Advanced Rust - Lab 6: Systems Programming in Rust

Lukáš Hozda

2025

Exercise 1: Interacting with libc (25 minutes)

Objective

Create a file management utility using direct libc bindings.

Instructions

Implement the following functions using the libc crate:

1. A function to check if a file exists
2. A function to get file permissions
3. A function to set file permissions
4. A function to read file contents into a buffer

Requirements

```rust
use libc::{self, c_char, c_int, size_t, mode_t};
use std::ffi::{CString, CStr};
use std::io::{self, Error, ErrorKind};

/// Checks if a file exists
pub fn file_exists(path: &str) -> bool {
    // TODO: Implement using libc::access
}

/// Gets file permissions
pub fn get_file_permissions(path: &str) -> io::Result<u32> {
    // TODO: Implement using libc::stat
}

/// Sets file permissions
pub fn set_file_permissions(path: &str, mode: u32) -> io::Result<()> {
    // TODO: Implement using libc::chmod
}

/// Reads file contents into a String
pub fn read_file_contents(path: &str) -> io::Result<String> {
    // TODO: Implement using libc::open, libc::read, and libc::close
}

/// Helper function to convert io::Error from errno
fn io_error_from_errno() -> io::Error {
```

```rust
    // TODO: Implement this helper
}

fn main() {
    let test_file = "test_file.txt";

    // Create a test file
    std::fs::write(test_file, "Hello, libc!").expect("Failed to write test file");

    // Check if file exists
    println!("File exists: {}", file_exists(test_file));

    // Get and print current permissions
    let perms = get_file_permissions(test_file).expect("Failed to get permissions");
    println!("Current permissions: {:o}", perms);

    // Set new permissions (read/write for owner only)
    let new_perms = 0o600;
    set_file_permissions(test_file, new_perms).expect("Failed to set permissions");

    // Verify new permissions
    let updated_perms = get_file_permissions(test_file).expect("Failed to get updated permissions");
    println!("Updated permissions: {:o}", updated_perms);
    assert_eq!(updated_perms, new_perms);

    // Read file contents
    let contents = read_file_contents(test_file).expect("Failed to read file");
    println!("File contents: {}", contents);

    // Clean up
    std::fs::remove_file(test_file).expect("Failed to remove test file");
}
```

Exercise 2: Foreign Function Interface (FFI) (25 minutes)

Objective

Write a simple C library and call it from Rust using FFI.

Instructions

1. Create a C file with simple functions
2. Set up a build script to compile the C code
3. Create Rust bindings to call the C functions
4. Test the FFI integration

Requirements

First, create a C file named ‘simple$_{\text{math}}$.c‘ with the following content:

```c
// simple_math.c
#include <stdio.h>

int add(int a, int b) {
```

```c
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}

void print_result(int result) {
    printf("Result from C: %d\n", result);
}

typedef struct {
    int x;
    int y;
} Point;

void print_point(Point p) {
    printf("Point from C: (%d, %d)\n", p.x, p.y);
}

Point create_point(int x, int y) {
    Point p = {x, y};
    return p;
}
```

Next, create a build script (build.rs) in your project root:

```rust
// build.rs
fn main() {
    // TODO: Use cc to compile the C file
}
```

Then, implement the Rust bindings:

```rust
use std::ffi::c_void;

// TODO: Define the Point struct in Rust to match the C struct

// TODO: Create extern "C" declarations for the C functions

fn main() {
    // Test the add function
    let a = 5;
    let b = 7;
    let sum = unsafe { add(a, b) };
    println!("From Rust: {} + {} = {}", a, b, sum);

    // Test the multiply function
    let product = unsafe { multiply(a, b) };
    println!("From Rust: {} * {} = {}", a, b, product);

    // Test the print_result function
    unsafe {
        print_result(sum);
        print_result(product);
    }
```

```
    // Test the Point struct and related functions
    let p1 = Point { x: 10, y: 20 };
    unsafe {
        print_point(p1);
    }

    let p2 = unsafe { create_point(30, 40) };
    println!("Point from Rust: ({}, {})", p2.x, p2.y);
}
```

Exercise 3: Self-Referential Structs (20 minutes)

Objective

Implement a safe self-referential struct using raw pointers and ManuallyDrop.

Instructions

Create a self-referential text parser struct that:

1. Holds both the text data and a pointer to a location within that data
2. Provides methods to navigate through the text
3. Ensures memory safety despite the self-references

Requirements

```
use std::mem::ManuallyDrop;
use std::ptr;

/// A self-referential text parser
pub struct TextParser {
    // TODO: Add necessary fields
    // - The text data
    // - A pointer to the current position within the text
}

impl TextParser {
    /// Creates a new TextParser with the given text
    pub fn new(text: String) -> Self {
        // TODO: Implement this function
        // - Set up the self-referential struct carefully
    }

    /// Returns the current position in the text
    pub fn position(&self) -> usize {
        // TODO: Implement this function
    }

    /// Returns the current character at the cursor
    pub fn current_char(&self) -> Option<char> {
        // TODO: Implement this function
    }
```

```rust
    /// Advances the cursor by one character
    pub fn advance(&mut self) -> bool {
        // TODO: Implement this function
        // Return true if advanced successfully, false if at the end
    }

    /// Resets the cursor to the beginning of the text
    pub fn reset(&mut self) {
        // TODO: Implement this function
    }

    /// Returns the remaining text from the current position
    pub fn remaining_text(&self) -> &str {
        // TODO: Implement this function
    }
}

// TODO: Implement Drop if necessary

fn main() {
    let mut parser = TextParser::new(String::from("Hello, world!"));

    // Print each character
    while let Some(c) = parser.current_char() {
        println!("Character at position {}: {}", parser.position(), c);
        if !parser.advance() {
            break;
        }
    }

    // Reset and print the remaining text at different positions
    parser.reset();
    println!("After reset: '{}'", parser.remaining_text());

    // Advance a few characters and check the remaining text
    for _ in 0..7 {
        parser.advance();
    }
    println!("After advancing 7 positions: '{}'", parser.remaining_text());
}
```