KU LEUVEN

Secure Software Lab 3
I/O and Parsing in Rust

Ruben Mechelinck
ruben.mechelinck@kuleuven.be

Stijn Volckaert
stijn.volckaert@kuleuven.be

March 2021

Setup (do this **before** the lab starts)

The Rust Programming Language

- We will again use the VM from the previous lab sessions
- Open a terminal, enter:
  - `sudo apt-get update && sudo apt-get upgrade && sudo apt-get install cmake libsdl2-dev`
  - `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
    - Proceed with default settings
  - `. ~/.profile`
- Test the installation: download sdl2_test.zip from Toledo and unzip it
  - `cd sdl2_test`
  - `cargo run`
  - A new window will open with a red 800x800 px square

Setup

- Recommended Editors:
  - Visual Studio Code + Rust language support plugin
    https://code.visualstudio.com/docs/setup/linux#_debian-and-ubuntu-based-distributions
    https://marketplace.visualstudio.com/items?itemName=rust-lang.rust
  - IntelliJ IDE + IntelliJ Rust plugin
    https://www.jetbrains.com/student/
    https://intellij-rust.github.io/
- For Windows (use the VM for the lab!!)
  - https://static.rust-lang.org/rustup/dist/x86_64-pc-windows-msvc/rustup-init.exe

Introduction

- Rust has **great documentation**!
  Available at https://doc.rust-lang.org/std/index.html (there is a dark theme)
- Rust is a great language for file I/O and parsing!
- The **File** struct represents a file that has been opened and gives read and/or write access to the underlying file.
- Since many things can go wrong when doing file I/O, all the **File** methods return the **io::Result<T>** type. This generic type is an alias for **Result<T, io::Error>**

Opening and creating files

Opening files is easy:

```
use std::error::Error;
use std::fs::File;
use std::path::Path;

fn main() {
  // Create a path to the desired file
  let path = Path::new("hello.txt");
  let display = path.display();

  // Open the path in read-only mode, returns `io::Result<File>`
  let mut file = match File::open(&path) {
    // The `description` method of `io::Error` returns a string that describes the error
    Err(why) => panic!("couldn't open {}: {}", display, why.description()),
    Ok(file) => file,
  };
  // `file` is dropped, so the "hello.txt" file gets closed
}
```

Opening and creating files

The file creation API looks almost exactly the same:

```
let path = Path::new("out/lorem_ipsum.txt");
let display = path.display();

// Open a file in write-only mode, returns `io::Result<File>`
let mut file = match File::create(&path) {
  Err(why) => panic!("couldn't create {}: {}", display, why.description()),
  Ok(file) => file,
};

// Write the `LOREM_IPSUM` string to `file`, returns `io::Result<()>`
match file.write_all(LOREM_IPSUM.as_bytes()) {
  Err(why) => { panic!("couldn't write to {}: {}", display, why.description()) },
  Ok(_) => println!("successfully wrote to {}", display),
}
```

Opening and creating files

You can also use the super convenient **OpenOptions** struct:

```
use std::fs::OpenOptions;

let file = OpenOptions::new()
  .read(true) // open for reading
  .write(true) // open for writing
  .create(true) // create the file if it doesn't exist
  .open("foo.txt"); // file name
```

For other options, see:

https://doc.rust-lang.org/std/fs/struct.OpenOptions.html

Using cursors for I/O

In this lab session, we'll be using **Cursor** objects to read and write from files:

```
let mut file = match File::open(&path) {
  Err(why) => panic!("Could not open file: {} (Reason: {})", display, why.description()),
  Ok(file) => file
};

// read the full file into memory. panic on failure
let mut raw_file = Vec::new();
file.read_to_end(&mut raw_file).unwrap();

// A Cursor wraps around an in-memory buffer and provides
// it with Seek, Read, and BufRead traits.
// This means that we can use the cursor object as if
// it was a file object.
// The only difference is that a cursor is much faster!
let mut cursor = Cursor::new(raw_file);
```

Using cursors for I/O

The **Cursor** struct implements all of the regular I/O traits, including **Read, BufRead**, **Seek**, and **Write**. This means we can call methods such as **read** on a cursor:

Overviews of other trait methods implemented for Cursor can be found here:

https://doc.rust-lang.org/std/io/trait.Read.html

https://doc.rust-lang.org/std/io/trait.BufRead.html

https://doc.rust-lang.org/std/io/trait.Write.html

https://doc.rust-lang.org/std/io/trait.Seek.html

```
let mut c: [u8; 1] = [0; 1];
// reads from the cursor into a slice
// containing u8 integers
cursor.read(&mut c)?;
// Now we can use the match operator to compare
// the contents of the slice with one of
// various patterns
match &c {
  b"\n" => break,
  _ => { result.push(c[0]); continue },
}
```

Assignment

- Write a parser that reads PPM images encoded in the P6 format
- Populate an image struct with pixel data read from the image file
- Use the show_image function to render the image to the screen


ANY
QUESTIONS?