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 {
```

```
    // 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");
}
```

Questions to Consider

1. What safety considerations are important when using libc functions?
2. How does error handling differ between Rust's standard library and libc?
3. What are the advantages and disadvantages of using direct libc calls versus Rust's standard library?
4. How would you handle different file path encodings across operating systems?

Solution

```
use libc::{self, c_char, c_int, size_t, mode_t, stat, F_OK};
use std::ffi::{CString, CStr};
use std::io::{self, Error, ErrorKind};
use std::ptr;
use std::slice;
use std::str;

/// Checks if a file exists
pub fn file_exists(path: &str) -> bool {
    let c_path = match CString::new(path) {
        Ok(p) => p,
        Err(_) => return false, // Path contains null bytes
```

```
    };

    unsafe {
        // F_OK checks for existence
        libc::access(c_path.as_ptr(), F_OK) == 0
    }
}

/// Gets file permissions
pub fn get_file_permissions(path: &str) -> io::Result<u32> {
    let c_path = CString::new(path)?;
    let mut stat_buf: stat = unsafe { std::mem::zeroed() };

    let result = unsafe {
        libc::stat(c_path.as_ptr(), &mut stat_buf)
    };

    if result == 0 {
        // Extract the permission bits (S_IRWXU | S_IRWXG | S_IRWXO)
        Ok(stat_buf.st_mode as u32 & 0o777)
    } else {
        Err(io_error_from_errno())
    }
}

/// Sets file permissions
pub fn set_file_permissions(path: &str, mode: u32) -> io::Result<()> {
    let c_path = CString::new(path)?;

    let result = unsafe {
        libc::chmod(c_path.as_ptr(), mode as mode_t)
    };

    if result == 0 {
        Ok(())
    } else {
        Err(io_error_from_errno())
    }
}

/// Reads file contents into a String
pub fn read_file_contents(path: &str) -> io::Result<String> {
    let c_path = CString::new(path)?;

    // Open the file
    let fd = unsafe {
        libc::open(c_path.as_ptr(), libc::O_RDONLY)
    };

    if fd < 0 {
        return Err(io_error_from_errno());
    }

    // Ensure the file is closed when we're done
```

```
    struct FileHandle(c_int);
    impl Drop for FileHandle {
        fn drop(&mut self) {
            unsafe { libc::close(self.0); }
        }
    }
    let _file_handle = FileHandle(fd);

    // Get file size
    let mut stat_buf: stat = unsafe { std::mem::zeroed() };
    if unsafe { libc::fstat(fd, &mut stat_buf) } != 0 {
        return Err(io_error_from_errno());
    }

    let file_size = stat_buf.st_size as usize;
    let mut buffer = vec![0u8; file_size];

    // Read the file
    let bytes_read = unsafe {
        libc::read(fd, buffer.as_mut_ptr() as *mut libc::c_void, file_size)
    };

    if bytes_read < 0 {
        return Err(io_error_from_errno());
    }

    // Resize buffer to actual bytes read
    buffer.truncate(bytes_read as usize);

    // Convert to String
    match String::from_utf8(buffer) {
        Ok(s) => Ok(s),
        Err(_) => Err(io::Error::new(ErrorKind::InvalidData, "Invalid UTF-8 data")),
    }
}

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

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);
```

```rust
    // 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");
}
```

Explanation

This implementation demonstrates how to interact with the operating system using libc bindings:

1. **File Existence Check**: Uses ‘libc::access‘ with ‘F_{OK}‘ flag to check if a file exists. This is similar to how ‘std::path::Path::exists()‘ works but uses direct syscalls.
2. **File Permissions**:
   - ‘$get_{filepermissions}$‘ uses ‘libc::stat‘ to get file metadata and extracts the permission bits.
   - ‘$set_{filepermissions}$‘ uses ‘libc::chmod‘ to change the file permissions.
3. **File Reading**: ‘$read_{filecontents}$‘ demonstrates a complete file I/O operation:
   - Opens the file with ‘libc::open‘
   - Gets the file size with ‘libc::fstat‘
   - Reads the contents with ‘libc::read‘
   - Closes the file with ‘libc::close‘ (via RAII pattern)
   - Converts the raw bytes to a UTF-8 string
4. **Error Handling**: We use ‘$io_{errorfromerrno}()$‘ to convert C's ‘errno‘ to Rust's ‘io::Error‘, making error handling more idiomatic.
5. **Resource Management**: The ‘FileHandle‘ struct with a ‘Drop‘ implementation ensures that the file descriptor is properly closed even if an error occurs.
6. **String Handling**: We use ‘CString‘ to convert Rust strings to null-terminated C strings, which is required for libc functions.

Key safety considerations:

- All unsafe blocks are carefully contained and documented
- Resources are properly cleaned up with RAII patterns
- String conversions handle potential null bytes and UTF-8 validation
- Error cases are properly propagated with Rust's ‘Result‘ type

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) {
    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);
}
```

Questions to Consider

1. What memory layout considerations are important when passing structs between Rust and C?
2. Why is using ‘unsafe‘ necessary when calling foreign functions?
3. How would you handle strings when passing them between Rust and C?
4. What challenges might arise when implementing more complex FFI interfaces?

Solution

Build script (build.rs):

```
// build.rs
fn main() {
    // Compile the C code into a static library
    cc::Build::new()
        .file("simple_math.c")
        .compile("simple_math");
}
```

Rust code:

```
use std::ffi::c_void;
use std::os::raw::c_int;

// Define the Point struct to match the C struct
// We use #[repr(C)] to ensure the same memory layout as C
#[repr(C)]
#[derive(Debug, Clone, Copy)]
pub struct Point {
    pub x: c_int,
    pub y: c_int,
```

```rust
}

// Create extern "C" declarations for the C functions
extern "C" {
    // Basic math functions
    pub fn add(a: c_int, b: c_int) -> c_int;
    pub fn multiply(a: c_int, b: c_int) -> c_int;
    pub fn print_result(result: c_int);

    // Functions that work with the Point struct
    pub fn print_point(p: Point);
    pub fn create_point(x: c_int, y: c_int) -> Point;
}

// A more complex example: String handling
pub fn print_string_from_rust(s: &str) {
    use std::ffi::CString;

    // Convert Rust string to C string
    let c_string = match CString::new(s) {
        Ok(s) => s,
        Err(_) => return, // String contains null bytes
    };

    unsafe {
        // This would require a corresponding C function:
        // void print_string(const char* s);
        // print_string(c_string.as_ptr());

        // For demonstration, we'll use printf directly:
        libc::printf(b"String from Rust: %s\n\0".as_ptr() as *const i8,
                     c_string.as_ptr());
    }
}

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);

    // Demonstrate string handling
    print_string_from_rust("Hello from Rust!");

    // Demonstrate error handling from C functions
    // (This would require additional C functions that can fail)
    fn call_c_with_error_handling() -> Result<i32, String> {
        unsafe {
            // Example of a C function that returns negative on error:
            // int risky_operation();
            let result = add(1, 2); // Placeholder for a function that can fail

            if result < 0 {
                Err(format!("C function failed with error code: {}", result))
            } else {
                Ok(result)
            }
        }
    }

    match call_c_with_error_handling() {
        Ok(value) => println!("C function succeeded with value: {}", value),
        Err(e) => eprintln!("Error: {}", e),
    }
}
```

Explanation

This implementation demonstrates how to create a Foreign Function Interface (FFI) between Rust and C:

1. **Build Script**: The ‘build.rs‘ file uses the ‘cc‘ crate to compile the C code into a static library that will be linked with the Rust binary.
2. **Type Mapping**:
   - We use ‘#[repr(C)]‘ on the ‘Point‘ struct to ensure it has the same memory layout as the C struct.
   - We use ‘std::os::raw::c_{int}‘ to match C's ‘int‘ type, ensuring correct ABI compatibility.
3. **Function Declarations**:
   - The ‘extern ”C”‘ block declares the C functions with their correct signatures.
   - The ‘unsafe‘ keyword is required when calling these functions because Rust can't verify their safety guarantees.
4. **String Handling**:
   - The ‘$print_{stringfromrust}$‘ function demonstrates how to convert a Rust string to a C-compatible null-terminated string.
   - We use ‘CString‘ to handle this conversion safely, checking for null bytes.
5. **Error Handling**:

- The ‘call_cwitherrorhandling‘ function shows how to wrap C functions that indicate errors through return values into Rust's ‘Result‘ type.
- This makes C functions more idiomatic in Rust code.

Key considerations for FFI:

- Memory layout: Rust structs must have ‘#[repr(C)]‘ to ensure compatibility
- Type mapping: Using the correct types to match C's ABI
- String handling: Proper conversion between Rust and C strings
- Error handling: Converting C's error reporting to Rust's ‘Result‘ type
- Safety: All FFI calls must be marked as ‘unsafe‘ since Rust can't verify the safety of foreign code

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

```rust
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
    }
```

```
    /// 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());
}
```

Questions to Consider

1. Why are self-referential structs challenging in Rust?
2. What would happen if we tried to implement this without raw pointers?
3. How does ManuallyDrop help with self-referential structs?
4. What problems might arise if the self-referential struct is moved in memory?

Solution

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

```rust
/// A self-referential text parser
pub struct TextParser {
    // We use ManuallyDrop to prevent dropping the String while we still have a pointer to it
    text: ManuallyDrop<String>,
    // Pointer to the current position in the text
    current_ptr: *const u8,
    // Start of the text for position calculations
    text_ptr: *const u8,
}

impl TextParser {
    /// Creates a new TextParser with the given text
    pub fn new(text: String) -> Self {
        // We need to be careful here to set up the self-reference correctly

        // First, get the pointer to the text's bytes
        let text_ptr = text.as_ptr();

        // Create the struct with the initial values
        let mut parser = TextParser {
            text: ManuallyDrop::new(text),
            current_ptr: ptr::null(),
            text_ptr,
        };

        // Now that we have the struct, set the current_ptr to the beginning of the text
        parser.current_ptr = parser.text.as_ptr();

        parser
    }

    /// Returns the current position in the text (byte offset)
    pub fn position(&self) -> usize {
        // Calculate the byte offset between current_ptr and text_ptr
        unsafe {
            self.current_ptr.offset_from(self.text_ptr) as usize
        }
    }

    /// Returns the current character at the cursor
    pub fn current_char(&self) -> Option<char> {
        if self.position() >= self.text.len() {
            return None;
        }

        // Get the character at the current position
        let remaining = self.remaining_text();
        remaining.chars().next()
    }

    /// Advances the cursor by one character
    pub fn advance(&mut self) -> bool {
        if self.position() >= self.text.len() {
            return false;
```

```rust
        }

        // Get the current character and its byte length
        let c = self.current_char().unwrap();
        let char_len = c.len_utf8();

        // Advance the pointer by the character's byte length
        unsafe {
            self.current_ptr = self.current_ptr.add(char_len);
        }

        true
    }

    /// Resets the cursor to the beginning of the text
    pub fn reset(&mut self) {
        self.current_ptr = self.text.as_ptr();
    }

    /// Returns the remaining text from the current position
    pub fn remaining_text(&self) -> &str {
        let text_len = self.text.len();
        let pos = self.position();

        if pos >= text_len {
            return "";
        }

        // Create a slice from the current position to the end
        unsafe {
            let remaining_len = text_len - pos;
            let slice = std::slice::from_raw_parts(self.current_ptr, remaining_len);
            std::str::from_utf8_unchecked(slice)
        }
    }
}

impl Drop for TextParser {
    fn drop(&mut self) {
        // Manually drop the String when the parser is dropped
        unsafe {
            ManuallyDrop::drop(&mut self.text);
        }
    }
}

// Implementation that handles moving the struct correctly
impl Clone for TextParser {
    fn clone(&self) -> Self {
        // Create a new parser with a clone of the text
        let mut new_parser = TextParser::new(ManuallyDrop::into_inner(self.text.clone()));

        // Set the position to match the original
        let position = self.position();
```

```
        if position > 0 {
            // Position the cursor at the same place
            unsafe {
                new_parser.current_ptr = new_parser.text_ptr.add(position);
            }
        }

        new_parser
    }
}

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());

    // Demonstrate that cloning works correctly with the self-reference
    let clone = parser.clone();
    println!("Cloned parser remaining text: '{}'", clone.remaining_text());

    // Demonstrate what happens if we create a new string with the same content
    let same_content = String::from("Hello, world!");
    let mut parser2 = TextParser::new(same_content);
    for _ in 0..7 {
        parser2.advance();
    }
    println!("Second parser remaining text: '{}'", parser2.remaining_text());
}
```

Explanation

This implementation demonstrates how to create a safe self-referential struct in Rust:

1. **ManuallyDrop**: We use ‘ManuallyDrop<String>‘ to prevent the automatic dropping of the String while we still have pointers into it. This is crucial for maintaining the validity of our self-references.
2. **Raw Pointers**: We use raw pointers (‘*const u8‘) to keep track of:
    - The start of the text (‘text_{ptr}‘) for position calculations
    - The current position in the text (‘$\text{current}_{\text{ptr}}$‘)

3. **Setup Process**:
    - We carefully initialize the struct by first creating it with a placeholder ‘$\text{current}_{\text{ptr}}$‘
    - Only after the struct is created do we set ‘$\text{current}_{\text{ptr}}$‘ to point to the beginning of the text
4. **Navigation Methods**:
    - ‘position()‘ calculates the byte offset between ‘$\text{current}_{\text{ptr}}$‘ and ‘$\text{text}_{\text{ptr}}$‘
    - ‘$\text{current}_{\text{char}}()$‘ extracts the character at the current position
    - ‘advance()‘ moves the pointer forward by the byte length of the current character
    - ‘reset()‘ returns the pointer to the beginning of the text
    - ‘$\text{remaining}_{\text{text}}()$‘ creates a string slice from the current position to the end
5. **Manual Drop Implementation**: We implement ‘Drop‘ to properly clean up the ‘ManuallyDrop<String>‘ when the parser is dropped.
6. **Clone Implementation**: We implement ‘Clone‘ to correctly handle moving the struct, creating a new one with the same position.

Key safety considerations:

- We ensure that the pointers never outlive the String by using ‘ManuallyDrop‘
- We properly handle UTF-8 characters, which can be multiple bytes
- We check for out-of-bounds access in all methods
- We implement ‘Drop‘ to prevent memory leaks

Self-referential structs are challenging in Rust because:

1. The borrow checker doesn't understand self-references
2. Moving the struct would invalidate the pointers if we didn't handle it correctly
3. The String's memory could be reallocated (e.g., if modified), invalidating our pointers

The solution demonstrates how to safely implement a self-referential struct while maintaining Rust's safety guarantees.