11
Exploring the Standard Library

Throughout this book, we have explained and used the concepts of Rust's Standard Library--the `std` crate. In this chapter, we will be discovering important standard modules that were not yet been covered (such as working with paths and files), and get a better grasp of working with collections.

We will discuss the following topics:

- Exploring `std` and the `prelude` module
- Collections - using hashmaps and hashsets
- Working with files
- Using Rust without the Standard Library

Exploring std and the prelude module

In previous chapters, we used built-in modules such as `str`, `vec`, and `io` from the `std` crate. The `std` crate contains many modules and functions that are used in real projects. For that reason, you won't see `extern crate std`, because `std` is imported by default in every other crate.

The small `prelude` module in `std` declares mostly traits (such as `Copy`, `Send`, `Sync`, `Drop`, `Clone`, `Eq`, `Ord`, and so on) and common types (such as `Option` and `Result`). For the same reason, the contents of the `prelude` module are imported by default in every module, as well as a number of standard macros (such as `println!`). That is the reason why we don't need to specify `Result::` before the `Ok` and `Err` variants in `match` branches.

The website `https://doc.rust-lang.org/std/` shows lists of the primitive types, modules, and macros contained in the standard library. We already discussed the most important macros from the standard library; see `Chapter 5`, *Higher-Order Functions and Error-Handling*, and `Chapter 8`, *Organizing Code and Macros*, in the *Some other built-in macros* section.

Collections - using hashmaps and hashsets

Often, you need to collect a large number of data items of a given type in one data structure. Until now, we have only worked with the `Vec` datatype to do this, but the `std::collections` module contains other implementations of sequences (such as `LinkedList`), maps (such as `HashMap`), and sets (such as `HashSet`). In most cases, you will only need `Vec` and `HashMap`; let's examine how to work with the latter.

A `HashMap <K,V>` is used when you want to store pairs consisting of a key of type `K` and a value of type `V`, both of generic type. `K` can be a Boolean, an integer, a string, or any other type that implements the `Eq` and `Hash` traits. A `HashMap` enables you to very quickly look up the value attached to a certain key using a hashing algorithm, and also because the data is cached. However, the keys of a `HashMap` are not sorted; if you need that, use a `BTreeMap`. `HashMap` is allocated on the heap, so it is dynamically resizable, just like a `Vec`.

Let's store monster names together with the planets they originate from:

```
// code from Chapter 11/code/hashmaps.rs:
use std::collections::HashMap;

fn main() {
    let mut monsters = HashMap::new();

    monsters.insert("Oron", "Uranus");
    monsters.insert("Cyclops", "Venus");
    monsters.insert("Rahav", "Neptune");
    monsters.insert("Homo Sapiens", "Earth");

    match monsters.get(&"Rahav") {
        Some(&planet) =>
println!("Rahav originates from: {}", planet),
        _ => println!("Planet of Rahav unknown."),
    }

    monsters.remove(&("Homo Sapiens"));

    // `HashMap::iter()` returns
    for (monster, planet) in monsters.iter() {
```

```
println!("Monster {} originates from planet {}", monster, planet);
    }
}
```

This program prints out:

```
Rahav originates from: Neptune
Monster Rahav originates from planet Neptune
Monster Cyclops originates from planet Venus
Monster Oron originates from planet Uranus
```

A new `HashMap` with default capacity is created with `HashMap::new()`. `insert` is used to put data into it, one pair at a time. If the inserted value is new, it returns `None`, otherwise, it returns `Some(value)`. To retrieve a value (here the planet on which a monster comes from), use the `get` method. This takes a reference to a key value and returns `Option<&V>`, so you have to use `match` to find the value. `remove` takes a key and removes the pair from the collection. The `iter()` method from `HashMap` returns an iterator that yields `(&key, &value)` pairs in arbitrary order.

If your requirement is to store unique values, such as unique monster names, `HashSet` guarantees that; it is in fact `HashMap` without values:

```
// code from Chapter 11/code/hashsets.rs:
use std::collections::HashSet;

fn main() {
    let mut m1: HashSet<&str> = vec!["Cyclops", "Raven",
"Gilgamesh"].into_iter().collect();
    let m2: HashSet<&str> = vec!["Moron", "Keshiu",
"Raven"].into_iter().collect();

    m1.insert("Moron");
    if m1.insert("Raven") {
        println!("New value added")
    }
    else {
        println!("This value is already present")
    }
    println!("m1: {:?}", m1);

println!("Intersection: {:?}", m1.intersection(&m2).collect::<Vec<_>>());
    println!("Union: {:?}", m1.union(&m2).collect::<Vec<_>>());
 println!("Difference: {:?}", m1.difference(&m2).collect::<Vec<_>>());
    println!("Symmetric Difference: {:?}",
             m1.symmetric_difference(&m2).collect::<Vec<_>>());
}
```

This program prints out:

```
This value is already present
m1: {"Raven", "Cyclops", "Gilgamesh", "Moron"}
Intersection: ["Raven", "Moron"]
Union: ["Raven", "Cyclops", "Gilgamesh", "Moron", "Keshiu"]
Difference: ["Cyclops", "Gilgamesh"]
Symmetric Difference: ["Cyclops", "Gilgamesh", "Keshiu"]
```

We can construct a `HashSet` from a `Vec` using `into_iter().collect()`. To add new values one by one, use `insert()`, which returns `true` if the value is new, or `false` if it is not. If the type of the set's elements implements the `Debug` trait, we can print them out with the `{:?}` format string. Methods for the intersection, union, difference, and symmetrical difference operations are also present.

Working with files

Now we show you how to do common operations with files, such as reading and writing, and making folders, and at the same time do proper error handling. The modules that contain these functionalities are `std::path`, which provides for cross platform file path manipulation, and `std::fs`.

Paths

File paths from the underlying file system are represented by the `Path` struct, which is created from a string slice containing the full path with `Path::new`. Suppose `hello.txt` is an existing file in the current directory; let's write some code to explore it:

```
// code from Chapter 11/code/paths.rs:use std::path::Path;

fn main() {
    let path = Path::new("hello.txt");
 let display = path.display();
    // test whether path exists:
    if path.exists() {
        println!("{} exists", display);
    }
    else {
        panic!("This path or file does not exist!");
    }

    let file = path.file_name().unwrap();
let extension = path.extension().unwrap();
let parent_dir = path.parent().unwrap();
    println!("This is file {:?} with extension {:?} in folder {:?}", file,
extension, parent_dir);

    // Check if the path is a file:
    if path.is_file() { println!("{} is a file", display);  }
    // Check if the path is a directory:
    if path.is_dir() { println!("{} is a directory", display); }
}
```

This code performs the following:

- Prints: `This is file "hello.txt" with extension "txt" in folder ""`.
- The `Path::new("e.txt").unwrap_or_else()` method creates the file `e.txt` or panics with an error.
- Tests whether a path is valid. Use the `exists()` method.
- Uses the `create_dir()` method to make folders.
- The `file_name()`, `extension()` and `parent()` methods return an Option value, so use `unwrap()` to get the value. The `is_file()` and `is_dir()` methods test respectively whether a path refers to a filename or to a directory.
- The `join()` method can be used to build paths, and it automatically uses the operating system-specific separator:

```
let new_path = path.join("abc").join("def");

// Convert the path into a string slice
match new_path.to_str() {
        None => panic!("new path is not a valid UTF-8 sequence"),
        Some(s) => println!("new path is {}", s),
      }
```

- This prints out the following on a Windows system:

```
 new path is hello.txt\abc\def
```

Reading a file

The `File` struct from `std::fs` represents a file that has been opened (it wraps a file descriptor), and gives read and/or write access to the underlying file.

Since many things can go wrong when doing file I/O, explicit and proactive handling of all possible errors is certainly needed. This can be done with pattern matching, because all the `File` methods return the `std::io::Result<T>` type, which is an alias for `Result<T, io::Error>`.

To open a file in read-only mode, use the static `File::open` method with a reference to its path, and match the file handler or a possible error like this:

```
// code from Chapter 11/code/read_file.rs:
use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;

fn main() {
    let path = Path::new("hello.txt");
    let display = path.display();

    let mut file = match File::open(&path) {
        Ok(file) => file,
        Err(why) => panic!("couldn't open {}: {}", display,
Error::description(&why))
    };
}
```

In the case of a failing open method (because the file does not exist or the program has no access) this prints out:

```
thread '<main>' panicked at 'couldn't open hello999.txt: os error',
F:\Rust\Rust book\Chapter 11 - Working with files\code\read_file.rs:11
```

Explicitly closing files is not necessary in Rust; the `file` handle goes out of scope at the end of main or the surrounding block, causing the file to automatically close.

We can read in the file in a `content` string by using the `read_to_string()` method, which again returns a `Result` value, necessitating a pattern match. In order to be able to use this, we need to do the import `use std::io::prelude::*`:

```
     let mut content = String::new();
        match file.read_to_string(&mut content) {
            Err(why) => panic!("couldn't read {}: {}",
    display, Error::description(&why)),
            Ok(_) => print!("{} contains:\n{}", display, content),
        }
```

This prints out:

```
hello.txt contains:
"Hello Rust World!"
```

`std::io::prelude` imports common I/O traits in our code, such as `Read`, `Write`, `BufRead`, and `Seek`.

Error-handling with try!

In order to program in a more structured way, we'll use the `try!` macro, which will propagate the error upwards, causing an early return from the function, or unpack the success value. When returning the result, it needs to be wrapped in either `Ok` to indicate success or `Err` to indicate failure. This is shown here:

```
// code from Chapter 11/code/read_file_try.rs:
use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;
use std::io;

fn main() {
    let path = Path::new("hello.txt");
    let display = path.display();

    let content = match read_file(path) {
         Err(why) => panic!("error reading {}: {}", display,
Error::description(&why)),
         Ok(content) => content
    };

    println!("{}", content);
}
```

```
fn read_file(path: &Path) -> Result<String, io::Error> {
let mut file = try!(File::open(path));
let mut buf = String::new();
try!(file.read_to_string(&mut buf));
Ok(buf)
}
```

This prints out:

```
"Hello Rust World!"
```

Buffered reading

Reading in a file in memory with `read_to_string` might not be that clever with large files, because that would use a large chunk of memory. In that case, it is much better to use the buffered reading provided by `BufReader` and `BufRead` from `std::io:`; this way lines are read in and processed one by one.

In the following example, a file with numerical information is processed. Each line contains two fields, an integer, and a float, separated by a space:

```
// code from Chapter 11/code/reading_text_file.rs:
use std::io::{BufRead, BufReader};
use std::fs::File;

fn main() {
    let file =
BufReader::new(File::open("numbers.txt").unwrap());
    let pairs: Vec<_> = file.lines().map(|line| {
        let line = line.unwrap();
        let line = line.trim();
        let mut words = line.split(" ");
let left = words.next().expect("Unexpected empty line!");
let right = words.next().expect("Expected number!");

(
left.parse::<u64>().ok().expect("Expected integer in first column!"),
right.parse::<f64>().ok().expect("Expected float in second column!")
)
}).collect();
     println!("{:?}", pairs);
}
```

This prints out:

```
[(120, 345.56), (125, 341.56)]
```

The information is collected in `pairs`, which is a `Vec<(u64, f64)>`, and which can then be processed as you want.

For a more complex line structure, you would want to work with struct values describing the line content instead of pairs, like this:

```
struct LineData {
    string1: String,
    int1 : i32,
    string2: String,
    // Some other fields
}
```

In production code, the `unwrap()` and `expect()` functions should be replaced by more robust code using pattern matching and/or `try!`.

Writing a file

To open a file in write-only mode, use the `File::create` static method with a reference to its path. This method also returns an `io::Result`, so we need to pattern match this value:

```
// code from Chapter 11/code/write_file.rs:
static CONTENT: &'static str =
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum.
";

use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;

fn main() {
    let path = Path::new("lorem_ipsum.txt");
    let display = path.display();

    let mut file = match File::create(&path) {
        Err(why) => panic!("couldn't create {}: {}",
                           display,
                           Error::description(&why)),
        Ok(file) => file,
    };
}
```

This command creates a new file or overwrites an existing one with that name, but it cannot create folders; if the path contains a subfolder, this must already exist. If you want to write a string (here a `static` string `CONTENT`) to that file, first convert the string with `as_bytes()` and then use the `write_all()` method, again matching the `Result` value:

```
    match file.write_all(CONTENT.as_bytes()) {
            Err(why) => {
                panic!("couldn't write to {}: {}",
                                display,
                                Error::description(&why))
            },
            Ok(_) => println!("successfully wrote to {}", display),
    }
```

This prints out:

```
successfully wrote to lorem_ipsum.txt
```

We need to write the string out as bytes; when writing a simple literal string like this to a file `file.write_all("line one\n");` we get `error: mismatched types: expected `&[u8]`, found `&'static str`(expected slice, found str) [E0308]`. To correct this, we must write (notice the b before the string for converting to bytes) this:

```
file.write_all(b"line one\n");
```

In fact, the `write_all` method continuously calls the `write` method until the buffer is written out completely. So we could just as well use `write`, and instead of a match we could also test on the successful outcome by checking `is_err`, like this:

```
if file.write(CONTENT.as_bytes()).is_err() {
println!("Failed to save response.");
return;
}
```

Instead of panicking in the case of an error, thus ending the program, we could also just display the error and return to the calling function, as in this variant:

```
let mut file = match File::create(&path) {
    Err(why) => { println!("couldn't create {}: {}",
                         display,
                         Error::description(&why));
                    return
                },
    Ok(file) => file,
};
```

Error-handling with try!

In the same way as we did for reading a file, we could also use the `try!` macro when writing to a file, like here:

```
// code from Chapter 11/code/write_file_try.rs:
use std::path::Path;
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;
use std::io;

struct Info {
    name: String,
    age: i32,
    rating: i32
}

impl Info {
    fn as_bytes(&self) -> &[u8] {
        self.name.as_bytes()
    }

    fn format(&self) -> String {
        format!("{};{};{}\n", self.name, self.age, self.rating)
    }
}

fn main() {
    let path = Path::new("info.txt");
    let display = path.display();

    let file = match write_file(&path) {
        Err(why) => panic!("couldn't write info to file {}: {}",
                           display,
                           Error::description(&why)),
        Ok(file) => file,
    };
}

fn write_file(path: &Path) -> Result<File, io::Error> {    let mut file =
try!(File::create(path));
    let info1 = Info { name:"Barak".to_string(), age: 56, rating: 8 };
    let info2 = Info { name:"Vladimir".to_string(), age: 55, rating: 6 };
    try!(file.write(info1.as_bytes()));
    try!(file.write(b"\r\n"));
    try!(file.write(info2.as_bytes()));
    Ok(file)
}
```

To open a file in other modes, use the `OpenOptions` struct from `std::fs`.

Filesystem operations

Here is an example that shows some info for all files in the current directory:

```
    // code from Chapter 11/code/read_files_in_dir.rs:
    use std::env;
    use std::fs;
    use std::error::Error;

    fn main() {
        show_dir().unwrap();
    }

    fn show_dir() -> Result<(), Box<Error>> {
        let here = try!(env::current_dir());
        println!("Contents in: {}", here.display());
        for entry in try!(fs::read_dir(&here)) {
            let path = try!(entry).path();
            let md = try!(fs::metadata(&path));        println!("  {} ({}
    bytes)", path.display(), md.len());
        }
        Ok(())
    }
```

This prints out:

```
Contents in: F:\Rust\Rust book\The Rust Programming Language\Chapter 11 -
Working with files\code F:\Rust\Rust book\The Rust Programming
Language\Chapter 11 - Working with files\code\read_file.rs (710 bytes)
F:\Rust\Rust book\The Rust Programming Language\Chapter 11 - Working with
files\code\read_files_in_dir.exe (2382143 bytes)
```

Here are some remarks to clarify the code:

- This code uses the `current_dir` method from `std::env`, reads that folder with `read_dir`, and iterates over all files it contains with `for entry in`
- To manipulate the filesystem, see `filesystem.rs` in the code download for examples, which demonstrates the use of the methods `create_dir`, `create_dir_all`, `remove_file`, `remove_dir`, and `read_dir`
- The `kind()` method is used here in the error case, to display the specific I/O error category (from the `std::io::ErrorKind` enum)
- The `fs::read_dir` method is used to read the contents of a directory, returning an `io::Result<Vec<Path>> paths`, which can be looped through with this code:

```
for path in paths {
    println!("> {:?}", path.unwrap().path());
}
```

Using Rust without the Standard Library

Rust is foremost a systems programming language and because the compiler can decide when a variable's lifetime ends, no garbage collection is needed for freeing memory. So when a Rust program executes, it runs in a very lightweight runtime, providing a heap, backtraces, stack guards, unwinding of the call stack when a panic occurs, and dynamic dispatching of methods on trait objects. Also, a small amount of initialization code is run before an executable project's `main` function starts up.

As we have seen, the standard library gives a lot of functionality. It offers support for various features of its host system: threads, networking, heap allocation, and more. It also links to its C equivalent, which also does some runtime initialization.

But Rust can also run on much more constrained systems that do not need (or do not have) this functionality. You can leave out the standard library from the compilation altogether by using the `#![no_std]` attribute at the start of the crate's code. In that case, Rust's runtime is roughly equivalent to that of C, and the size of the native code is greatly reduced.

When working without the standard library, the `core` crate and its `prelude` module are automatically made available to your code. The `core` library provides the minimal foundation needed for all Rust code to work. It comprises the basic primitive types, traits, and macros. For an overview, see; `https://doc.rust-lang.org/core/`.

At present, this only works for a library crate on the Rust stable version. Compiling an executable application without the standard library on Rust stable is much more involved; we refer you to `https://doc.rust-lang.org/unstable-book/language-features/lang-items.html; #using-libc` for a thorough and up-to-date discussion.

Summary

In this chapter, we showed you how to work with hashmaps and hashsets, two other important collection data structures. Then, we learned how to read and write files and explore the filesystem in Rust code. Finally, we looked at the Rust runtime, and how to make a project that runs without the standard library. We will conclude our overview of Rust's essentials in the next chapter by taking a closer look at Rust's ecosystem of crates.