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!

We closely study the root cause trends of vulnerabilities & search for patterns

% of memory safety vs. non-memory safety CVEs by patch year

Patch Year

% of CVEs
0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

Memory safety  Not memory safety
But Good Developers Don’t Need Safety!

We closely study the root cause trends of vulnerabilities and search for patterns.

We are using the wrong tools!
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

Afternoon
- Generics, traits, and error handling + exercise
- Unsafe, FFI, interior mutability
- Exercise: implement semaphores for a small kernel
## Agenda

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

### Afternoon
- Generics, traits, and error handling + exercise
- Unsafe, FFI, interior mutability
- Exercise: implement semaphores for a small kernel
Repository

To get the slides and the exercises:

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

1. Getting Started
2. Ownership
3. Basic Features
4. Structs, Enums, and Closures
5. Generics, Traits, and Error Handling
6. Unsafe, FFI, Interior Mutability
Installation

- We need rustup (not rustc) to install the nightly version
- Some distributions (e.g., Arch) have a package for 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

1. Getting Started
2. Ownership
3. Basic Features
4. Structs, Enums, and Closures
5. Generics, Traits, and Error Handling
6. Unsafe, FFI, Interior Mutability
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**

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

**Invalid example**

```rust
let mut var = 4;
let var_ref = &mut var; // mutable reference to modify

drop(var); // explicit drop
*var_ref = 5; // error (use after free)
```
Ownership Rules – Examples

Valid example

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

Invalid example

```rust
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.

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

Others can *borrow* a value from the owner.

```rust
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
```
Ownership Transfer and Borrowing

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

```rust
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.

```rust
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

1. Getting Started
2. Ownership
3. Basic Features
4. Structs, Enums, and Closures
5. Generics, Traits, and Error Handling
6. Unsafe, FFI, Interior Mutability
Data Types (1)

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

- Structs

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

  - str::chars
  - char::is_uppercase
  - str::split_whitespace
Outline

1. Getting Started
2. Ownership
3. Basic Features
4. Structs, Enums, and Closures
5. Generics, Traits, and Error Handling
6. Unsafe, FFI, Interior Mutability
More on Structs

- Struct definitions

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

- Methods

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

- Methods with mutable self

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

- Methods that take ownership

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

- **Struct definitions**

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

- **Methods**

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

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

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

- **Struct definitions**

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

- **Methods**

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

- **Methods with mutable self**

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

- Struct definitions

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

- Methods

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

- Methods with mutable self

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

- Methods that take ownership

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

- Simple enumeration (like in C++)

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

- Enums with data (tagged union)

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

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

```rust
match msg {
    Message::Open(filename) => ...,
    _ => println!("Unsupported"),
}
```

```rust
if let Message::Read(pos, num) = msg {
}
```
 Enums

- Simple enumeration (like in C++)

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

- Enums with data (tagged union)

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

Construction

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

Matching

```rust
match msg {
    Message::Open(filename) => ...,
    _ => println!(&#34;Unsupported&#34;),
}

if let Message::Read(pos, num) = msg {
}
```
 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++)

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

- Enums with data (tagged union)

```rust
enum Message {
    Open(String::from("Hello!"));
    Read(usize, usize);
}
```

- Construction

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

- Matching

```rust
match msg {
    Message::Open(filename) => ..., 
    _ => println!("Unsupported"),
}
```

```rust
if let Message::Read(pos, num) = msg {
}
```
Closures are anonymous functions that can be stored:

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

Closures can also capture their environment:

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

- Closures are anonymous functions that can be stored:

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

- Closures can also capture their environment:

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

1. **Fn**: capture environment by immutable references
2. **FnMut**: capture environment by mutable references
3. **FnOnce**: capture environment by ownership transfer

Example

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

1. **Fn**: capture environment by immutable references
2. **FnMut**: capture environment by mutable references
3. **FnOnce**: capture environment by ownership transfer

Example

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

1. Getting Started
2. Ownership
3. Basic Features
4. Structs, Enums, and Closures
5. Generics, Traits, and Error Handling
6. Unsafe, FFI, Interior Mutability
Basics of Generics

- Generics allow to define functions/structs/enums for a variety of concrete types:
  ```rust
  fn foo<T>(arg: T) { /* ... */ }
  ```

- Generics have no runtime overhead due to monomorphization:
  ```rust
  fn foo<T>(arg: T) { /* ... */ }
  // is compiled to something like:
  fn foo_u32(arg: u32) { /* ... */ }
  fn foo_u64(arg: u64) { /* ... */ }
  ```

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

- Generic function

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

- Generic struct

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

- Generic enum

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

- Generic method

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

- **Generic function**

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

- **Generic struct**

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

- **Generic enum**

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

- **Generic method**

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

- Generic function
  ```rust
def fn head<T>(elems: &Vec<T>) -> &T {
    &elems[0]
  }
  ```

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

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

- Generic enum
  ```rust
enum Option<T> {
  Some(T),
  None,
}
```
Generic Types

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

- **Generic struct**
  
  ```rust
  struct Rectangle<T> {
    width: T,
    height: T,
  }
  Rectangle { width: 1.2, height: 4.5 }
  ```

- **Generic enum**
  
  ```rust
  enum Option<T> {
    Some(T),
    None,
  }
  ```

- **Generic method**
  
  ```rust
  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:

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

- A *trait* defines a behavior that can be implemented by multiple types:
  ```rust
  trait Shape {
    fn area(&self) -> u32;
  }
  ```

- Implementing a trait for a type:
  ```rust
  impl Shape for Rectangle {
    fn area(&self) -> u32 {
      self.width * self.height
    }
  }
  ```
More on Traits (1)

- Using trait bounds:

```rust
fn sum<T: AddAssign + Copy + Default>(nums: &Vec<T>) -> T {
    let mut sum = T::default();
    for n in nums { sum += *n; }
    sum
}
```

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 (1)

- Using trait bounds:

```rust
fn sum<T: AddAssign + Copy + Default>(nums: &Vec<T>) -> T {
    let mut sum = T::default();
    for n in nums { sum += *n; }
    sum
}
```

- Static vs. dynamic dispatch:

```rust
// 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:

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

  ```rust
  let p = Point { x: 0, y: 16 }; // prints "p = Point { x: 0, y: 16 }",
  ```
More on Traits (2)

- Derive attribute:

```rust
#[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.

Rust

Move semantics by default: ownership is transferred
Programmer can opt into copy semantics via 

```rust
#[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)
## 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**:

```rust
typedef enum Result<T, E> {
    Ok(T),
    Err(E),
}
```
Error Handling Basics

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

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

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

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

Handling errors

```rust
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

```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),
};

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:

```rust
enum Option<T> {
    Some(T),
    None,
}
```
Option Instead of Nullpointers

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

  ```rust
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
Option Instead of Nullpointers

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

```rust
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

Exercise 3 – Proper Error Handling

- Let’s add proper error handling to our books collection
- Use Result and Option where appropriate
- Get rid of all panics/unwraps

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

1. Getting Started
2. Ownership
3. Basic Features
4. Structs, Enums, and Closures
5. Generics, Traits, and Error Handling
6. Unsafe, FFI, Interior Mutability
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:

  ```rust
  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:
  ```rust
  extern "C" {
    fn abs(input: i32) -> i32;
  }
  unsafe { abs(-2) };  
  ```
Rust can interface with other languages through the foreign function interface (FFI). Allows to call C functions from Rust:

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

And to export Rust functions to C:

```rust
#[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
// 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 any 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
- RefCell is used in the small kernel for global variables (see StaticRefCell)
Exercise 4 – Semaphores

- Last exercise is in directory “kernel”
- Simple kernel that supports exactly two programs and runs in physical memory
- The program is instantiated two times and performs prints in a loop
- The prints currently mix occasionally; use semaphores to prevent it
- You need:
  - Add the Semaphore implementation (with up and down based on existing task module)
  - Use the syscalls in the user program