to discuss how to create and update each of the collections, as well as what makes each special.

Vectors

The first type we’ll look at is Vec<T>, also known as a vector. Vectors allow us 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. They are useful in situations where you have a list of items, such as the lines of text in a file or the prices of items in a shopping cart.

Creating a New Vector

To create a new, empty vector, we can call the Vec::new function:

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

Note that we added a type annotation here. Since we aren’t inserting any values into this vector, Rust doesn’t know what kind of elements we intend to store. This is an important point. Vectors are homogeneous: they may store many values, but those values must all be the same type. Vectors are implemented using generics, which Chapter 10 will cover how to use in your own types. For now, all you need to know is that the Vec type provided by the standard library can hold any type, and when a specific Vec holds a specific type, the type goes within angle brackets. We’ve told Rust that the Vec in v will hold elements of the i32 type.

In real code, Rust can infer the type of value we want to store once we insert values, so you rarely need to do this type annotation. It’s more common to create a Vec that has initial values, and Rust provides the vec! macro for convenience. The macro will create a new Vec that holds the values we give it. This will create a new Vec<i32> that holds the values 1, 2, and 3:

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

Because we’ve given initial i32 values, Rust can infer that the type of v is Vec<i32>, and the type annotation isn’t necessary. Let’s look at how to modify a vector next.

Updating a Vector

To create a vector then add elements to it, we can use the push method:

let mut v = Vec::new();

v.push(5);

v.push(6);

v.push(7);

v.push(8);

As with any variable as we discussed in Chapter 3, if we want to be able to change its value, we need to make it mutable with the mut keyword. The numbers we place inside are all i32s, and Rust infers this from the data, so we don’t need the Vec<i32> annotation.

Dropping a Vector Drops its Elements

Like any other struct, a vector will be freed when it goes out of scope:

{

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

// do stuff with v

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

When the vector gets dropped, all of its contents will also be dropped, meaning those integers it holds will be cleaned up. This may seem like a straightforward point, but can get a little more complicated once we start to introduce references to the elements of the vector. Let’s tackle that next!

Reading Elements of Vectors

Now that you know how to create, update, and destroy vectors, knowing how to read their contents is a good next step. There are two ways to reference a value stored in a vector. In the examples, we’ve annotated the types of the values that are returned from these functions for extra clarity.

This example shows both methods of accessing a value in a vector either with indexing syntax or the get method:

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

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

let third: Option<&i32> = v.get(2);

There are a few things to note here. First, that we use the index value of 2 to get the third element: vectors are indexed by number, starting at zero. Second, the two different ways to get the third element are: using & and [], which gives us a reference, or using the get method with the index passed as an argument, which gives us an Option<&T>.

The reason Rust has two ways to reference an element is so that you can choose how the program behaves when you try to use an index value that the vector doesn’t have an element for. As an example, what should a program do if it has a vector that holds five elements then tries to access an element at index 100 like this:

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

let does\_not\_exist = &v[100];

let does\_not\_exist = v.get(100);

When you run this, you will find that with the first [] method, Rust will cause a panic! when a non-existent element is referenced. This method would be preferable if you want your program to consider an attempt to access an element past the end of the vector to be a fatal error that should crash the program.

When the get method is passed an index that is outside the array, it will return None without causing a panic! call. You would use this if accessing an element beyond the range of the vector will happen occasionally under normal circumstances. Your code can then have logic to handle having either Some(&element) or None, as we discussed in Chapter 6. For example, the index could be coming from a person entering a number. If they accidentally enter a number that’s too large and your program gets a None value, you could tell the user how many items are in the current Vec and give them another chance to enter a valid value. That would be more user-friendly than crashing the program for a typo!

Invalid References

Once the program has a valid reference, the borrow checker will enforce the ownership and borrowing rules covered in Chapter 4 to ensure this reference and any other references to the contents of the vector stay valid. Recall the rule that says we can’t have mutable and immutable references in the same scope. That rule applies in this example, where we hold an immutable reference to the first element in a vector and try to add an element to the end:

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

let first = &v[0];

v.push(6);

Compiling this will give us this error:

error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable

|

4 | let first = &v[0];

| - immutable borrow occurs here

5 |

6 | v.push(6);

| ^ mutable borrow occurs here

7 | }

| - immutable borrow ends here

This code might look like it should work: why should a reference to the first element care about what changes about the end of the vector? The reason why this code isn’t allowed 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 over to the new space, in the circumstance that there isn’t enough room to put all the elements next to each other where the vector was. In that case, the reference to the first element would be pointing to deallocated memory. The borrowing rules prevent programs from ending up in that situation.

Note: For more on this, see The Nomicon at <https://doc.rust-lang.org/stable/nomicon/vec.html>.

Using an Enum to Store Multiple Types

At the beginning of this chapter, we said that vectors can only store values that are all the same type. This can be inconvenient; there are definitely use cases for needing to store a list of things of different types. Luckily, the variants of an enum are all 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!

For example, let’s say we want to get values from a row in a spreadsheet, where some of the columns in the row contain integers, some floating point numbers, and some strings. We can define an enum whose variants will hold the different value types, and then all of 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:

enum SpreadsheetCell {

Int(i32),

Float(f64),

Text(String),

}

let row = vec![

SpreadsheetCell::Int(3),

SpreadsheetCell::Text(String::from("blue")),

SpreadsheetCell::Float(10.12),

];

The reason Rust needs to know exactly what types will be in the vector at compile time is so that it knows exactly how much memory on the heap will be needed to store each element. A secondary advantage to this is that we can be explicit about what types are allowed in this vector. If Rust allowed a vector to hold any type, there would be a chance that one or more of the types would cause errors with the operations performed on the elements of the vector. Using an enum plus a match means that Rust will ensure at compile time that we always handle every possible case, as we discussed in Chapter 6.

If you don’t know at the time that you’re writing a program the exhaustive set of types the program will get at runtime to store in a vector, the enum technique won’t work. Instead, you can use a trait object, which we’ll cover in Chapter 17.

Now that we’ve gone over some of the most common ways to use vectors, be sure to take a look at the API documentation for all of the many useful methods defined on Vec by the standard library. For example, in addition to push there’s a pop method that will remove and return the last element. Let’s move on to the next collection type: String!

Strings

We’ve already talked about strings a bunch in Chapter 4, but let’s take a more in-depth look at them now. Strings are an area that new Rustaceans commonly get stuck on. This is due to a combination of three things: Rust’s propensity for making sure to expose possible errors, strings being a more complicated data structure than many programmers give them credit for, and UTF-8. These things combine in a way that can seem difficult when coming from other languages.

The reason strings are in the collections chapter is that strings are implemented as a collection of bytes plus some methods to provide useful functionality when those bytes are interpreted as text. In this section, we’ll talk about the operations on String that every collection type has, like creating, updating, and reading. We’ll also discuss the ways in which String is different than the other collections, namely how indexing into a String is complicated by the differences in which people and computers interpret String data.

What is a String?

Before we can dig into those aspects, we need to talk about what exactly we mean by the term string. Rust actually only has one string type in the core language itself: str, the string slice, which is usually seen in its borrowed form, &str. We talked about string slices in Chapter 4: these are a reference to some UTF-8 encoded string data stored elsewhere. String literals, for example, are stored in the binary output of the program, and are therefore string slices.

The type called String is provided in Rust’s standard library rather than coded into the core language, and is a growable, mutable, owned, UTF-8 encoded string type. When Rustaceans talk about “strings” in Rust, they usually mean both the String and the string slice &str types, not just one of those. This section is largely about String, but both these types are used heavily in Rust’s standard library. Both String and string slices are UTF-8 encoded.

Rust’s standard library also includes a number of other string types, such as OsString, OsStr, CString, and CStr. Library crates may provide even more options for storing string data. Similar to the \*String/\*Str naming, they often provide an owned and borrowed variant, just like String/&str. These string types may store different encodings or be represented in memory in a different way, for example. We won’t be talking about these other string types in this chapter; see their API documentation for more about how to use them and when each is appropriate.

Creating a New String

Many of the same operations available with Vec are available with String as well, starting with the new function to create a string, like so:

let s = String::new();

This creates a new empty string called s that we can then load data into.

Often, we’ll have some initial data that we’d like to start the string off with. For that, we use the to\_string method, which is available on any type that implements the Display trait, which string literals do:

let data = "initial contents";

let s = data.to\_string();

// the method also works on a literal directly:

let s = "initial contents".to\_string();

This creates a string containing initial contents.

We can also use the function String::from to create a String from a string literal. This is equivalent to using to\_string:

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

Because strings are used for so many things, there are many different generic APIs that can be used for strings, so there are a lot of options. Some of them can feel redundant, but they all have their place! In this case, String::from and .to\_string end up doing the exact same thing, so which you choose is a matter of style.

Remember that strings are UTF-8 encoded, so we can include any properly encoded data in them:

let hello = "السلام عليكم";

let hello = "Dobrý den";

let hello = "Hello";

let hello = "שָׁלוֹם";

let hello = "नमस्ते";

let hello = "こんにちは";

let hello = "안녕하세요";

let hello = "你好";

let hello = "Olá";

let hello = "Здравствуйте";

let hello = "Hola";

Updating a String

A String can grow in size and its contents can change just like the contents of a Vec, by pushing more data into it. In addition, String has concatenation operations implemented with the + operator for convenience.

Appending to a String with Push

We can grow a String by using the push\_str method to append a string slice:

let mut s = String::from("foo");

s.push\_str("bar");

s will contain “foobar” after these two lines. The push\_str method takes a string slice because we don’t necessarily want to take ownership of the parameter. For example, it would be unfortunate if we weren’t able to use s2 after appending its contents to s1:

let mut s1 = String::from("foo");

let s2 = String::from("bar");

s1.push\_str(&s2);

The push method is defined to have a single character as a parameter and add it to the String:

let mut s = String::from("lo");

s.push('l');

After this, s will contain “lol”.

Concatenation with the + Operator or the format! Macro

Often, we’ll want to combine two existing strings together. One way is to use the + operator like this:

let s1 = String::from("Hello, ");

let s2 = String::from("world!");

let s3 = s1 + &s2; // Note that s1 has been moved here and can no longer be used

After this code the String s3 will contain Hello, world!. The reason that s1 is no longer valid after the addition and the reason that we used a reference to s2 has to do with the signature of the method that gets called when we use the + operator. The + operator uses the add method, whose signature looks something like this:

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

This isn’t the exact signature that’s in the standard library; there add is defined using generics. Here, we’re looking at the signature of add with concrete types substituted for the generic ones, which is what happens when we call this method with String values. We'll be discussing generics in Chapter 10. This signature gives us the clues we need to understand the tricky bits of the + operator.

First of all, s2 has an &, meaning that we are adding a reference of the second string to the first string. This is 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. Remember back in Chapter 4 when we talked about how &String will coerce to &str: we write &s2 so that the String will coerce to the proper type, &str. Because this method does not take ownership of the parameter, s2 will still be valid after this operation.

Second, we can see in the signature that add takes ownership of self, because self does not have an &. This means s1 in the above example will be moved into the add call and no longer be valid after that. So while 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 s2’s contents, then returns ownership of the result. In other words, it looks like it’s making a lot of copies, but isn’t: the implementation is more efficient than copying.

If we need to concatenate multiple strings, the behavior of + gets unwieldy:

let s1 = String::from("tic");

let s2 = String::from("tac");

let s3 = String::from("toe");

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

s will be “tic-tac-toe” at this point. With all of the + and " characters, it gets hard to see what’s going on. For more complicated string combining, we can use the format! macro:

let s1 = String::from("tic");

let s2 = String::from("tac");

let s3 = String::from("toe");

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

This code will also set s to “tic-tac-toe”. 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. This version is much easier to read, and also does not take ownership of any of its parameters.

Indexing into Strings

In many other languages, accessing individual characters in a string by referencing them by index is a valid and common operation. In Rust, however, if we try to access parts of a String using indexing syntax, we’ll get an error. That is, this code:

let s1 = String::from("hello");

let h = s1[0];

will result in this error:

error: the trait bound `std::string::String: std::ops::Index<\_>` is not

satisfied [--explain E0277]

|>

|> let h = s1[0];

|> ^^^^^

note: the type `std::string::String` cannot be indexed by `\_`

The error and the note tell the story: Rust strings don’t support indexing. So the follow-up question is, why not? In order to answer that, we have to talk a bit about how Rust stores strings in memory.

Internal Representation

A String is a wrapper over a Vec<u8>. Let’s take a look at some of our properly-encoded UTF-8 example strings from before. First, this one:

let len = String::from("Hola").len();

In this case, len will be four, which means the Vec storing the string “Hola” is four bytes long: each of these letters takes one byte when encoded in UTF-8. What about this example, though?

let len = String::from("Здравствуйте").len();

A person asked how long the string is might say 12. However, Rust’s answer is 24. This is the number of bytes that it takes to encode “Здравствуйте” in UTF-8, since each Unicode scalar value takes two bytes of storage. Therefore, an index into the string’s bytes will not always correlate to a valid Unicode scalar value.

To demonstrate, consider this invalid Rust code:

let hello = "Здравствуйте";

let answer = &hello[0];

What should the value of answer be? Should it be З, the first letter? When encoded in UTF-8, the first byte of З is 208, and the second is 151, so answer should in fact be 208, but 208 is not a valid character on its own. Returning 208 is likely not what a person would want if they asked for the first letter of this string, but that’s the only data that Rust has at byte index 0. Returning the byte value is probably not what people want, even with only Latin letters: &"hello"[0] would return 104, not h. To avoid returning an unexpected value and causing bugs that might not be discovered immediately, Rust chooses to not compile this code at all and prevent misunderstandings earlier.

Bytes and Scalar Values and Grapheme Clusters! Oh my!

This leads to another point about UTF-8: there are really three relevant ways to look at strings, from Rust’s perspective: as bytes, scalar values, and grapheme clusters (the closest thing to what people would call letters).

If we look at the Hindi word “नमस्ते” written in the Devanagari script, it is ultimately stored as a Vec 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 this 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 does not allow you to index into a String to get a character is that indexing operations are expected to always take constant time (O(1)). It isn’t possible to guarantee that performance with a String, though, since Rust would have to walk through the contents from the beginning to the index to determine how many valid characters there were.

All of these problems mean that Rust does not implement [] for String, so we cannot directly do this.

Slicing Strings

However, indexing the bytes of a string is very useful, and is not expected to be fast. While we can’t use [] with a single number, we 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, so that means that s will be “Зд”.

What would happen if we did &hello[0..1]? The answer: it will panic at runtime, in the same way that accessing an invalid index in a vector does:

thread 'main' panicked at 'index 0 and/or 1 in `Здравствуйте` do not lie on

character boundary', ../src/libcore/str/mod.rs:1694

You should use this with caution, since it can cause your program to crash.

Methods for Iterating Over Strings

Luckily, there are other ways we can access elements in a String.

If we need to perform operations on individual Unicode scalar values, the best way to do so is to use the chars method. Calling chars on “नमस्ते” separates out and returns six values of type char, and you can iterate over the result in order to access each element:

for c in "नमस्ते".chars() {

println!("{}", c);

}

This code will print:

न

म

स

्

त

े

The bytes method returns each raw byte, which might be appropriate for your domain:

for b in "नमस्ते".bytes() {

println!("{}", b);

}

This code will print the 18 bytes that make up this String, starting with:

224

164

168

224

// ... etc

But make sure to remember that valid Unicode scalar values may be made up of more than one byte.

Getting grapheme clusters from Strings is complex, so this functionality is not provided by the standard library. There are crates available on crates.io if this is the functionality you need.

Strings are Not so Simple

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 does mean programmers have to put more thought into handling UTF-8 data upfront. This tradeoff exposes more of the complexity of strings than other programming languages do, but this will prevent you from having to handle errors involving non-ASCII characters later in your development lifecycle.

Let’s switch to something a bit less complex: hash map!

Hash Maps

The last of our fundamental collections is the 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. Many different programming languages support this kind of data structure, but often with a different name: hash, map, object, hash table, or associative array, just to name a few.

Hash maps are useful for when you want to be able to look up data not by an index, as you can with vectors, but by using a key that can be of any type. For example, in a game, you could keep track of each team’s score in a hash map where each key is a team’s name and the values are each team’s score. Given a team name, you can retrieve their score.

We’ll go over the basic API of hash maps in this chapter, but there are many more goodies hiding in the functions defined on HashMap by the standard library. As always, check the standard library documentation for more information.

Creating a New Hash Map

We can create an empty HashMap with new, and add elements with insert. Here we’re keeping track of the scores of two teams whose names are Blue and Yellow. The Blue team will start with 10 points and the Yellow team starts with 50:

use std::collections::HashMap;

let mut scores = HashMap::new();

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

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

Note that we need to first use the HashMap from the collections portion of the standard library. Of our three fundamental collections, this one is the least often used, so it’s not included in the features imported automatically in the prelude. Hash maps also have less support from the standard library; there’s no built-in macro to construct them, for example.

Just like vectors, hash maps store their data on the heap. This HashMap has keys of type String and values of type i32. 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.

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. The collect method gathers up data into a number of collection types, including HashMap. For example, if we had the team names and initial scores in two separate vectors, we can use the zip method to create a vector of tuples where “Blue” is paired with 10, and so forth. Then we can use the collect method to turn that vector of tuples into a HashMap:

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();

The type annotation HashMap<\_, \_> is needed here because it’s possible to collect into many different data structures, and Rust doesn’t know which you want unless you specify. For the type parameters for the key and value types, however, we use underscores and Rust can infer the types that the hash map contains based on the types of the data in the vector.

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

We would not be able to use the bindings field\_name and field\_value after they have been moved into the hash map with the call to insert.

If we insert references to values into the hash map, the values themselves will not 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, though. We will talk more about these issues in the Lifetimes section of Chapter 10.

Accessing Values in a Hash Map

We can get a value out of the hash map by providing its key to the get method:

use std::collections::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);

Here, score will have the value that’s associated with the Blue team, and the result will be Some(10). The result is wrapped in Some because get returns an Option<V>; if there’s no value for that key in the hash map, get will return None. The program will need to handle the Option in one of the ways that we covered in Chapter 6.

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;

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);

}

This will print each pair, in an arbitrary order:

Yellow: 50

Blue: 10

Updating a Hash Map

While the number of keys and values is growable, each individual key can only have one value associated with it at a time. When we want to change the data in a hash map, we have to decide how to handle the case when a key already has a value assigned. We could choose to replace the old value with the new value, completely disregarding the old value. We could choose to keep the old value and ignore the new value, and only add the new value if the key doesn’t already have a value. Or we could combine the old value and the new value. Let’s look at how to do each of these!

Overwriting a Value

If we insert a key and a value into a hash map, then insert that same key with a different value, the value associated with that key will be replaced. Even though this following code calls insert twice, the hash map will only contain one key/value pair because we’re inserting the value for the Blue team’s key both times:

use std::collections::HashMap;

let mut scores = HashMap::new();

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

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

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

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

Only Insert If the Key Has No Value

It’s common to want to check if a particular key has a value and, if it does not, insert a value for it. Hash maps have a special API for this, called entry, that takes the key we want to check as an argument. The return value of the entry function is an enum, Entry, that represents a value that might or might not exist. Let’s say that we want to check if the key for the Yellow team has a value associated with it. If it doesn’t, we want to insert the value 50, and the same for the Blue team. With the entry API, the code for this looks like:

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 returns the value for the corresponding Entry key if it exists, and if not, inserts its argument as the new value for this key and returns the modified Entry. This is much cleaner than writing the logic ourselves, and in addition, plays more nicely with the borrow checker.

This code will print {"Yellow": 50, "Blue": 10}. The first call to entry will insert the key for the Yellow team with the value 50, since the Yellow team doesn’t have a value already. The second call to entry will not change the hash map since the Blue team already has the value 10.

Update a Value Based on the Old Value

Another common use case for hash maps is to look up a key’s value then update it, based on the old value. For instance, if we wanted to count how many times each word appeared in some text, we could use a hash map with the words as keys and increment the value to keep track of how many times we’ve seen that word. If this is the first time we’ve seen a word, we’ll first insert the value 0.

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 will print {"world": 2, "hello": 1, "wonderful": 1}. The or\_insert method actually returns a mutable reference (&mut V) to the value for this key. Here we store that mutable reference in the count variable, so in order to assign to that value we must first dereference count using the asterisk (\*). The mutable reference goes out of scope at the end of the for loop, so all of these changes are safe and allowed by the borrowing rules.

Hashing Function

By default, HashMap uses a cryptographically secure hashing function that can provide resistance to Denial of Service (DoS) attacks. This is not the fastest hashing algorithm out there, but the tradeoff 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. We’ll be talking about traits and how to implement them in Chapter 10. You don't necessarily have to implement your own hasher from scratch; crates.io has libraries that others have shared that provide hashers implementing many common hashing algorithms.

Summary

Vectors, strings, and hash maps will take you far in programs where you need to store, access, and modify data. Here are some exercises you should now be equipped to solve:

Given a list of integers, use a vector and return the mean (average), 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.

Convert strings to Pig Latin, where the first consonant of each word is moved to the end of the word with an added “ay”, so “first” becomes “irst-fay”. Words that start with a vowel get “hay” added to the end instead (“apple” becomes “apple-hay”). Remember about UTF-8 encoding!

Using a hash map and vectors, create a text interface to allow a user to add employee names to a department in the 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.

The standard library API documentation describes methods these types have that will be helpful for these exercises!

We’re getting into more complex programs where operations can fail, which means it’s a perfect time to go over error handling next!