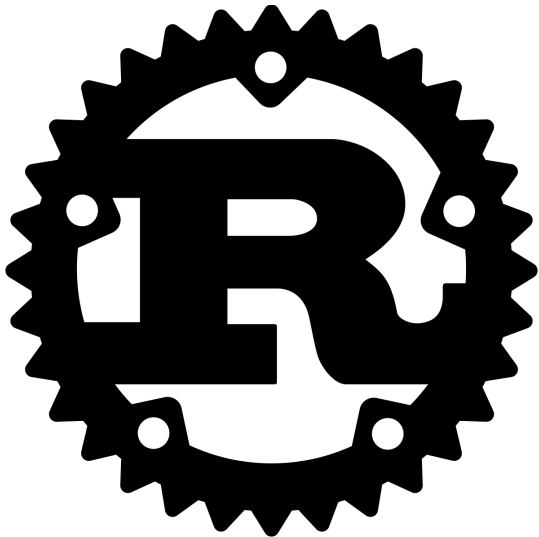


Getting Started with Rust: Working with Files and Doing File I/O

How to develop command-line utilities in Rust.

By Mihalīs Tsoukalos



This article demonstrates how to perform basic file and file I/O operations in Rust, and also introduces Rust's ownership concept and the Cargo tool. If you are seeing Rust code for the first time, this article should provide a pretty good idea of how Rust deals with files and file I/O, and if you've used Rust before, you still will appreciate the code examples in this article.

Ownership

It would be unfair to start talking about Rust without first discussing ownership. Ownership is the Rust way of the developer having control over the lifetime of a variable and the language in order to be safe. Ownership means that the passing of a variable also passes the ownership of the value to the new variable.

DEEP DIVE

Another Rust feature related to ownership is borrowing. Borrowing is about taking control over a variable for a while and then returning that ownership of the variable back. Although borrowing allows you to have multiple references to a variable, only one reference can be mutable at any given time.

Instead of continuing to talk theoretically about ownership and borrowing, let's look at a code example called `ownership.rs`:

```
fn main() {
    // Part 1
    let integer = 321;
    let mut _my_integer = integer;
    println!("integer is {}", integer);
    println!("_my_integer is {}", _my_integer);
    _my_integer = 124;
    println!("_my_integer is {}", _my_integer);

    // Part 2
    let a_vector = vec![1, 2, 3, 4, 5];
    let ref _a_correct_vector = a_vector;
    println!("_a_correct_vector is {:?}", _a_correct_vector);

    // Part 3
    let mut a_var = 3.14;
    {
        let b_var = &mut a_var;
        *b_var = 3.14159;
    }
    println!("a_var is now {}", a_var);
}
```

So, what's happening here? In the first part, you define an integer variable (`integer`) and create a mutable variable based on `integer`. Rust performs a

DEEP DIVE

full copy for primitive data types because they are cheaper, so in this case, the `integer` and `_my_integer` variables are independent from each other.

However, for other types, such as a vector, you aren't allowed to change a variable after you have assigned it to another variable. Additionally, you should use a reference for the `_a_correct_vector` variable of Part 2 in the above example, because Rust won't make a copy of `a_vector`.

The last part of the program is an example of borrowing. If you remove the curly braces, the code won't compile because you'll have two mutable variables (`a_var` and `b_var`) that point to the same memory location. The curly braces make `b_var` a local variable that references `a_var`, changes its value and returns the ownership back to `a_var` as soon as the end of the block is reached. As both `a_var` and `b_var` share the same memory address, any changes to `b_var` will affect `a_var` as well.

Executing `ownership.rs` creates the following output:

```
$ ./ownership
integer is 321
_my_integer is 321
_my_integer is 124
my_vector is [1, 2, 3, 4, 5]
a_var is now 3.14159
```

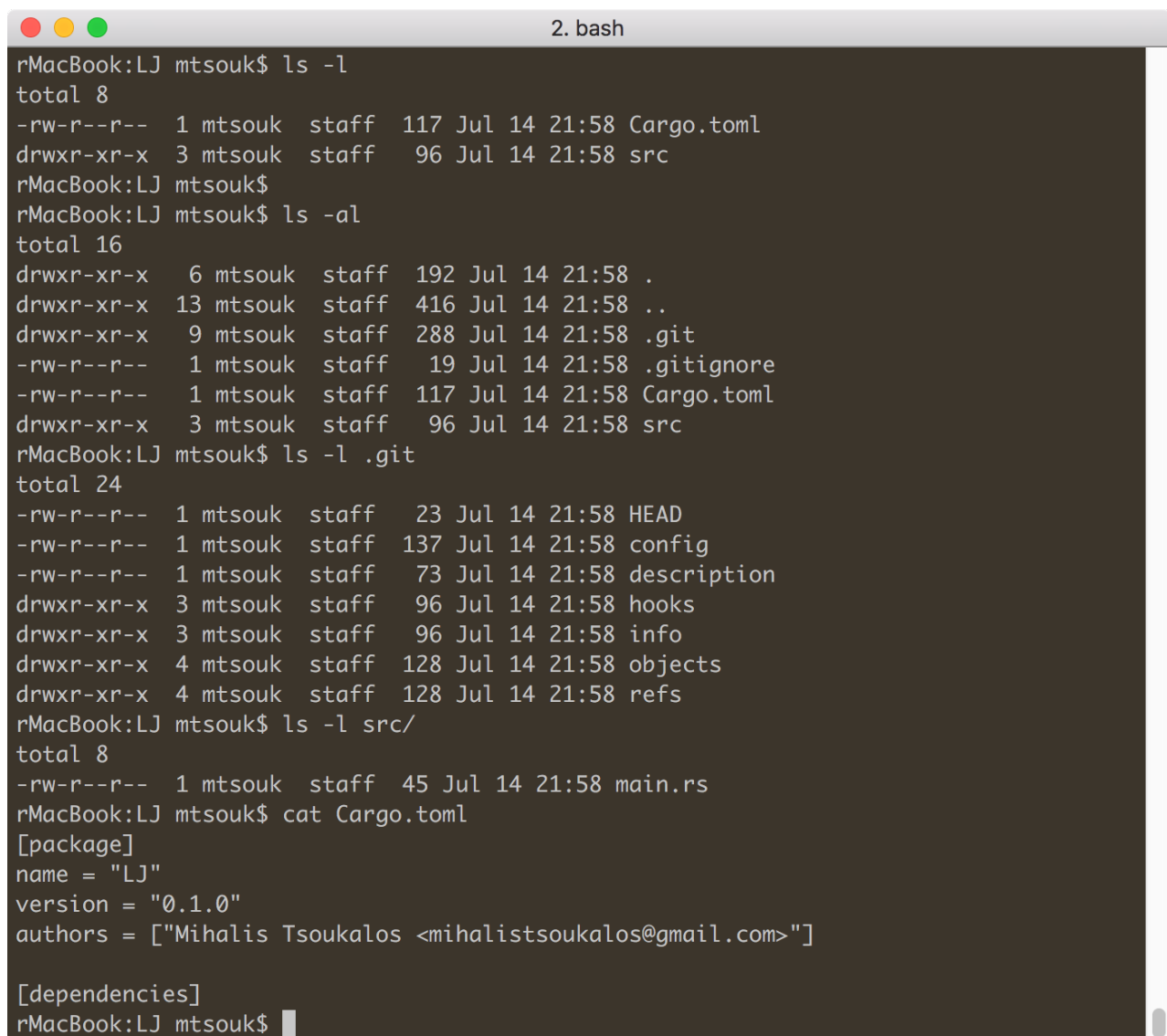
Notice that Rust catches mistakes related to ownership at compile time—it uses ownership to provide code safety.

The remaining Rust code shown in this article is pretty simple; you won't need to know about ownership to understand it, but it's good to have an idea of how Rust works and thinks.

The Cargo Tool

Cargo is the Rust package and compilation manager, and it's a useful tool for creating projects in Rust. In this section, I cover the basics of Cargo using a small example Rust project. The command for creating a Rust project named LJ with Cargo is `cargo new LJ --bin`.

The `--bin` command-line parameter tells Cargo that the outcome of the project will

A terminal window titled "2. bash" on a macOS system. The user is in a directory named "LJ". They run "ls -l" showing "Cargo.toml" and "src". Then they run "ls -al" showing the full directory listing including ".git", ".gitignore", and "Cargo.toml". Next, they run "ls -l .git" showing the contents of the git directory. Finally, they run "ls -l src/" showing "main.rs". They also run "cat Cargo.toml" which displays the project's metadata.

```
rMacBook:LJ mtsouk$ ls -l
total 8
-rw-r--r--  1 mtsouk  staff  117 Jul 14 21:58 Cargo.toml
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 src
rMacBook:LJ mtsouk$
rMacBook:LJ mtsouk$ ls -al
total 16
drwxr-xr-x  6 mtsouk  staff  192 Jul 14 21:58 .
drwxr-xr-x 13 mtsouk  staff  416 Jul 14 21:58 ..
drwxr-xr-x  9 mtsouk  staff  288 Jul 14 21:58 .git
-rw-r--r--  1 mtsouk  staff   19 Jul 14 21:58 .gitignore
-rw-r--r--  1 mtsouk  staff  117 Jul 14 21:58 Cargo.toml
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 src
rMacBook:LJ mtsouk$ ls -l .git
total 24
-rw-r--r--  1 mtsouk  staff   23 Jul 14 21:58 HEAD
-rw-r--r--  1 mtsouk  staff  137 Jul 14 21:58 config
-rw-r--r--  1 mtsouk  staff   73 Jul 14 21:58 description
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 hooks
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 info
drwxr-xr-x  4 mtsouk  staff  128 Jul 14 21:58 objects
drwxr-xr-x  4 mtsouk  staff  128 Jul 14 21:58 refs
rMacBook:LJ mtsouk$ ls -l src/
total 8
-rw-r--r--  1 mtsouk  staff   45 Jul 14 21:58 main.rs
rMacBook:LJ mtsouk$ cat Cargo.toml
[package]
name = "LJ"
version = "0.1.0"
authors = ["Mihalis Tsoukalos <mihalistsoukalos@gmail.com>"]

[dependencies]
rMacBook:LJ mtsouk$
```

Figure 1. Using Cargo to Create Rust Projects

DEEP DIVE

be an executable file, not a library. After that, you'll have a directory named LJ with the following contents:

```
$ cd LJ
$ ls -l
total 8
-rw-r--r--  1 mtsouk  staff  117 Jul 14 21:58 Cargo.toml
drwxr-xr-x  3 mtsouk  staff   96 Jul 14 21:58 src
$ ls -l src/
total 8
-rw-r--r--  1 mtsouk  staff   45 Jul 14 21:58 main.rs
```

Next, you'll typically want to edit one or both of the following files:

```
$ vi Cargo.toml
$ vi ./src/main.rs
```

Figure 1 shows all the files and directories of that minimal Cargo project as well as the contents of Cargo.toml.

Note that the Cargo.toml configuration file is where you declare the dependencies of your project as well as other metadata that Cargo needs in order to compile your project. To build your Rust project, issue the following command:

```
$ cargo build
```

You can find the debug version of the executable file in the following path:

```
$ ls -l target/debug/LJ
-rwxr-xr-x  2 mtsouk  staff 491316 Jul 14 22:02
↳target/debug/LJ
```

Clean up a Cargo project by executing **cargo clean**.

Readers and Writers

Rust uses readers and writers for reading and writing to files, respectively. A Rust reader is a value that you can read from; whereas a Rust writer is a value that you can write data to. There are various traits for readers and writers, but the standard ones are `std::io::Read` and `std::io::Write`, respectively. Similarly, the most common and generic ways for creating readers and writers are with the help of `std::fs::File::open()` and `std::fs::File::create()`, respectively. Note: `std::fs::File::open()` opens a file in read-only mode.

The following code, which is saved as `readWrite.rs`, showcases the use of Rust readers and writers:

```
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::create("/tmp/LJ.txt")?;
    let buffer = "Hello Linux Journal!\n";
    file.write_all(buffer.as_bytes())?;
    println!("Finish writing...");

    let mut input = File::open("/tmp/LJ.txt")?;
    let mut input_buffer = String::new();
    input.read_to_string(&mut input_buffer)?;
    print!("Read: {}", input_buffer);
    Ok(())
}
```

So, `readWrite.rs` first uses a writer to write a string to a file and then a reader to read the data from that file. Therefore, executing `readWrite.rs` creates the following output:

```
$ rustc readWrite.rs
$ ./readWrite
```

```
Finish writing...
Read: Hello Linux Journal!
$ cat /tmp/LJ.txt
Hello Linux Journal!
```

File Operations

Now let's look at how to delete and rename files in Rust using the code of `operations.rs`:

```
use std::fs;
use std::fs::File;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut file = File::create("/tmp/test.txt")?;
    let buffer = "Hello Linux Journal!\n";
    file.write_all(buffer.as_bytes())?;
    println!("Finish writing...");

    fs::rename("/tmp/test.txt", "/tmp/LJ.txt")?;
    fs::remove_file("/tmp/LJ.txt")?;
    println!("Finish deleting...");
    Ok(())
}
```

The Rust way to rename and delete files is straightforward, as each task requires the execution of a single function. Additionally, you can see that `/tmp/test.txt` is created using the technique found in [readWrite.rs](#). Compiling and executing `operations.rs` generates the following kind of output:

```
$ ./operations
Finish writing...
Finish deleting...
```

The code of `operations.rs` is far from complete, as there is no error-handling code in it. Please feel free to improve it!

Working with Command-Line Arguments

This section explains how to access and process the command-line arguments of a Rust program. The Rust code of `cla.rs` is the following:

```
use std::env;

fn main()
{
    let mut counter = 0;
    for argument in env::args()
    {
        counter = counter + 1;
        println!("{}", counter, argument);
    }
}
```

Let's look at what's happening in this example. First, it's using the `env` module of the `std` crate, because this is how to get the command-line arguments of your program, which will be kept in `env::args()`, which is an iterator over the arguments of the process. Then you iterate over those arguments using a `for` loop.

Say you want to add the command-line arguments of a program, the ones that are valid integers, in order to find their total. You can use the next `for` loop, which is included in the final version of `cla.rs`:

```
let mut sum = 0;
for input in env::args()
{
    let _i = match input.parse::<i32>() {
```



```
Ok(_i) => {
    sum = sum + _i
},
Err(_e) => {
    println!("{}", input)
}
};
}

println!("Sum: {}", sum);
```

Here you iterate over the `env::args()` iterator, but this time with a different purpose, which is finding the command-line arguments that are valid integers and summing them up.

If you are used to programming languages like C, Python or Go, you most likely will find the aforementioned code over-complicated for such a simple task, but that's the way Rust works. Additionally, `cla.rs` contains Rust code related to error-handling.

Note that you should compile `cla.rs` and create an executable file before running it, which means that Rust can't easily be used as a scripting programming language. So in this case, compiling and executing `cla.rs` with some command-line arguments creates this kind of output:

```
$ rustc cla.rs
$ ./cla 12 a -1 10
1: ./cla
2: 12
3: a
4: -1
5: 10
./cla: Not a valid integer!
a: Not a valid integer!
Sum: 21
```

Anyway, that's enough for now about the command-line arguments of a program. The next section describes using the three standard UNIX files.

Standard Input, Output and Error

This section shows how to use `stdin`, `stdout` and `stderr` in Rust. Every UNIX operating system has three files open all the time for its processes. Those three files are `/dev/stdin`, `/dev/stdout` and `/dev/stderr`, which you also can access using file descriptors 0, 1 and 2, respectively. UNIX programs write regular data to standard output and error messages to standard error while reading from standard input.

The following Rust code, which is saved as `std.rs`, reads data from standard input and writes to standard output and standard error:

```
use std::io::Write;
use std::io;

fn main() {
    println!("Please give me your name:");
    let mut input = String::new();
    match io::stdin().read_line(&mut input) {
        Ok(n) => {
            println!("{}", bytes read", n);
            print!("Your name is {}", input);
        }
        Err(error) => println!("error: {}", error),
    }

    let mut stderr = std::io::stderr();
    writeln!(&mut stderr, "This is an error message!").unwrap();
    eprintln!("That is another error message!")
}
```

Rust uses the `eprint` and `eprintln` macros for writing to standard error, which is a pretty handy approach. Alternatively, you can write your text to `std::io::stderr()`. Both techniques are illustrated in `std.rs`.

As you might expect, you can use the `print` and `println` macros for writing to standard output. Finally, you can read from standard input with the help of the `io::stdin().read_line()` function. Compiling and executing `std.rs` creates the following output:

```
$ rustc std.rs
$ ./std
Please give me your name:
Mihalis
8 bytes read
Your name is Mihalis
This is an error message!
That is another error message!
```

If you're using the Bash shell on your Linux machine, you can discard standard output or standard error data by redirecting them to `/dev/null`:

```
$ ./std 2>/dev/null
Please give me your name:
Mihalis
8 bytes read
Your name is Mihalis
$ ./std 2>/dev/null 1>/dev/null
Mihalis
```

The previous commands depend on the UNIX shell you are using and have nothing to do with Rust. Note that various other techniques exist for working with UNIX `stdin`, `stdout` and `stderr` in Rust.

Working with Plain-Text Files

Now let's look at how to read a plain-text file line by line, which is the most frequent way of processing plain-text files. At the end of the program, the total number of characters as well as the number of lines read will be printed on the screen—consider this as a simplified version of the **wc(1)** command-line utility.

The name of the Rust utility is **lineByLine.rs**, and its code is the following:

```
use std::env;
use std::io::{BufReader, BufRead};
use std::fs::File;

fn main() {
    let mut total_lines = 0;
    let mut total_chars = 0;
    let mut total_uni_chars = 0;

    let args: Vec<_> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} text_file", args[0]);
        return;
    }

    let input_path = ::std::env::args().nth(1).unwrap();
    let file = BufReader::new(File::open(&input_path).unwrap());
    for line in file.lines() {
        total_lines = total_lines + 1;
        let my_line = line.unwrap();
        total_chars = total_chars + my_line.len();
        total_uni_chars = total_uni_chars + my_line.chars().count();
    }
```

```
println!("Lines processed:\t\t{}", total_lines);
println!("Characters read:\t\t{}", total_chars);
println!("Unicode Characters read:\t{}", total_uni_chars);
}
```

The `lineByLine.rs` utility uses buffered reading as suggested by the use of `std::io::{BufReader, BufRead}`. The input file is opened using `BufReader::new()` and `File::open()`, and it's read using a `for` loop that keeps going as long as there is something to read from the input file.

Additionally, notice that the output of the `len()` function and the output of the `chars().count()` function might not be the same when dealing with text files that contain Unicode characters, which is the main reason for including both of them in `lineByLine.rs`. For an ASCII file, their output should be the same. Keep in mind that if what you want is to allocate a buffer to store a string, the `len()` function is the correct choice.

Compiling and executing `lineByLine.rs` using a plain-text file as input will generate this kind of output:

```
$ ./lineByLine lineByLine.rs
Lines processed:          28
Characters read:          756
Unicode Characters read:  756
```

Note that if you rename `total_lines` to `totalLines`, you'll most likely get the following warning message from the Rust compiler when trying to compile your code:

```
warning: variable 'totalLines' should have a snake case name
such as 'total_lines'
--> lineByLine.rs:7:6
|
```

```
7 |     let mut totalLines = 0;
  |           ^^^^^^^^^^^^^
  |
  |
= note: #[warn(non_snake_case)] on by default
```

You can turn off that warning message, but following the Rust way of defining variable names should be considered a good practice. (In a future Rust article, I'll cover more about text processing in Rust, so stay tuned.)

File Copy

Next let's look at how to copy a file in Rust. The `copy.rs` utility requires two command-line arguments, which are the filename of the source and the destination, respectively. The Rust code of `copy.rs` is the following:

```
use std::env;
use std::fs;

fn main()
{
    let args: Vec<_> = env::args().collect();
    if args.len() >= 3
    {
        let input = ::std::env::args().nth(1).unwrap();
        println!("input: {}", input);
        let output = ::std::env::args().nth(2).unwrap();
        println!("output: {}", output);
        match fs::copy(input, output)
        {
            Ok(n) => println!("{}", n),
            Err(err) => println!("Error: {}", err),
        };
    } else {
```

```
        println!("Not enough command line arguments")
    }
}
```

All the dirty work is done by the `fs::copy()` function, which is versatile, as you do not have to deal with opening a file for reading or writing, but it gives you no control over the process, which is a little bit like cheating. Other ways exist to copy a file, such as using a buffer for reading and writing in small byte chunks. If you execute `copy.rs`, you'll see output like this:

```
$ ./copy copy.rs /tmp/output
input: copy.rs
output: /tmp/output
515
```

You can use the handy `diff(1)` command-line utility for verifying that the copy of the file is identical to the original. (Using `diff(1)` is left as an exercise for the reader.)

UNIX File Permissions

This section describes how to find and print the UNIX file permissions of a file, which will be given as a command-line argument to the program using the `permissions.rs` Rust code:

```
use std::env;
use std::os::unix::fs::PermissionsExt;

fn main() -> std::io::Result<()> {
    let args: Vec<_> = env::args().collect();
    if args.len() < 2 {
        panic!("Usage: {} file", args[0]);
    }
    let f = ::std::env::args().nth(1).unwrap();
```

```
let metadata = try!(std::fs::metadata(f));
let perm = metadata.permissions();
println!("{:o}", perm.mode());
Ok(())
}
```

All the work is done by the `permissions()` function that's applied to the return value of `std::fs::metadata()`. Notice the `{:o}` format code in the `println()` macro, which indicates that the output should be printed in the octal system. Once again, the Rust code looks ugly at first, but you'll definitely get used to it after a while.

Executing `permissions.rs` produces the output like the following—the last three digits of the output is the data you want, where the remaining values have to do with the file type and the sticky bits of a file or directory:

```
$ ./permissions permissions
100755
$ ./permissions permissions.rs
100644
$ ./permissions /tmp/
41777
```

Note that `permissions.rs` works only on UNIX machines.

Conclusion

This article describes performing file input and output operations in Rust, as well as working with command-line arguments, UNIX permissions and using standard input, output and error. Due to space limitations, I couldn't present every technique for dealing with files and file I/O in Rust, but it should be clear that Rust is a great choice for creating system utilities of any kind, including tools that deal with files, directories and permissions, provided you have the time to learn its idiosyncrasies. At the end of the day though, developers should decide

for themselves whether they should use Rust or another systems programming language for creating UNIX command-line tools. ■

Mihalis Tsoukalos is a UNIX administrator and developer, a DBA and mathematician who enjoys technical writing. He is the author of *Go Systems Programming* and *Mastering Go*. You can reach him at <http://www.mtsoukalos.eu> and @mactsouk.

Resources

- [The Rust Programming Language](#)
- [Rust Documentation](#)
- [The Cargo Book](#)
- [Rust Crates](#)
- *Programming Rust* by Jim Blandy and Jason Orendorff, O'Reilly, 2017
- *The Rust Programming Language* by Steve Klabnik and Carol Nichols, No Starch Press, 2018

Send comments or feedback
via <http://www.linuxjournal.com/contact>
or email ljeditor@linuxjournal.com.