Common Collections

Rust's standard library includes a number of very 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 your current situation is a skill you'll develop over time. In this chapter, we'll discuss three collections that are used very often in Rust programs:

IMP

- 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 specific key. It's a particular implementation of the more general data structure called a *map*.

To learn about the other kinds of collections provided by the standard library, see the documentation.

We'll discuss how to create and update vectors, strings, and hash maps, as well as what makes each special.

Storing Lists of Values with Vectors

The first collection type we'll look at is `Vec<T>`, also known as a *vector*. 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. They are useful when 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 call the `Vec::new` function, as shown in Listing 8-1.

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

Listing 8-1: Creating a new, empty vector to hold values of type `i32`

Note that we added a type annotation here. Because 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 implemented using generics; we'll cover how to use generics with your own types in Chapter 10. For now, know that the `Vec<T>` type provided by the standard library can hold any type. When we create a vector to hold a specific type, we can specify the type within angle brackets. In Listing 8-1, we've told Rust that the `Vec<T>` in `v` will hold elements of the `i32` type.

More often, you'll create a `Vec<T>` with initial values and Rust will infer the type of value you want to store, so you rarely need to do this type annotation. Rust conveniently provides the `vec!` macro, which will create a new vector that holds the values you give it. Listing 8-2 creates a new `Vec<i32>` that holds the values `1`, `2`, and `3`. The integer type is `i32` because that's the default integer type, as we discussed in the "Data Types" section of Chapter 3.

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

Listing 8-2: Creating a new vector containing values

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. Next, we'll look at how to modify a vector.

Updating a Vector

To create a vector and then add elements to it, we can use the `push` method, as shown in Listing 8-3.

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

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

Listing 8-3: Using the `push` method to add values to a vector

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

Reading Elements of Vectors

There are two ways to reference a value stored in a vector: via indexing or by using the `get` method. In the following examples, we've annotated the types of the values that are returned from these functions for extra clarity.

Listing 8-4 shows both methods of accessing a value in a vector, with indexing syntax and the `get` method.

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

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

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {third}"),
    None => println!("There is no third element."),
}
```

Listing 8-4: Using indexing syntax and using the `get` method to access an item in a vector

Note a few details here. We use the index value of `2` to get the third element because vectors are indexed by number, starting at zero. Using `&` and `[]` gives us a reference to the element at the index value. When we use the `get` method with the index passed as an argument, we get an `Option<&T>` that we can use with `match`. IMP

Rust provides these two ways to reference an element so you can choose how the program behaves when you try to use an index value outside the range of existing elements. As an example, let's see what happens when we have a vector of five elements and then we try to access an element at index 100 with each technique, as shown in Listing 8-5.

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

let does_not_exist = &v[100];
let does_not_exist = v.get(100);
```

Listing 8-5: Attempting to access the element at index 100 in a vector containing five elements

When we run this code, the first `[]` method will cause the program to panic because 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 if accessing an element beyond the range of the vector may happen occasionally under normal circumstances. Your code will then have logic to handle having either `Some(&element)` or `None`, as 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 the program gets a `None` value, you could tell the user how many items are in the current vector and give them another chance to enter a valid value. That would be more user-friendly than crashing the program due to a typo!

When the program has a valid reference, the borrow checker enforces the ownership and borrowing rules (covered in Chapter 4) to ensure this reference and any other references to the contents of the vector remain valid. Recall the rule that states you can't have mutable and immutable references in the same scope. That rule applies in Listing 8-6, where we hold an immutable reference to the first element in a vector and try to add an element to the end. This program won't work if we also try to refer to that element later in the function.

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

let first = &v[0];

v.push(6);

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

Listing 8-6: Attempting to add an element to a vector while holding a reference to an item

Compiling this code will result in this error:

```
$ 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

For more information about this error, try `rustc --explain E0502`.
error: could not compile `collections` (bin "collections") due to 1 previous error
```

The code in Listing 8-6 might look like it should work: why should a reference to the first element care about changes at the end of the vector? This error is due to the way vectors work: because vectors put the values next to each other in memory, 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 is currently stored. 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 the implementation details of the `Vec<T>` type, see "The Rustonomicon".

Iterating Over the Values in a Vector

To access each element in a vector in turn, we would iterate through all of the elements rather than use indices to access one at a time. Listing 8-7 shows how to use a `for` loop to get immutable references to each element in a vector of `i32` values and print them.

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

Listing 8-7: Printing each element in a vector by iterating over the elements using a `for` loop

We can also iterate over mutable references to each element in a mutable vector in order to make changes to all the elements. The `for` loop in Listing 8-8 will add `50` to each element.

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

Listing 8-8: Iterating over mutable references to elements in a vector

To change the value that the mutable reference refers to, we have to use the `*` dereference operator to get to the value in `i` before we can use the `+=` operator. We'll talk more about the dereference operator in the "Following the Pointer to the Value with the Dereference Operator" section of Chapter 15.

Iterating over a vector, whether immutably or mutably, is safe because of the borrow checker's rules. If we attempted to insert or remove items in the `for` loop bodies in Listing 8-7 and Listing 8-8, we would get a compiler error similar to the one we got with the code in Listing 8-6. The reference to the vector that the `for` loop holds prevents simultaneous modification of the whole vector.

Using an Enum to Store Multiple Types Vector in combination with enum

Vectors can only store values that are of the same type. This can be inconvenient; there are definitely use cases for needing to store a list of items of different types. Fortunately, the variants of an enum are defined under the same enum type, so when we need one type to represent elements of different types, we can define and use an enum!

For example, say we want to get values from a row in a spreadsheet in which 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 all the enum variants will be considered the same type: that of the enum. Then we can create a vector to hold that enum and so, ultimately, hold different types. We've demonstrated this in Listing 8-9.

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

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

Listing 8-9: Defining an `enum` to store values of different types in one vector

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. We must also 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` expression means that Rust will ensure at compile time that every possible case is handled, as discussed in Chapter 6.

If you don't know the exhaustive set of types a 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 discussed some of the most common ways to use vectors, be sure to review the API documentation for all of the many useful methods defined on `Vec<T>` by the standard library. For example, in addition to `push`, a `pop` method removes and returns the last element.

Dropping a Vector Drops Its Elements

Like any other `struct`, a vector is freed when it goes out of scope, as annotated in Listing 8-10.

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

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

Listing 8-10: Showing where the vector and its elements are dropped

When the vector gets dropped, all of its contents are also dropped, meaning the integers it holds will be cleaned up. The borrow checker ensures that any references to contents of a vector are only used while the vector itself is valid. IMP

Let's move on to the next collection type: `String`!

Storing UTF-8 Encoded Text with Strings

We talked about strings in Chapter 4, but we'll look at them in more depth now. New Rustaceans commonly get stuck on strings for a combination of three reasons: Rust's propensity for exposing possible errors, strings being a more complicated data structure than many programmers give them credit for, and UTF-8. These factors combine in a way that can seem difficult when you're coming from other programming languages.

We discuss strings in the context of collections because 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, such as creating, updating, and reading. We'll also discuss the ways in which `String` is different from the other collections, namely how indexing into a `String` is complicated by the differences between how people and computers interpret `String` data.

What Is a String?

We'll first define what we mean by the term *string*. 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` . In Chapter 4, we talked about *string slices*, which are references to some UTF-8 encoded string data stored elsewhere. 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. Although this section is largely about `String` , both types are used heavily in Rust's standard library, and both `String` and string slices are UTF-8 encoded.

Creating a New String

Many of the same operations available with `Vec<T>` are available with `String` as well because `String` is actually implemented as a wrapper around a vector of bytes with some extra guarantees, restrictions, and capabilities. An example of a function that works the same way with `Vec<T>` and `String` is the `new` function to create an instance, shown in Listing 8-11.

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

Listing 8-11: Creating a new, empty `String`

This line creates a new, empty string called `s`, into which we can then load data. Often, we'll have some initial data with which we want to start the string. For that, we use the `to_string` method, which is available on any type that implements the `Display` trait, as string literals do. Listing 8-12 shows two examples.

```
let data = "initial contents";

let s = data.to_string();

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

Listing 8-12: Using the `to_string` method to create a `String` from a string literal

This code creates a string containing `initial contents`.

We can also use the function `String::from` to create a `String` from a string literal. The code in Listing 8-13 is equivalent to the code in Listing 8-12 that uses `to_string`.

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

Listing 8-13: Using the `String::from` function to create a `String` from a string literal

Because strings are used for so many things, we can use many different generic APIs for strings, providing us with a lot of options. Some of them can seem redundant, but they all have their place! In this case, `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, as shown in Listing 8-14.

```
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");
```

Listing 8-14: Storing greetings in different languages in strings

All of these are valid `String` values.

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 push_str and push

We can grow a `String` by using the `push_str` method to append a string slice, as shown in Listing 8-15.

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

Listing 8-15: Appending a string slice to a `String` using the `push_str` method

After these two lines, `s` will contain `foobar`. The `push_str` method takes a string slice because we don't necessarily want to take ownership of the parameter. For example, in the code in Listing 8-16, we want to be able to use `s2` after appending its contents to `s1`.

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

Listing 8-16: Using a string slice after appending its contents to a `String`

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`. Listing 8-17 adds the letter *l* to a `String` using the `push` method.

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

Listing 8-17: Adding one character to a `String` value using `push`

As a result, `s` will contain `lol`.

Concatenation with the + Operator or the `format!` Macro

Often, you'll want to combine two existing strings. One way to do so is to use the `+` operator, as shown in Listing 8-18.

```
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
```

Listing 8-18: Using the `+` operator to combine two `String` values into a new `String` value

The string `s3` will contain `Hello, world!`. 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's called when we use the `+` operator. The `+` operator uses the `add` method, whose signature looks something like this:

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

In the standard library, you'll see `add` defined using generics and associated types. Here, we've substituted in concrete types, which is what happens when we call this method with `String` values. We'll discuss generics in Chapter 10. This signature gives us the clues we need in order to understand the tricky bits of the `+` operator.

First, `s2` has an `&`, meaning that we're 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. But wait—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.

Second, we can see in the signature that `add` takes ownership of `self` because `self` does *not* have an `&`. This means `s1` in Listing 8-18 will be moved into the `add` call and will no longer be valid after that. 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. In other words, it looks like it's making a lot of copies, but it isn't; the implementation is more efficient than copying.

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

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

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

At this point, `s` will be `tic-tac-toe`. With all of the `+` and `"` characters, it's difficult to see what's going on. For combining strings in more complicated ways, we can instead 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 also sets `s` to `tic-tac-toe`. The `format!` macro works like `println!`, but instead of printing the output to the screen, it returns a `String` with the contents. The version of the code using `format!` is much easier to read, and the code generated by the `format!` macro uses references so that this call doesn't take ownership of any of its parameters.

Indexing into Strings

In many other programming languages, accessing individual characters in a string by referencing them by index is a valid and common operation. However, if you try to access parts of a `String` using indexing syntax in Rust, you'll get an error. Consider the invalid code in Listing 8-19.

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

Listing 8-19: Attempting to use indexing syntax with a String

This code will result in the following error:

```
$ cargo run
   Compiling collections v0.1.0 (file:///projects/collections)
error[E0277]: the type `str` cannot be indexed by `{integer}`
 --> src/main.rs:3:16
  |
3 |     let h = s1[0];
  |                ^ string indices are ranges of `usize`
  |
  = help: the trait `SliceIndex<str>` is not implemented for `{integer}`, which is
required by `String: Index<_>`
  = note: you can use `.chars().nth()` or `.bytes().nth()`
          for more information, see chapter 8 in The Book: <https://doc.rust-
lang.org/book/ch08-02-strings.html#indexing-into-strings>
  = help: the trait `SliceIndex<[_]>` is implemented for `usize`
  = help: for that trait implementation, expected `[_]`, found `str`
  = note: required for `String` to implement `Index<{integer}>`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `collections` (bin "collections") due to 1 previous error
```

The error and the note tell the story: Rust strings don't support indexing. But why not? To answer that question, we need to discuss how Rust stores strings in memory. IMP

Internal Representation

A `String` is a wrapper over a `Vec<u8>`. Let's look at some of our properly encoded UTF-8 example strings from Listing 8-14. First, this one:

```
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. The following line, however, may surprise you (note that this string begins with the capital Cyrillic letter *Ze*, not the number 3):

```
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. To demonstrate, consider this invalid Rust code:

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

You already know that `answer` will not be `3`, the first letter. When encoded in UTF-8, the first byte of `3` is `208` and the second is `151`, so it would seem that `answer` should in fact be `208`, but `208` is not a valid character on its own. Returning `208` is likely not what a user would want if they asked for the first letter of this string; however, that's the only data that Rust has at byte index 0. Users generally don't want the byte value returned, even if the string contains only Latin letters: if `&"hello"[0]` were valid code that returned the byte value, it would return `104`, not `h`.

The answer, then, is that to avoid returning an unexpected value and causing bugs that might not be discovered immediately, Rust doesn't compile this code at all and prevents misunderstandings early in the development process.

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 'З' (bytes 0..2) of
`Здравствуйте`
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

You should use caution when creating string slices with ranges, because doing so can crash your program.

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}");
}
```

This code will print the following:

```
З
д
```

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

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

This code will print the four bytes that make up this string:

```
208
151
208
180
```

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

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.

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 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.

The good news is that the standard library offers a lot of functionality built off the `String` and `&str` types to help handle these complex situations correctly. Be sure to check out the

documentation for useful methods like `contains` for searching in a string and `replace` for substituting parts of a string with another string.

Let's switch to something a bit less complex: hash maps!

Storing Keys with Associated Values in Hash Maps

The last of our common collections is the *hash map*. 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. Many programming languages support this kind of data structure, but they often use a different name, such as *hash*, *map*, *object*, *hash table*, *dictionary*, or *associative array*, just to name a few.

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. For example, in a game, you could keep track of each team's score in a hash map in which each key is a team's name and the values are each team's score. Given a team name, you can retrieve its score.

We'll go over the basic API of hash maps in this section, but many more goodies are hiding in the functions defined on `HashMap<K, V>` by the standard library. As always, check the standard library documentation for more information.

Creating a New Hash Map

One way to create an empty hash map is to use `new` and to add elements with `insert`. In Listing 8-20, we're keeping track of the scores of two teams whose names are *Blue* and *Yellow*. The Blue team starts 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);
```

Listing 8-20: Creating a new hash map and inserting some keys and values

Note that we need to first `use` the `HashMap` from the collections portion of the standard library. Of our three common collections, this one is the least often used, so it's not included in the features brought into scope 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.

Accessing Values in a Hash Map

We can get a value out of the hash map by providing its key to the `get` method, as shown in Listing 8-21.

```
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).copied().unwrap_or(0);
```

Listing 8-21: Accessing the score for the Blue team stored in the hash map

Here, `score` will have the value that's associated with the Blue team, and the result will be `10`. The `get` method returns an `Option<&V>`; if there's no value for that key in the hash map, `get` will return `None`. This program handles the `Option` by calling `copied` to get an `Option<i32>` rather than an `Option<&i32>`, then `unwrap_or` to set `score` to zero if `scores` doesn't have an entry for the key.

V IMP

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 code will print each pair in an arbitrary order:

```
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, as demonstrated in Listing 8-22. IMP

```
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!
```

Listing 8-22: Showing that keys and values are owned by the hash map once they're inserted

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. 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.

Updating a Hash Map

Although the number of key and value pairs is growable, each unique key can only have one value associated with it at a time (but not vice versa: for example, both the Blue team and the Yellow team could have the value `10` stored in the `scores` hash map).

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. You could replace the old value with the new value, completely disregarding the old value. 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. Or you 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 and then insert that same key with a different value, the value associated with that key will be replaced. Even though the code in Listing 8-23

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:?}");
```

Listing 8-23: Replacing a value stored with a particular key

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

Adding a Key and Value Only If a Key Isn't Present

It's common to check whether a particular key already exists in the hash map with a value and then to take the following actions: if the key does exist in the hash map, the existing value should remain the way it is; if the key doesn't exist, insert it and 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. Let's say we want to check whether 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. Using the `entry` API, the code looks like Listing 8-24.

```
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:?}");
```

similar to C++ insert

Listing 8-24: Using the `entry` method to only insert if the key does not already have a value

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.

Running the code in Listing 8-24 will print `{"Yellow": 50, "Blue": 10}`. The first call to `entry` will insert the key for the Yellow team with the value `50` because the Yellow team doesn't have a value already. The second call to `entry` will not change the hash map because the Blue team already has the value `10`.

Updating a Value Based on the Old 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. For instance, Listing 8-25 shows code that counts how many times each word appears in some text. We 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 it's 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:?}");
```

Listing 8-25: Counting occurrences of words using a hash map that stores words and counts

This code will print `{"world": 2, "hello": 1, "wonderful": 1}`. You might see the same key–value pairs printed in a different order: recall from the "Accessing Values in a Hash Map" section that iterating over a hash map happens in an arbitrary order.

The `split_whitespace` method returns an iterator over subslices, separated by whitespace, of the value in `text`. The `or_insert` method returns a mutable reference (`&mut V`) to the value for the specified 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 Functions

By default, `HashMap` uses a hashing function called *SipHash* that can provide resistance to denial-of-service (DoS) attacks involving hash tables[1]. 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. We'll talk 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 shared by other Rust users that provide hashers implementing many common hashing algorithms.

[1] https://en.wikipedia.org/wiki/SipHash

Summary

Vectors, strings, and hash maps will provide a large amount of functionality necessary in programs when you need to store, access, and modify data. Here are some exercises you should now be equipped to solve:

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.

The standard library API documentation describes methods that vectors, strings, and hash maps have that will be helpful for these exercises!

We're getting into more complex programs in which operations can fail, so it's a perfect time to discuss error handling. We'll do that next!