Rust
A safe language for low-level programming

Embedded World 2022

Stefan Wehr, Hochschule Offenburg

Outline

- Lecture 1: Motivation, Rust basics
- Practice session 1

- Lecture 2: High-level language constructs
- Practice session 2

- Lecture 3: Advanced concepts
- Practice session 3
- Summary

Material available on github:

https://github.com/skogsbaer/rust_class_ew2022

Lecture 1

Motivation, Rust basics

Motivation

Tension between **safety** and **control**

Safety

- No memory errors
- Automatic memory management
- Data encapsulation
- Java, C#, Go, Haskell, ...

Motivation

Tension between safety and control

Safety

- No memory errors
- Automatic memory management
- Data encapsulation
- Java, C#, Go, Haskell, ...

Control

- Control memory layout
- Optimize time and space usage
- Real-time requirements
- C, C++

Motivation

Tension between **safety** and **control**

Safety

- No memory errors
- Automatic memory management
- Data encapsulation
- Java, C#, Go, Haskell, ...

Want both?

Pick one!

Control

- Control memory layout
- Optimize time and space usage
- Real-time requirements
- C, C++

- Systems programming requires control
- Microsoft: ~70% of security violations are caused by memory violations.
- Mozilla: majority of critical bugs in Firefox are memory related.

What makes a language unsafe?

Root of all evil:
unrestricted combination
of **aliasing** and **mutation**

- Use after free error
- Dangling references
- Data races
- ...

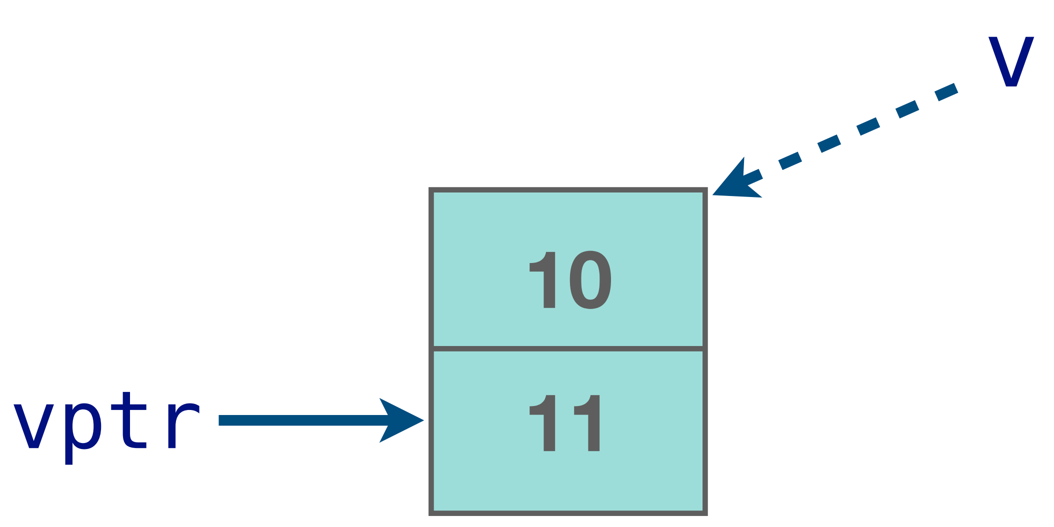
What makes a language unsafe?

Root of all evil:
unrestricted combination
of **aliasing** and **mutation**

- Use after free error
- Dangling references
- Data races
- ...

C++

```cpp
std::vector<int> v { 10, 11 };
int *vptr = &v[1];    // Alias, points into v
v.push_back(12);      // Mutate the vector
std::cout << *vptr;   // Bug (use-after-free)
```



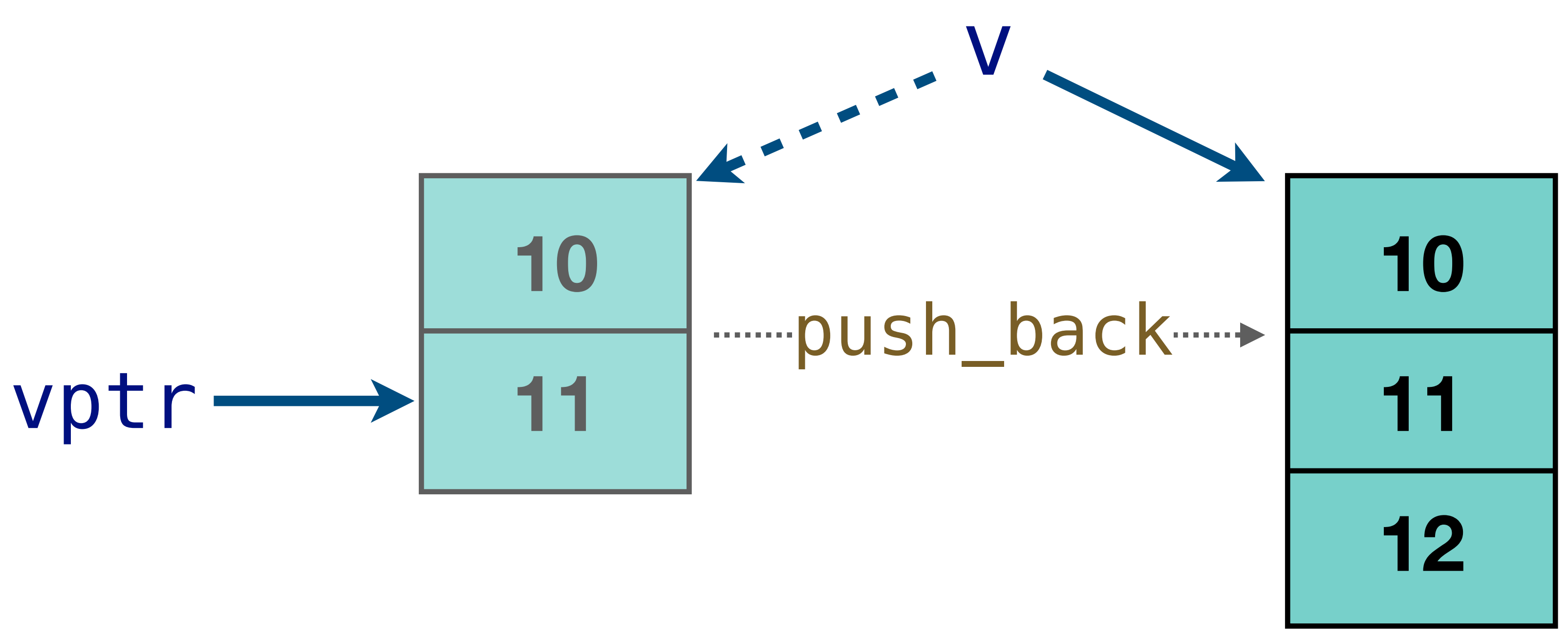
What makes a language unsafe?

Root of all evil:
unrestricted combination
of **aliasing** and **mutation**

- Use after free error
- Dangling references
- Data races
- ...

C++

```cpp
std::vector<int> v { 10, 11 };
int *vptr = &v[1];   // Alias, points into v
v.push_back(12);     // Mutate the vector
std::cout << *vptr;  // Bug (use-after-free)
```



Type checker to the rescue

- Rust detects this bug at **compile time.**
- Rust prevents unrestricted combination of **aliasing** and **mutation.**
- Details explained later.

Rust

```
let mut v = vec![10, 11];
let vptr = &v[1];         // Alias, points into v
v.push(12);               // Mutate the vector
println!("{}", *vptr); // Compile error
```

Compile error: cannot borrow v as mutable because it is also borrowed as immutable

Ownership

- Each value in Rust has a variable that's called its **owner.**
- There can only be **one owner at a time.**
- Ownership is **moved** on assignment and function calls.
- When the owner goes out of scope, the value will be **released.**

Rust

```rust
fn consume(w: Vec<i32>) {
    println!("Length of vector: {}", w.len());
    // Memory of w is released automatically
}

let mut v = vec![10, 11];
consume(v); // Transfers ownership (call by value)
```

Ownership

- Each value in Rust has a variable that's called its **owner.**
- There can only be **one owner at a time.**
- Ownership is **moved** on assignment and function calls.
- When the owner goes out of scope, the value will be **released.**

Rust

```rust
fn consume(w: Vec<i32>) {
    println!("Length of vector: {}", w.len());
    // Memory of w is released automatically
}

let mut v = vec![10, 11];
consume(v); // Transfers ownership (call by value)
v.push(12); // Compile error
```

Compile error: borrow of moved value v

Borrowing

- Ownership transfer is not always want we want.
- We can also temporarily **borrow** a value.
- **&T** is a **reference** to a value of type **T.**
- References are borrowed.
    - No ownership transfer
    - No release at end of borrow

Borrowing

- Ownership transfer is not always want we want.
- We can also temporarily **borrow** a value.
- **&T** is a **reference** to a value of type **T.**
- References are borrowed.
  - No ownership transfer
  - No release at end of borrow

Call by reference

Rust

```
fn just_use(w: &Vec<i32>) {
    println!("Length of vector: {}", w.len());
    // No release
}

let mut v = vec![10, 11];
just_use(&v); // Borrows reference
v.push(12);   // Works
```

Mutability

- Variables are **immutable by default.**
- Immutability is **deep:** cannot change *anything* inside.

```
let v = vec![10, 11];
v = vec![1,2];
```

Compile error: cannot assign twice to immutable variable v.

Mutability

- Variables are **immutable by default.**
- Immutability is **deep:** cannot change *anything* inside.

```
let v = vec![10, 11];
v = vec![1,2];
```

Compile error: cannot assign twice to immutable variable v.

```
let v = vec![10, 11];
v.push(13);
```

Compile error: cannot borrow v as mutable.

Mutability

- Variables are **immutable by default.**
- Immutability is **deep:** cannot change *anything* inside.

```
let v = vec![10, 11];
v = vec![1,2];
```

Compile error: cannot assign twice to immutable variable v.

```
let v = vec![10, 11];
v.push(13);
```

Compile error: cannot borrow v as mutable.

Type of v: `Vec<Vec<i32>`

```
let v = vec![vec![10, 11]];
v[0].push(13);
```

Compile error: cannot borrow v as mutable.

Mutability

- Variables are **immutable by default.**
- Immutability is **deep:** cannot change *anything* inside.

Type of v: `Vec<Vec<i32>>`

```
let v = vec![10, 11];
v = vec![1,2];
```

Compile error: cannot assign twice to immutable variable v.

```
let v = vec![10, 11];
v.push(13);
```

Compile error: cannot borrow v as mutable.

```
let v = vec![vec![10, 11]];
v[0].push(13);
```

Compile error: cannot borrow v as mutable.

- `let mut` declares a variable as mutable

```
let mut v = vec![10, 11];
v = vec![1,2]; // Works
```

```
let mut v = vec![10, 11];
v.push(13); // Works
```

```
let mut v = vec![vec![10, 11]];
v[0].push(13); // Works
```

Mutability

- Variables are **immutable by default.**
- Immutability is **deep:** cannot change *anything* inside.

Type of v: `Vec<Vec<i32>>`

```
let v = vec![10, 11];
v = vec![1,2];
```

Compile error: cannot assign twice to immutable variable v.

```
let v = vec![10, 11];
v.push(13);
```

Compile error: cannot borrow v as mutable.

```
let v = vec![vec![10, 11]];
v[0].push(13);
```

Compile error: cannot borrow v as mutable.

- `let mut` declares a variable as mutable

```
let mut v = vec![10, 11];
v = vec![1,2]; // Works
```

```
let mut v = vec![10, 11];
v.push(13); // Works
```

```
let mut v = vec![vec![10, 11]];
v[0].push(13); // Works
```

- `&T` immutable borrow
- `&mut T` mutable borrow

Rules of ownership and borrowing

- **Exactly one owner.**
  - Ownership moved on assigment.
  - Memory is released if owner goes out of scope.
- **One mutable borrow XOR any number of immutable borrows at the same time.**
- The lifetime of a borrow is determined from the source code or explicit annotations.

Rules of ownership and borrowing

- **Exactly one owner.**
  - Ownership moved on assigment.
  - Memory is released if owner goes out of scope.
- **One mutable borrow XOR any number of immutable borrows at the same time.**
- The lifetime of a borrow is determined from the source code or explicit annotations.

Value of type
- `T` *ownership*
- `&T` *immutable borrow*
- `&mut T` *mutable borrow*

Expression
- `e` *move*
- `&e` *immutable ref*
- `&mut e` *mutable ref*

Rules of ownership and borrowing

- **Exactly one owner.**
  - Ownership moved on assigment.
  - Memory is released if owner goes out of scope.
- **One mutable borrow XOR any number of immutable borrows at the same time.**
- The lifetime of a borrow is determined from the source code or explicit annotations.

Value of type
- `T` *ownership*
- `&T` *immutable borrow*
- `&mut T` *mutable borrow*

Expression
- `e` *move*
- `&e` *immutable ref*
- `&mut e` *mutable ref*

```
let mut v = vec![10, 11];
let vptr = &v[1];
v.push(12);
println!("{}", *vptr);
```

- *Declaration of push:* `fn push(v: &mut Vec<i32>, i: i32)`
- `v.push(12)` *is short for* `push(&mut v, 12)`

Rules of ownership and borrowing

- **Exactly one owner.**
  - Ownership moved on assigment.
  - Memory is released if owner goes out of scope.
- **One mutable borrow XOR any number of immutable borrows at the same time.**
- The lifetime of a borrow is determined from the source code or explicit annotations.

Value of type
- `T` *ownership*
- `&T` *immutable borrow*
- `&mut T` *mutable borrow*

Expression
- `e` *move*
- `&e` *immutable ref*
- `&mut e` *mutable ref*

```
let mut v = vec![10, 11];
let vptr = &v[1];
v.push(12);
println!("{}", *vptr);
```

Lifetime of immutable borrow `vptr`

Lifetime of mutable borrow for calling `push`

Overlapping lifetimes of two borrows, one borrow is mutable.

Compile error: `cannot borrow v as mutable because it is also borrowed as immutable`

- *Declaration of push:* `fn push(v: &mut Vec<i32>, i: i32)`
- `v.push(12)` *is short for* `push(&mut v, 12)`

Aliasing and mutation

`&T` Immutable borrow / shared reference: **aliasing but no mutation**

`&mut T` Mutable borrow / unique reference: **no aliasing but mutation**

Aliasing and mutation

`&T` Immutable borrow / shared reference: **aliasing but no mutation**

`&mut T` Mutable borrow / unique reference: **no aliasing but mutation**

OK: immutable references shared between threads

```
let mut v = vec![10,11];
join(|| println!("v[1] = {}", &v[1]),
     || println!("v[1] = {}", &v[1]));
```

Spawns two threads, waits until completion

Anonymous function, taking no arguments

Aliasing and mutation

`&T` Immutable borrow / shared reference: **aliasing but no mutation**

`&mut T` Mutable borrow / unique reference: **no aliasing but mutation**

OK: immutable references shared between threads

```
let mut v = vec![10,11];
join(|| println!("v[1] = {}", &v[1]),
     || println!("v[1] = {}", &v[1]));
```

Spawns two threads, waits until completion

Anonymous function, taking no arguments

Data race: parallel access to the same data, one write access

```
let mut v = vec![10,11];
join(|| println!("v[1] = {}", &v[1]),
     || v.push(13));
```

Compile error: `cannot borrow v as mutable because it is also borrowed as immutable`

Rust Basics (1/2)

Rust book:
`https://doc.rust-lang.org/book/`

Variable declarations

```
let a = 42;        // immutable
let mut b = 0;     // mutable
let c: i32 = 5;    // optional type annotation
```

Rust Basics (1/2)

Rust book:
`https://doc.rust-lang.org/book/`

Variable declarations

```rust
let a = 42;       // immutable
let mut b = 0;    // mutable
let c: i32 = 5;   // optional type annotation
```

Function definitions

Result type (omit if void)

```rust
pub fn some_function(x: i32, y: i32) -> i32 {
    let z = x + y;
    z + 1
}
```

Visibility (public)

No distinction between statements and expression:
Everything is an expression
; sequences expressions
No return needed (but possible)
No ; at the end!

Rust Basics (1/2)

Rust book:
https://doc.rust-lang.org/book/

Variable declarations

```
let a = 42;        // immutable
let mut b = 0;     // mutable
let c: i32 = 5;    // optional type annotation
```

Function definitions

Result type (omit if void)

```
pub fn some_function(x: i32, y: i32) -> i32 {
    let z = x + y;
    z + 1
}
```

Visibility (public)

No distinction between statements and expression:
Everything is an expression
; sequences expressions
No return needed (but possible)
No ; at the end!

Important types

- signed ints: `i8, i16, i32, i64, i128, isize`
- unsigned ints: `u8, u16, u32, u64, u128, usize`
- floats: f32, f64
- `bool, char`
- borrows: `&T, &mut T`
- vectors: `Vec<T>, &[T]`
- strings: `String, &str`

Type of string literals

Rust Basics (2/2)

Structs

```
#[derive(Debug, PartialEq)]
pub struct Point2D {
    x: i32,
    y: i32
}
```

```
// Construct value
Point2D {x: x, y: y}
```

Rust Basics (2/2)

Structs

```
#[derive(Debug, PartialEq)]
pub struct Point2D {
    x: i32,
    y: i32
}
```

```
// Construct value
Point2D {x: x, y: y}
```

Control structures

```
if a == 0 {
    println!("zero");
} else {
    println!("not zero");
}
for x in vec![1,2,3] {
    println!("{}", x);
}
while a > 0 {
    a = a - 1;
}
```

```
match res {
    None => 1,
    Some(i) => i
}
```

Rust Basics (2/2)

Structs

```
#[derive(Debug, PartialEq)]
pub struct Point2D {
    x: i32,
    y: i32
}
```

```
// Construct value
Point2D {x: x, y: y}
```

Control structures

```
if a == 0 {
    println!("zero");
} else {
    println!("not zero");
}
for x in vec![1,2,3] {
    println!("{}", x);
}
while a > 0 {
    a = a - 1;
}
```

```
match res {
    None => 1,
    Some(i) => i
}
```

Important expression forms

- function call: `some_function(1, 42)`
- arithmetic: `1 + 2 * 7`
- boolean logic: `true || (x < 1 && false)`
- references: `&T, &mut T`
- vectors: `Vec<T>, &[T]`
- strings `"zero"`
- string formatting `println!("{}", x);`
  `println!("{:?}", x);`
- panic `panic!("crashing");`
- struct values `Point2D { x: x, y: y}`

Development infrastructure

- Cargo: tool for building, testing, benchmarking, profiling, running and packaging Rust code
  - `cargo run`
  - `cargo test`
- Rust package registry: `https://crates.io`
- Rust playground: `https://play.rust-lang.org`
- IDE: Visual Studio Code
  - Use the `rust-analyzer` extension, **not** the `Rust` extension!
  - For debugging: extension `CodeLLDB`

Coding Conventions

Namen

- `snake_case`
  - variables
  - functions/methods
  - macros
  - crates / modules
- `UpperCamelCase`
  - structs, enums, types
  - traits
  - enum variants
- `SCREAMING_SNAKE_CASE`
  - constants
  - static variables

```
if a == 0 {
    println!("zero");
} else {
    println!("not zero");
}
```

Style

- No newline before opening brace `{`
- Closing brace `}` on a line on its own (except with `else`)
- Indent using 4 spaces
- Do not use `return` unless necessary
- Style guide: `https://github.com/rust-dev-tools/fmt-rfcs/blob/master/guide/guide.md`
- `rustfmt`: Tool for checking or fixing code style

Practice Session 1

Material available on github:

https://github.com/skogsbaer/rust_class_ew2022

Lecture 2

High-level language constructs

Enums

- More powerful than in C/C++
- Each alternative may carry values
- Like algebraic datatypes in functional languages
- Support **pattern matching**
- No NULL in Rust
- **Type-safe optional values:** `Option<T>`

Type parameter

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

Enums

- More powerful than in C/C++
- Each alternative may carry values
- Like algebraic datatypes in functional languages
- Support **pattern matching**
- No NULL in Rust
- **Type-safe optional values:** `Option<T>`

Type parameter

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

Type system force us to deal with the **None** case

Example: method `get` for `HashMap<K, V>`

```
fn get(&self, k: &K) -> Option<&V>
```

```
let hm: HashMap<String, i32> = ...;
let foo_val = match hm.get("foo") {
    None => 0,
    Some(i) => i
};
```

More Enums

Example: representing JSON

Recursion

```
enum Json {
    Object(HashMap<String, Json>),
    Array(Vec<Json>),
    String(String),
    Number(f64),
    Bool(bool),
    Null,
}
```

```
fn is_primitive(j: &Json) -> bool {
    match j {
        Json::Object(_) | Json::Array(_) => false,
        _ => true
    }
}
```

Error handling

1. Unexpected problems: `panic!("some bug")`
   - Aborts the current thread
   - Catching is possible but discouraged.
2. Expected problems: **`Result`** type
   - Examples: Working with files, network operations, ...
   - Result is either **`Ok`** or **`Err`**
   - Ways to handle the result:
     - Pattern matching
     - Error propagation via ?
     - `unwrap` (crashes, don't use)

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

Pattern Matching on `Result`

Example: TCP server

from Rust's stdlib

```
type io::Result<T> = Result<T, io::Error>; // specialized result for io

fn TcpListener::bind(addr: &str) -> io::Result<TcpListener>
fn TcpListener::accept(&self) -> io::Result<(TcpStream, SocketAddr)>
fn TcpStream::write(&mut self, buf: &[u8]) -> Result<usize>
```

```
fn start_tcp_server_1() -> io::Result<()> {
    match TcpListener::bind("127.0.0.1:7878") {
        Err(err) => Err(err),
        Ok(listener) => {
            match listener.accept() {
                Err(err) => Err(err),
                Ok((mut stream, _)) => {
                    match stream.write(&[1]) {
                        Err(err) => Err(err),
                        Ok(bytes_written) =>
                            if bytes_written == 1 {
                                Ok(())
                            } else {
                                Err(io::Error::new(io::ErrorKind::Other, ""))
                            }
                    }
                }
            }
        }}}}
```

Error propagation for `Result`

from Rust's stdlib

```
type io::Result<T> = Result<T, io::Error>; // specialized result for io

fn TcpListener::bind(addr: &str) -> io::Result<TcpListener>
fn TcpListener::accept(&self) -> io::Result<(TcpStream, SocketAddr)>
fn TcpStream::write(&mut self, buf: &[u8]) -> Result<usize>
```

```
fn start_tcp_server_2() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:7878")?;
    let (mut stream, _) = listener.accept()?;
    let bytes_written = stream.write(&[1])?;
    if bytes_written == 1 {
        Ok(())
    } else {
        Err(io::Error::new(io::ErrorKind::Other, "error"))
    }
}
```

? *propagates errors to the caller, unpacks* **Ok** *values*

Panic for `Result`

from Rust's stdlib

```
type io::Result<T> = Result<T, io::Error>; // specialized result for io

fn TcpListener::bind(addr: &str) -> io::Result<TcpListener>
fn TcpListener::accept(&self) -> io::Result<(TcpStream, SocketAddr)>
fn TcpStream::write(&mut self, buf: &[u8]) -> Result<usize>
```

```
fn start_tcp_server_3() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let (mut stream, _) = listener.accept().unwrap();
    let bytes_written = stream.write(&[1]).unwrap();
    if bytes_written != 1 {
        panic!("error")
    }
}
```

`unwrap` *causes a panic*

Don't use

Closures

- Anonymous functions
- May refer to variables from the context
- Can be passed around as arguments and results

```
let opt1 = Some("some string");
let opt2 = opt1.map(|s| s.len());
// opt2 is Some(11)
```

Parameter of closure

Body of closure, you can also enclose the body in `{ ... }`

Traits

from Rust's stdlib

```
// debugging output
trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

// user-facing output
trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

Rust can implement several standard traits automatically

Standard trait for equality

```
#[derive(Debug, PartialEq)]
pub struct Point2D {
    x: i32,
    y: i32
}

impl Display for Point2D {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result
{
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Manual implementation

Traits

- Similar to interfaces (but different)
- Method signatures for shared behavior

from Rust's stdlib

```
// debugging output
trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}

// user-facing output
trait Display {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

Rust can implement several standard traits automatically

Standard trait for equality

```
#[derive(Debug, PartialEq)]
pub struct Point2D {
    x: i32,
    y: i32
}

impl Display for Point2D {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result
{
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Manual implementation

Iterators

Usage

```
let v = vec![1,2,3];
print_all(v.iter());
```

`print_all` *is a generic method. It works for all iterators over* `T` *where* `T` *implements the* `Display` *trait.*

```
fn print_all<T, I>(iter: I) where I : Iterator<Item=T>, T: Display {
    for (i, x) in iter.enumerate() {
        println!("Element at index {}: {}", i, x);
    }
}
```

Iterators

- A stream of values
- Used for iterating over elements of a collection
- Implemented by many types in the Rust standard library
- Can be implemented for custom types
- `Iterator` is a **trait** (similar to an interface)

Usage

```
let v = vec![1,2,3];
print_all(v.iter());
```

`print_all` *is a generic method. It works for all iterators over* T *where* T *implements the* `Display` *trait.*

```
fn print_all<T, I>(iter: I) where I : Iterator<Item=T>, T: Display {
    for (i, x) in iter.enumerate() {
        println!("Element at index {}: {}", i, x);
    }
}
```

String Formatting

```rust
format!("Hello, {}!", "world");    // => "Hello, world!"
format!("The number is {}", 1);    // => "The number is 1"
format!("{:?}", (3, 4));           // => "(3, 4)"
format!("{value}", value=4);       // => "4"
let people = "Rustaceans";
format!("Hello {people}!");        // => "Hello Rustaceans!"
format!("{:04}", 42);              // => "0042" with leading zeros
```

String Formatting

- `println!` prints to stdout
- `format!` returns the formatted string
- `{}` is replaced by argument that implements `Display`
- `{:?}` is replaced by argument that implements `Debug`
- Details: `https://doc.rust-lang.org/std/fmt/`

```
format!("Hello, {}!", "world");    // => "Hello, world!"
format!("The number is {}", 1);    // => "The number is 1"
format!("{:?}", (3, 4));           // => "(3, 4)"
format!("{value}", value=4);       // => "4"
let people = "Rustaceans";
format!("Hello {people}!");        // => "Hello Rustaceans!"
format!("{:04}", 42);              // => "0042" with leading zeros
```

Strings

std::string in C++

char* in C++, but slightly more sophisticated

```
fn play_with_strings() {
    let mut s = String::from("hello"); // turns a &str into String
    s.push_str(" world");
    println!("{}", s); // prints hello world
    let (first, second): (&str, &str) = s.split_at(5);
    println!("first={first}, second={second}"); // prints first=hello, second= world
    call_me(&s);
    call_me("foo")
}

fn call_me(s: &str) { }
```

automatic conversion between `&String` *and* `&str`

Strings

- `String` heap-allocated, mutable/growable strings

std::string in C++

- `&str` string slice: immutable reference to some string data
    - view into parts of some `String` (string data lives in heap)
    - string literal (string data lives in binary's text section)

char* in C++, but slightly more sophisticated

```
fn play_with_strings() {
    let mut s = String::from("hello"); // turns a &str into String
    s.push_str(" world");
    println!("{}", s); // prints hello world
    let (first, second): (&str, &str) = s.split_at(5);
    println!("first={first}, second={second}"); // prints first=hello, second= world
    call_me(&s);
    call_me("foo")
}

fn call_me(s: &str) { }
```

automatic conversion between `&String` *and* `&str`

Vectors

```
let mut v = Vec::new(); // type of v: Vec<i32>
v.push(1);
v.push(2);
v.push(3);
let sub = v.get(0..2); // type of sub: Option<&[i32]>
println!("v={v:?}, sub={sub:?}"); // prints v=[1, 2, 3], sub=Some([1, 2])
let v2 = vec![4,5,6];  // type of v: Vec<i32>
```

range from index 0 to 2 (exclusive)

Vectors

- `Vec<T>` heap-allocated, mutable and growable arrays of T
- `&[T]` slice: immutable reference to some vector data

```
let mut v = Vec::new(); // type of v: Vec<i32>
v.push(1);
v.push(2);
v.push(3);
let sub = v.get(0..2); // type of sub: Option<&[i32]>
println!("v={v:?}, sub={sub:?}"); // prints v=[1, 2, 3], sub=Some([1, 2])
let v2 = vec![4,5,6];  // type of v: Vec<i32>
```

range from index 0 to 2 (exclusive)

More high-level features

- Macros
  - More powerful than in C/C++
  - Macro body partially checked by the compiler
  - Macro invocations end with `!`
    - `println!`
    - `panic!`
    - `format!`
    - `vec!`
- Module system
  - Fine-grained visibility rules

Performance

- **Performance of idiomatic Rust code is comparable to C/C++**
- Sometimes it's even faster
  - Invariants checked dynamically in C/C++ are checked statically in Rust
- Bechmarks:
  - Constructing an UTF-16 string from a bytearray: Rust is slightly faster than C
  - Parsing JSON with parser combinators: Rust is faster than nodejs
  - Bezier Benchmark aus dem OABench 2.0: Rust and C perform pretty much the same

Performance

- **Performance of idiomatic Rust code is comparable to C/C++**
- Sometimes it's even faster
  - Invariants checked dynamically in C/C++ are checked statically in Rust
- Bechmarks:
  - Constructing an UTF-16 string from a bytearray: Rust is slightly faster than C
  - Parsing JSON with parser combinators: Rust is faster than nodejs
  - Bezier Benchmark aus dem OABench 2.0: Rust and C perform pretty much the same

Rust

```rust
fn boring(k: i32) -> i32 {
    let mut result = 0;
    for j in (k..).step_by(2).zip(1..5) {
        result += j.0 + j.1;
    }
    result
}
```

LLVM (C syntax)

```c
int boring2(int i) {
    int j = i << 1;
    int k = j + 5;
    int l = i << 1;
    int m = k + l;
    int n = m + 17;
    return n;
}
```

Practice Session 2

Material available on github:

https://github.com/skogsbaer/rust_class_ew2022

Lecture 3

Advanced concepts

Lifetimes

- Example from part 1

- Automatic management of lifetimes, most of the time
- Explicit lifetime variables: `'a` `'b` `'c` ...

Lifetimes

- Example from part 1

```
let mut v = vec![10, 11];
let vptr = &v[1];
v.push(12);
println!("{}", *vptr);
```

- *Declaration of push:* `fn push(v: &mut Vec<i32>, i: i32)`
- `v.push(12)` *is short for* `push(&mut v, 12)`

- Automatic management of lifetimes, most of the time
- Explicit lifetime variables: `'a` `'b` `'c` ...

Lifetimes

- Example from part 1

```
let mut v = vec![10, 11];
let vptr = &v[1];
v.push(12);
println!("{}", *vptr);
```

Lifetime of borrow `vptr`

Lifetime of mutable borrow for calling `push`

Overlapping lifetimes of two borrows, one borrow is mutable.

`Compile error: cannot borrow v as mutable because it is also borrowed as immutable`

- *Declaration of push:* `fn push(v: &mut Vec<i32>, i: i32)`
- `v.push(12)` *is short for* `push(&mut v, 12)`

- Automatic management of lifetimes, most of the time
- Explicit lifetime variables: `'a` `'b` `'c` ...

Explicit Lifetimes

Rust infers: `x: &'a str, y: &'b str` *Need the shorter lifetime here. But is* `'a` *shorter or* `'b`*?*

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
```

Compile error: missing lifetime specifier

Explicit Lifetimes

Rust infers: `x: &'a str, y: &'b str` *Need the shorter lifetime here. But is* `'a` *shorter or* `'b`?

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}
```

Compile error: missing lifetime specifier

- Solution: explicit lifetimes
- `&'a str` means: a reference that lives at least as long as lifetime `'a`

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Heap-allocated data

- Consider implementing a linked list

```
pub struct Node<T> {
    data: T,
    next: Option<Node<T>>
}
```

Heap-allocated data

- Consider implementing a linked list

```
pub struct Node<T> {
    data: T,
    next: Option<Node<T>>
}
```

```
Compile error: recursive type
`Node` has infinite size
```

- Struct values live on the stack
- Size must be known to the compiler

Heap-allocated data

- Consider implementing a linked list

```
pub struct Node<T> {
    data: T,
    next: Option<Node<T>>
}
```

Compile error: recursive type `Node` has infinite size

- Struct values live on the stack
- Size must be known to the compiler

```
pub struct Node<T> {
    data: T,
    next: Option<Box<Node<T>>>
}
fn play_with_node() {
    let n1 = Node { data: 42, next: None };
    let n2 = Node { data: 10, next: Some(Box::new(n1)) };
    // Owner n2 goes out of scope => automatic free
}
```

- Heap-allocated data to the rescue
- `Box<T>` places value of type T in the heap
- **Automatic free** once the owner of the box is dropped

Shared mutable state

- Rust's ownership and borrowing is sufficient for many programming idioms.
- But it prevents **shared mutable state.**
    - Doubly-linked lists
    - Sharing writeable data between threads
- Still, you can have shared mutable state in Rust.
- *Example:* `Mutex` allows to share mutable data between threads.

Shared mutable state

- Rust's ownership and borrowing is sufficient for many programming idioms.
- But it prevents **shared mutable state.**
  - Doubly-linked lists
  - Sharing writeable data between threads
- Still, you can have shared mutable state in Rust.
- *Example:* `Mutex` allows to share mutable data between threads.

```
let mutex_v = Mutex::new(vec![10, 11]);
join(|| { let guard = mutex_v.lock().unwrap();
          let v = guard.deref();
          println!("v[1] = {}", v[1]); },
     || { let mut guard = mutex_v.lock().unwrap();
          guard.deref_mut().push(13) });
```

Mutex unlocked once guard goes out of scope.

Unsafe Rust

- How can `Mutex` be realized?
- *Option 1:* make the typesystem more powerful (read: more complicated)
- *Option 2:* realize `Mutex` as a builtin construct
- *Option 3:* **unsafe code with a safe API**
  - Approach used by Rust
  - Unsafe code has superpowers
    - Use raw pointers
    - Invoke unsafe functions
  - **Unsafe code is the exception not the rule**

Safe API for Mutex

Mutex<T> is public, data field only accessible from the same module

Lifetime parameter

Marker trait: Mutex<T> can be shared between threads

Success or error

```
guard.deref() -> &'a T
guard.deref_mut() -> &'a mut T
```

releases the lock when dropped

```
pub struct Mutex<T> {
  data: UnsafeCell<T>
}

impl<T> Mutex<T> {

    pub fn lock<'a>(self: &'a Mutex<T>) -> LockResult<MutexGuard<'a, T>> {
     unsafe {
       // Superpowers: dereference raw pointers, call unsafe functions ...
     }
  }
  // more functions ...
}

unsafe impl<T: Send> Sync for Mutex<T> {}
```

Correctness guarantees

- *Without* `unsafe`: **100% memory safe** and **free of data races.**
  - Verified proof for a significant subset of Rust (RustBelt project)
- *With* `unsafe`: same guarantees but
  - `unsafe` features must be wrapped in a safe API, and
  - there must be a proof that the `unsafe` code satisfies the safe API.
    - The RustBelt project did such proofs for several abstractions of Rust's standard library, e.g. `Mutex`
    - Complicated!
- **Miri:** tool to test your program against undefined behavior

Smart Points for shared mutable state

- **Rc<T>**
  - reference-counting pointer
  - allows multiple owners (immutable)
  - memory deallocated when the last owner goes out of scope
  - single-threaded
- **Arc<T>**
  - thread-safe variant of Rc<T>

- **RefCell<T>**
  - mutable memory location
  - borrow rules are checked dynamically
- **Weak<T>** weak pointer
- **Mutex<T>** allows to share mutable data between threads.

Practice Session 3

Material available on github:

https://github.com/skogsbaer/rust_class_ew2022

Summary

Resources

Start with this book!

- Steve Klabnik and Carol Nichols: *The Rust Programming Language.* 2019, `https://doc.rust-lang.org/book`
- *Embedded Rust documentation:* `https://docs.rust-embedded.org`
- *Microsoft documentation:* `https://docs.microsoft.com/de-de/learn/paths/rust-first-steps/`
- *Learn Rust With Entirely Too Many Linked Lists.* `https://rust-unofficial.github.io/too-many-lists/`

Advanced material! work through this tutorial if you really want to understand ownership, borrows, various smart pointers and unsafe code.

Rust Summary

- **Safety** and **control**
- No unrestricted combination of **aliasing** and **mutation**
- Strong type system: **100% memory safe**
- **Unsafe code** embedded within a **safe API**
- **High-level** language features
- Very good **performance**

Rust Summary

- **Safety** and **control**
- No unrestricted combination of **aliasing** and **mutation**
- Strong type system: **100% memory safe**
- **Unsafe code** embedded within a **safe API**
- **High-level** language features
- Very good **performance**

- Steep learning curve (but you don't need a PhD!)
- Open-source licence: Apache and MIT
- Developed since 2009 mainly at Mozilla.
- Rust Foundation since 2021, founded by AWS, Huawei, Google, Microsoft und Mozilla.