System Programming – Day 2

Rust

Nils Asmussen

09/24/2024

What is Rust?

A language empowering everyone
to build reliable and efficient software.

(rust-lang.org)

Why Another Language?

- We have plenty of languages to build reliably software:
  - Java, C#, Go, Python, Ruby, …
  - All of these trade performance for safety
  - All of them have a runtime (garbage collector, …)

Why Another Language?

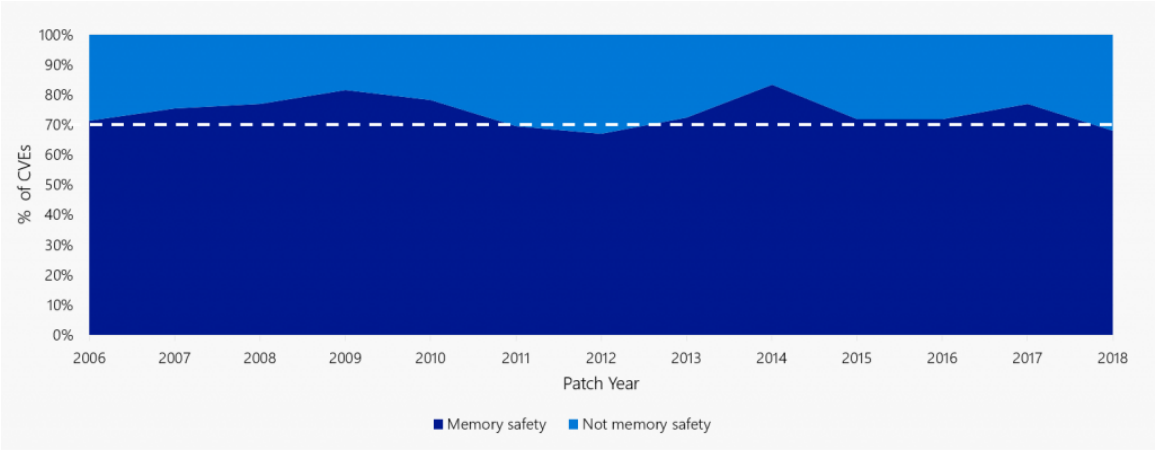
- We have plenty of languages to build reliably software:
  - Java, C#, Go, Python, Ruby, …
  - All of these trade performance for safety
  - All of them have a runtime (garbage collector, …)
- We have plenty of languages to build efficient software:
  - C, C++, D, Assembly, …
  - All of them trade safety for performance

Why Another Language?

- We have plenty of languages to build reliably software:
  - Java, C#, Go, Python, Ruby, …
  - All of these trade performance for safety
  - All of them have a runtime (garbage collector, …)
- We have plenty of languages to build efficient software:
  - C, C++, D, Assembly, …
  - All of them trade safety for performance
- System programming requires efficiency/control *and* safety!

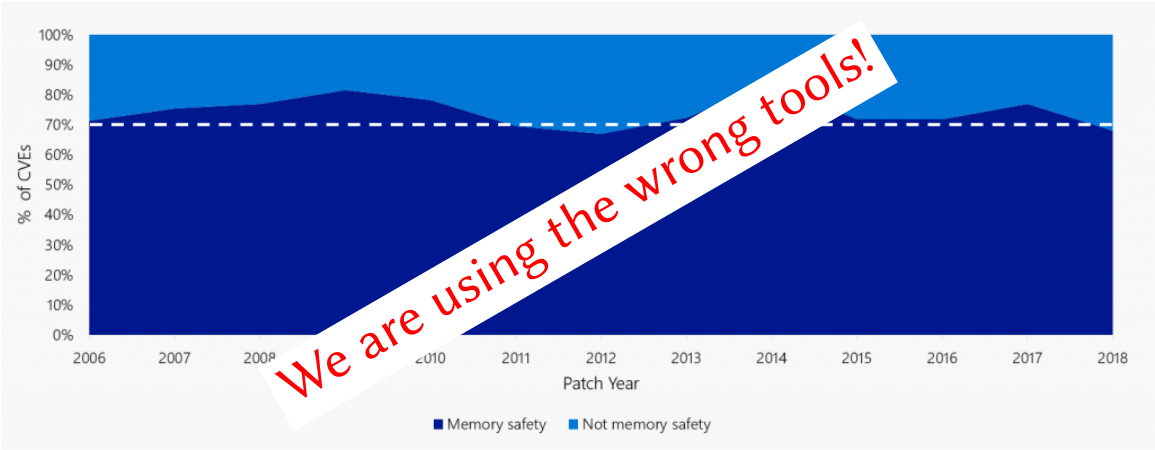
But Good Developers Don’t Need Safety!

But Good Developers Don’t Need Safety!



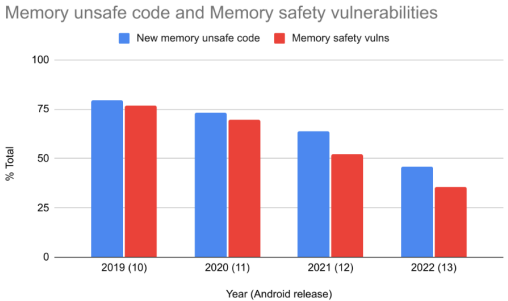
Source: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

But Good Developers Don't Need Safety!



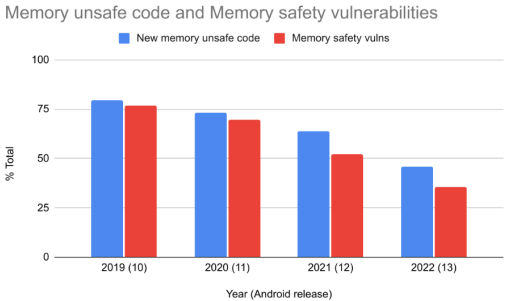
Source: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

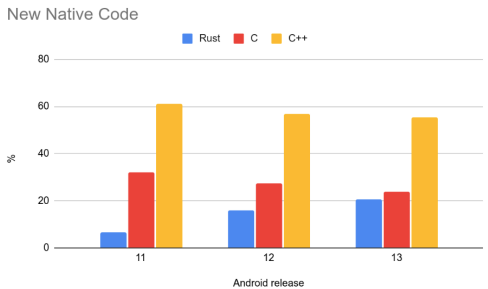
It works!



Source: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html?m=1

It works!





Source: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html?m=1

General Idea of Rust

- C/C++ declare everything that is unsafe as “undefined behavior”
  - That pushes the problem to the developer
  - There is no way out: the developer has the control all the time

General Idea of Rust

- C/C++ declare everything that is unsafe as “undefined behavior”
  - That pushes the problem to the developer
  - There is no way out: the developer has the control all the time
- Rust provides safety without undefined behavior by default
  - The developer can opt out by marking code as “unsafe”
  - The developer *only* has the control if explicitly requested

General Idea of Rust

- C/C++ declare everything that is unsafe as “undefined behavior”
  - That pushes the problem to the developer
  - There is no way out: the developer has the control all the time
- Rust provides safety without undefined behavior by default
  - The developer can opt out by marking code as “unsafe”
  - The developer *only* has the control if explicitly requested
- Rust tracks ownership at compile time and thereby is
  - memory safe
  - data-race free

Agenda

Morning

- Getting started
- Ownership
- Basic features + exercise
- Structs and enums + exercise

Agenda

Morning

- Getting started
- Ownership
- Basic features + exercise
- Structs and enums + exercise

Afternoon

- Generics, traits, and error handling + exercise
- Unsafe, FFI, interior mutability + exercise

Repository

To get the slides and the exercises:

```
$ git clone https://github.com/Nils-TUD/sysprog-rust.git
```

Outline

Installation

- Please install the latest stable version of Rust
- Primary way: `rustup` (installer and version management)
- Some distributions (e.g., Arch) have a package for Rust or `rustup`
- Otherwise:

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf > rup.sh
# check if it's safe and use a fresh shell
$ sh rup.sh
```

Overview

- `rustc` is the Rust compiler; almost never invoked by the user
- `cargo` is Rust's build system and package manager
  - Cargo.toml describes what to build and its dependencies
  - `cargo` downloads dependencies and builds everything automatically
  - Every library/application is a *crate*
  - Crates can be found on `https://crates.io` (or `https://lib.rs`)

Let's Build Hello World!

```
$ cargo new hello
$ cd hello
$ cargo run
```

Outline

Different Memory Management Approaches

- Many high-level languages use garbage collection to manage memory
  - Often not acceptable for OSes, bootloaders, VMMs, …

Different Memory Management Approaches

- Many high-level languages use garbage collection to manage memory
  - Often not acceptable for OSes, bootloaders, VMMs, …
- Many low-level languages let the developer manage memory explicitly
  - Error prone and the main cause for memory-safety issues

Different Memory Management Approaches

- Many high-level languages use garbage collection to manage memory
  - Often not acceptable for OSes, bootloaders, VMMs, …
- Many low-level languages let the developer manage memory explicitly
  - Error prone and the main cause for memory-safety issues
- Rust uses Ownership
  - No garbage collection, no manual allocation
  - The compiler defines a set of rules and enforces them

Ownership Rules

1. Each value has a variable that's called its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be *dropped*.

Ownership Rules – Examples

Valid example

```
{
    let mut var = 4;    // mutable variable
    var += 1;           // we are the owner
}   // var is dropped
```

Ownership Rules – Examples

Valid example

```
{
    let mut var = 4;    // mutable variable
    var += 1;           // we are the owner
}   // var is dropped
```

Invalid example

```
let mut var = 4;
let var_ref = &mut var; // mutable reference to modify `var`
drop(var);              // explicit drop
*var_ref = 5;           // error (use after free)
```

Ownership Transfer and Borrowing

1. The owner of a value can *transfer* the ownership to someone else.

```
let var = String::from("hello"); // heap-allocated string
fn foo(name: String) { /* name is dropped */ }
foo(var);                        // transfer ownership to foo
```

Ownership Transfer and Borrowing

1. The owner of a value can *transfer* the ownership to someone else.

```
let var = String::from("hello"); // heap-allocated string
fn foo(name: String) { /* name is dropped */ }
foo(var);                        // transfer ownership to foo
```

2. Others can *borrow* a value from the owner.

```
let mut var = String::from("hello");    // mutable String
fn foo(name: &String) { /* use name */ }
foo(&var);                              // let foo borrow var
var.push(' ');                          // we are the owner again
```

Outline

Data Types (1)

- Scalars
  - Integers: `u32`, `i64`, `usize`, `...`
  - Floats: `f32`, `f64`
  - Boolean: `bool`
  - Character: `char`
- Structs

```
struct Foo {
    field1: u32,
    field2: String,
}
```

Data Types (2)

- Tuples

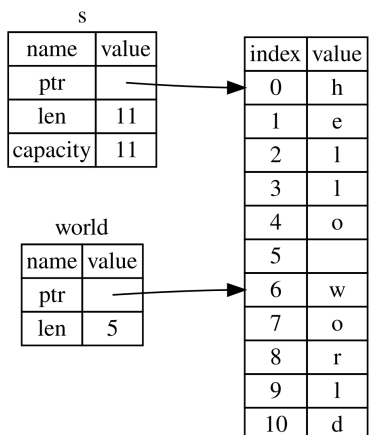
```
let mut tuple = (1, "foo", 42);      // tuple length is fixed
tuple.0 += 1;                        // values are mutable
let (x, y, z) = tuple;               // destructuring
```

- Arrays

```
let mut array: [u32; 2] = [1, 2];    // arrays have a fixed size
array[3] += 1;                       // runtime error (bounds checked)
let foo = [0; 12];                   // array with 12 elements with value 0
```

Strings and Slices

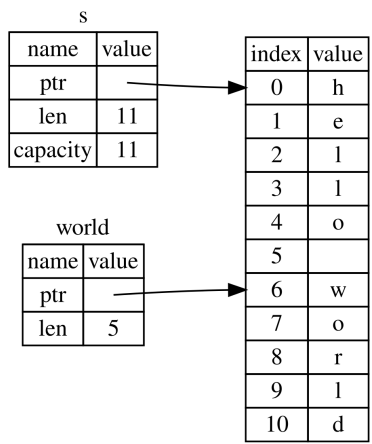
```
let s = String::from("hello world");
// String ~= Vec<char>
let world = &s[6..11];
// &str ~= &[char]
```



Strings and Slices

```
let s = String::from("hello world");
// String ~= Vec<char>
let world = &s[6..11];
// &str ~= &[char]

&s[0..11]   // = "hello world"
&s[6..]     // = "world"
&s[..5]     // = "hello"
&s[..]      // = "hello world"
```

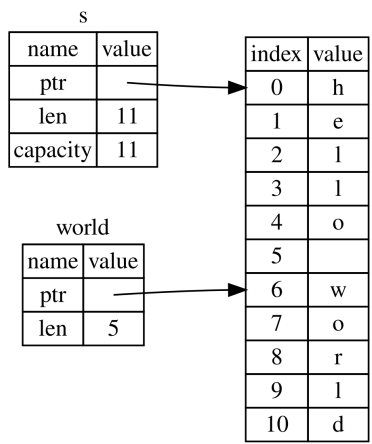


Strings and Slices

```
let s = String::from("hello world");
// String ~= Vec<char>
let world = &s[6..11];
// &str ~= &[char]

&s[0..11]   // = "hello world"
&s[6..]     // = "world"
&s[..5]     // = "hello"
&s[..]      // = "hello world"

let a = [1, 2, 3];
&a[0..1]    // = [1]
```



Control Structures

- If expressions

```
if condition { println!("foo"); } else { println!("bar"); }
let val = if condition { 4 } else { 5 };
```

- Loop

```
loop { }
```

- While

```
while condition { }
```

- For

```
for i in 0..10 { }
for c in "test".chars() { }
```

Functions

```rust
pub fn func_without_return_val(arg: u32) {
    if arg > 0 {
        return;
    }
    // do something
}

pub fn func_with_return_val(arg1: usize, arg2: usize) -> usize {
    // last expression is the return value
    arg1 + arg2
}
```

Exercise 1 – String Operations

- First exercise is in directory “words”
- Fill in the implementation of the functions
- Use the existing tests to verify your implementation:

```
$ cargo test
```

- Hint: use the standard library (`https://doc.rust-lang.org/stable`):
  - `str::chars`
  - `char::is_uppercase`
  - `str::split_whitespace`

Outline

More on Structs

- Struct definitions

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

More on Structs

- Struct definitions

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

- Methods

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

More on Structs

- Struct definitions

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

- Methods

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

- Methods with mutable self

```
fn widen(&mut self, amount: u32) {
    self.width += amount;
}
```

More on Structs

- Struct definitions

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

- Methods

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

- Methods with mutable self

```
fn widen(&mut self, amount: u32) {
    self.width += amount;
}
```

- Methods that take ownership

```
fn flip(self) -> Rectangle {
    Rectangle {
        width: self.height,
        height: self.width,
    }
}
```

Enums

- Simple enumeration (like in C++)

```
enum Animal {
    Sheep,
    Cow,
}
```

Enums

- Simple enumeration (like in C++)

```
enum Animal {
    Sheep,
    Cow,
}
```

- Enums with data (tagged union)

```
enum Message {
    Open(String),
    Read(usize, usize),
}
```

Enums

- Simple enumeration (like in C++)

```
enum Animal {
    Sheep,
    Cow,
}
```

- Enums with data (tagged union)

```
enum Message {
    Open(String),
    Read(usize, usize),
}
```

- Construction

```
Message::Open(String::from("Hello!"));
Message::Read(0, 1024);
```

Enums

- Simple enumeration (like in C++)

```
enum Animal {
    Sheep,
    Cow,
}
```

- Enums with data (tagged union)

```
enum Message {
    Open(String),
    Read(usize, usize),
}
```

- Construction

```
Message::Open(String::from("Hello!"));
Message::Read(0, 1024);
```

- Matching

```
match msg {
    Message::Open(filename) => ...,
    _ => println!("Unsupported"),
}
if let Message::Read(pos, num) = msg {
}
```

Closures: Basics

- Closures are anonymous functions that can be stored:

```
let adder = |x| { x += 1 };
```

Closures: Basics

- Closures are anonymous functions that can be stored:

```
let adder = |x| { x += 1 };
```

- Closures can also capture their environment:

```
fn foo() {
    let y = 42;
    let adder = |x| { x += y };
}
```

Closures: Contract between Implementer and Consumer

Example

```
fn count<F: ...>(elems: &[u32], func: F) -> usize {
    let mut count = 0;
    for e in elems { if func(e) { count += 1; } }
    count
}
```

Closures: Contract between Implementer and Consumer

Example

```
fn count<F: ...>(elems: &[u32], func: F) -> usize {
    let mut count = 0;
    for e in elems { if func(e) { count += 1; } }
    count
}
```

	Implementer	**Consumer**
`FnOnce`	Move out ✓; Mutate ✓	Can call only once
`FnMut`	Move out x; Mutate ✓	Can call multiple times with unique access
`Fn`	Move out x; Mutate x	Can call multiple times, no restrictions

Closures: Contract between Implementer and Consumer

Example

```
fn count<F: FnMut(&u32) -> bool>(elems: &[u32], mut func: F) -> usize {
    let mut count = 0;
    for e in elems { if func(e) { count += 1; } }
    count
}
```

	Implementer	**Consumer**
`FnOnce`	Move out ✓; Mutate ✓	Can call only once
`FnMut`	Move out x; Mutate ✓	Can call multiple times with unique access
`Fn`	Move out x; Mutate x	Can call multiple times, no restrictions

Exercise 2 – Command Line Book Collection

- Second exercise is in directory “books”
- Simple command line program that lets the user manage a collection of books
- Fill in the missing parts (parsing, command execution)
- For simplicity:
  - It’s okay to only support single-word book titles
  - If you see `Option`/`Result`: use unwrap/panic (we’ll add proper error handling later)
- The following building blocks might be helpful:
  - `Iterator::collect`
  - `Iterator::find`
  - `Vec::push`
  - `Vec::retain`

Outline

Basics of Generics

- Generics allow to define functions/structs/enums for a variety of concrete types:

```
fn foo<T>(arg: T) { /* ... */ }
```

- Generics have no runtime overhead due to monomorphization:

```
fn foo<T>(arg: T) { /* ... */ }
// is compiled to something like:
fn foo_u32(arg: u32) { /* ... */ }
fn foo_u64(arg: u64) { /* ... */ }
```

- Rust is strict about the requirements for type parameters
  (based on traits, as we will see shortly)

Generic Types

- Generic function

```
fn head<T>(elems: &Vec<T>) -> &T {
    &elems[0]
}
assert_eq!(*head(&vec![1, 2]), 1);
```

Generic Types

- Generic function

```
fn head<T>(elems: &Vec<T>) -> &T {
    &elems[0]
}
assert_eq!(*head(&vec![1, 2]), 1);
```

- Generic struct

```
struct Rectangle<T> {
    width: T,
    height: T,
}
Rectangle { width: 1.2, height: 4.5 }
```

Generic Types

- Generic function

```
fn head<T>(elems: &Vec<T>) -> &T {
    &elems[0]
}
assert_eq!(*head(&vec![1, 2]), 1);
```

- Generic struct

```
struct Rectangle<T> {
    width: T,
    height: T,
}
Rectangle { width: 1.2, height: 4.5 }
```

- Generic enum

```
enum Option<T> {
    Some(T),
    None,
}
```

Generic Types

- Generic function

```
fn head<T>(elems: &Vec<T>) -> &T {
    &elems[0]
}
assert_eq!(*head(&vec![1, 2]), 1);
```

- Generic struct

```
struct Rectangle<T> {
    width: T,
    height: T,
}
Rectangle { width: 1.2, height: 4.5 }
```

- Generic enum

```
enum Option<T> {
    Some(T),
    None,
}
```

- Generic method

```
impl<T: AddAssign> Rectangle<T> {
    fn widen(&mut self, amount: T) {
        self.width += amount;
    }
}
```

Trait Basics

- A *trait* defines a behavior that can be implemented by multiple types:

```
trait Shape {
    fn area(&self) -> u32;
}
```

Trait Basics

- A *trait* defines a behavior that can be implemented by multiple types:

```
trait Shape {
    fn area(&self) -> u32;
}
```

- Implementing a trait for a type:

```
impl Shape for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

More on Traits (1)

- Using trait bounds:

```
fn to_vec<T: Clone>(slice: &[T]) -> Vec<T> {
    let mut vec = Vec::new();
    for e in slice { vec.push(e.clone()); }
    vec
}
```

More on Traits (1)

- Using trait bounds:

```
fn to_vec<T: Clone>(slice: &[T]) -> Vec<T> {
    let mut vec = Vec::new();
    for e in slice { vec.push(e.clone()); }
    vec
}
```

- Static vs. dynamic dispatch:

```
// one function for each type
fn static_dispatch<T: Shape>(sh: &T) { }
fn static_dispatch(sh: &impl Shape) { } // syntactic sugar
// one function for all types, dispatched at runtime
fn dynamic_dispatch(sh: &dyn Shape) { }
```

More on Traits (2)

- Derive attribute:

```
#[derive(Debug)]
struct Point {
    x: u32,
    y: u32,
}
```

More on Traits (2)

- Derive attribute:

```
#[derive(Debug)]
struct Point {
    x: u32,
    y: u32,
}

let p = Point { x: 0, y: 16 };
println!("p = {:?}", p); // prints "p = Point { x: 0, y: 16 }"
```

Copy vs. Move Semantics

C++

- Copy semantics by default
- Copy constructor etc. is auto-implemented by compiler (opt out possible)
- Programmer can opt into move semantics by implementing move constructor etc.

Copy vs. Move Semantics

C++

- Copy semantics by default
- Copy constructor etc. is auto-implemented by compiler (opt out possible)
- Programmer can opt into move semantics by implementing move constructor etc.

Rust

- Move semantics by default: ownership is transferred
- Programmer can opt into copy semantics via `#[derive(Copy)]`
- If a type implements `Copy`, a flat copy is performed instead of ownership transfer
- Deep copies are explicit via `clone` (see `Clone` trait)

Error Handling

- Unrecoverable errors with `panic!`:
  - Sometimes the best you can do
  - Can perform stack unwinding or not (set panic=abort)
  - Provides a backtrace to the user

Error Handling

- Unrecoverable errors with `panic!`:
  - Sometimes the best you can do
  - Can perform stack unwinding or not (set panic=abort)
  - Provides a backtrace to the user
- Recoverable errors with `Result`:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Error Handling Basics

Returning errors (simplified `std::fs::File::open`)

```
pub fn open(path: &str) -> Result<File, Error> {
    ...
    if ... { return Err(Error::NotFound); }
    ...
}
```

Error Handling Basics

Returning errors (simplified `std::fs::File::open`)

```
pub fn open(path: &str) -> Result<File, Error> {
    ...
    if ... { return Err(Error::NotFound); }
    ...
}
```

Handling errors

```
let mut file = std::fs::File::open("myfile.txt").expect("open failed");
```

Passing Errors Upwards

```rust
let mut file = std::fs::File::open(path)?;
// is equivalent to:
let mut file = match std::fs::File::open(path) {
    Ok(file) => file,
    Err(e) => return Err(e),
};
```

Passing Errors Upwards

```
let mut file = std::fs::File::open(path)?;
// is equivalent to:
let mut file = match std::fs::File::open(path) {
    Ok(file) => file,
    Err(e) => return Err(e),
};

fn read_file(path: &str) -> Result<String, Error> {
    let mut file = std::fs::File::open(path)?;
    let mut s = String::new();
    file.read_to_string(&mut s)?;
    Ok(s)
}
```

Option Instead of Nullpointers

- Similar to `Result` for errors, Rust uses `Option` for optional values:

```
enum Option<T> {
    Some(T),
    None,
}
```

Option Instead of Nullpointers

- Similar to `Result` for errors, Rust uses `Option` for optional values:

```
enum Option<T> {
    Some(T),
    None,
}
```

- Important methods on `Result` and `Option`
  - `unwrap`: panic if `None`/`Err`
  - `expect`: panic with message if `None`/`Err`
  - `*_or_else`: transformation

Option Instead of Nullpointers

- Similar to `Result` for errors, Rust uses `Option` for optional values:

```
enum Option<T> {
    Some(T),
    None,
}
```

- Important methods on `Result` and `Option`
  - `unwrap`: panic if `None`/`Err`
  - `expect`: panic with message if `None`/`Err`
  - `*_or_else`: transformation
- More at `https://doc.rust-lang.org/stable`

Exercise 3 – Proper Error Handling

- Let's add proper error handling to our books collection
- Get rid of all panics/unwraps
- Use `Result` and `Option` where appropriate
- Hints:
  - Introduce your own error enum
  - Attach `#[derive(Debug)]` to your error enum
  - Implement `From<std::num::ParseIntError>` for your enum
  - Implement `Display` for Book

Outline

Unsafe

- Rust allows you to enable additional features via `unsafe`
- Tells the compiler that you know what you're doing
- Does *not* turn off safety checks, but allows you additionally to:
  - Dereference raw pointers
  - Call unsafe functions
- Unsafe code is typically used to build safe abstractions (`Vec`, `String`, ...)
- Example:

```
let mut_ptr = 0xB8000 as *mut u32;          // VGA frame buffer
let const_ptr = 0xDEAD_BEEF as *const u32;
unsafe { *mut_ptr = *const_ptr; }
```

FFI: Interfacing with Other Languages

- Rust can interface with other languages through the foreign function interface (FFI)
- Allows to call C functions from Rust:

```
extern "C" {
    fn abs(input: i32) -> i32;
}
unsafe { abs(-2) };
```

FFI: Interfacing with Other Languages

- Rust can interface with other languages through the foreign function interface (FFI)
- Allows to call C functions from Rust:

```
extern "C" {
    fn abs(input: i32) -> i32;
}
unsafe { abs(-2) };
```

- And to export Rust functions to C:

```
#[no_mangle]
extern "C" fn rust_double(arg: u64) -> u64 {
    arg * 2
}
```

Interior Mutability

- The ownership model is sometimes too restrictive
- Interior mutability allows to mutate data with an immutable reference

Interior Mutability

- The ownership model is sometimes too restrictive
- Interior mutability allows to mutate data with an immutable reference
- How can that be safe?
  - `Cell`: no reference to internal data; data is copied
  - `RefCell`: track references at runtime
  - `Mutex`: track references at runtime in a thread-safe way

Interior Mutability: UnsafeCell

```rust
// simplified implementation
pub struct UnsafeCell<T> { value: T }

impl<T> UnsafeCell<T> {
    pub unsafe fn get_mut(&self) -> &mut T {
        let mut_ptr = &self.value as *const T as *mut T;
        unsafe { &mut *mut_ptr }
    }
}
```

Interior Mutability: `RefCell`

- Implemented based on `UnsafeCell` and `Cell`
- Does not hand out “plain” references
- Instead hands out the types `Ref` and `RefMut`:
  - `pub fn borrow(&self) -> Ref<T>`
  - `pub fn borrow_mut(&self) -> RefMut<T>`
- `Ref`/`RefMut` hold a reference and provide access to the data

Exercise 4 – Cells

- Final exercise is in directory “cells”
- Book collection that is shared (stored in `Rc`)
- Attributes of books should be changable
- You need to:
  - Implement the `set_*` methods
  - Use `Cell` and `RefCell` as appropriate