Unsafe und das Foreign Function Interface

Mit fremdem Code sprechen

Jonas Schievink

Programmieren in Rust

Recap: Raw Pointers

- `*const T` und `*mut T` sind „Raw Pointer“ auf ein `T`
- Intern wie Referenzen
- `NULL` erlaubt (`std::ptr::null()`)
- Sind `Copy`, aber nicht `Send` oder `Sync`
- Kein Lifetime-Tracking
- Konvertierung von/zu `usize` (`0xabcdef as *const ()`)
- Konvertierung von Referenzen (Automatisch als Coercion)
- Muss nicht unbedingt dereferenzierbar sein

$\Rightarrow$ Dereferenzieren nicht in „safe“ Rust möglich

Unsafe

- `unsafe { /* ... */ }`-Block in Funktionen
- `unsafe fn danger(...) { ... }`
- Erlauben von genau 3 Operationen, die der Compiler sonst verweigert:
  - Lesen / Schreiben von `static mut` Variablen
  - Raw Pointer dereferenzieren
  - Als `unsafe` markierte Funktionen aufrufen
- Borrowchecker und Typsystem bleiben aber intakt!
- Außerdem (wird gerne vergessen): Markieren von Traits als `unsafe`
- `unsafe trait Send {}`
- `unsafe impl` zum Implementieren nötig

Aussagen von Unsafe

- `unsafe fn`
  - „Zum Aufrufen musst du spezielle Invarianten einhalten“
- `unsafe trait`
  - „Dieser Trait darf nur von Typen implementiert werden, die meine Invarianten einhalten“
- Bezieht sich auf Invarianten, die nicht mit Rust’s Typsystem erzwungen werden (können)
- Genauer beschrieben in Dokumentation (hoffentlich!)

$\Rightarrow$ Werden sie nicht eingehalten, *so ist das Verhalten undefiniert*

Undefined Behaviour (UB)

- Hauptsächlich bekannt aus C/C++
- “When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose”
- **Jedes** Verhalten entspricht der Spezifikation
- Beinahe unmöglich zu erkennen (auch vom Compiler)
- Aber: Rust-Compiler verhindert UB in sicherem Code (Sprache ist so designed)
- Voraussetzung: Kein `unsafe`-Code verursacht UB (und der Compiler hat keinen Bug)

UB in Rust

Wenn...

- Ein **`&str`** kein gültiges UTF-8 enthält
- Ein Data Race auftritt
- Ein **`bool`** andere Werte als 0 (**`false`**) oder 1 (**`true`**) enthält
- Ein Raw Pointer dereferenziert wird der NULL oder *dangling* ist
- Eine Referenz (**`&T`**/**`&mut T`**) nicht dereferenzierbar ist (NULL oder *dangling* ist) - auch ohne sie zu dereferenzieren
- Die Alias-Garantien von Rust / LLVM nicht eingehalten werden (kompliziert!)
- ...

... *so ist das Verhalten undefiniert*

(trotzdem immer noch weniger Fälle als bei C++)

Warum also Unsafe?

- Der Compiler ist schlau, aber selten (sehr, sehr selten!!!) weiß es der Programmierer doch besser
- Einige Dinge entziehen sich der Macht des Compilers: Code aus fremden Libraries (später mehr dazu), primitive Speicheroperationen
- `unsafe` ist nötig, um viele Grundbausteine herzustellen (wie ist `Vec` implementiert?)

Warum also Unsafe?

```
struct Vec<T> {
    len: usize,
    cap: usize,
    ptr: *mut T,
}
```

- Wie **Vec** funktioniert wisst ihr: Länge, Kapazität und Pointer auf die eigentlichen Daten
- Low-Level Speicheroperationen sind nicht in sicherem Rust möglich
- Intern **unsafe**, Implementierung sorgt dafür, dass alles sicher bleibt

Auswirkungen von Unsafe Rust

```
impl<T> Vec<T> {
    pub fn set_length(&mut self, length: usize) {
        self.len = length;
    }
}
```

Auswirkungen von Unsafe Rust

```
impl<T> Vec<T> {
    pub fn set_length(&mut self, length: usize) {
        self.len = length;
    }
}
```

- Kein **`unsafe`** in Sicht
- Diese (völlig sichere) Funktion erlaubt **UNDEFINED BEHAVIOUR**

Auswirkungen von Unsafe Rust

```
impl<T> Vec<T> {
    pub fn set_length(&mut self, length: usize) {
        self.len = length;
    }
}
```

- Kein **`unsafe`** in Sicht
- Diese (völlig sichere) Funktion erlaubt **UNDEFINED BEHAVIOUR**

Rust ist also doch unsicher, dann kann ich ja auch wieder C++ benutzen...

Auswirkungen von Unsafe Rust

```
impl<T> Vec<T> {
    pub fn set_length(&mut self, length: usize) {
        self.len = length;
    }
}
```

- Kein **unsafe** in Sicht
- Diese (völlig sichere) Funktion erlaubt **UNDEFINED BEHAVIOUR**

~~Rust ist also doch unsicher, dann kann ich ja auch wieder C++ benutzen...~~

Nein!

Auswirkungen von Unsafe Rust

- `unsafe`-Code trifft gewisse Annahmen und verlangt die Einhaltung von Invarianten!
- Gute und korrekte Dokumentation ist wichtig
- Im Falle von `Vec`: Elemente `0..self.len` sind gültig (u. A.)
- Wird `Vec::set_length` mit `length > self.len` aufgerufen, *so ist das Verhalten undefiniert*

⇒ „Sichere Abstraktion“ (safe abstraction), `unsafe`-Teile bleiben verborgen

Was ist sonst alles Unsafe?

- `extern`-Funktionen (später mehr)
- `Vec::set_len` (die gibt es nämlich wirklich)
- `Vec::from_raw_parts` zum Erzeugen eines Vektors aus Länge, Kapazität und Datenpointer
- `String::from_utf8_unchecked`, konvertiert von `Vec<u8>` nach `String`, ohne auf gültiges UTF-8 zu prüfen
- Vieles in `std::mem`
  - `transmute()` ändert den Typ eines Wertes
  - `uninitialized()` erzeugt einen Wert mit undefiniertem Bitmuster
  - `zeroed()` erzeugt einen Wert aus Null-Bytes

mem::transmute()

```
pub unsafe fn transmute<T, U>(e: T) -> U
```

„Reinterprets the bits of a value of one type as another type.“

- Ungeprüfte Konvertierung zwischen **allen** Typen
- „`transmute` is **incredibly** unsafe. There are a vast number of ways to cause undefined behavior with this function. `transmute` should be the absolute last resort.“
- Nützlich für „type punning“ (Aushebeln des Typsystems weil wir Coole Sachen™ machen wollen)
- Eingabewert muss gültiges Bitmuster für Ausgabe haben, *sonst ist das Verhalten undefiniert*

mem::transmute()

```
pub unsafe fn transmute<T, U>(e: T) -> U
```

„Reinterprets the bits of a value of one type as another type.“

Beispiel: Konvertieren zwischen **f32** und **u32**. Bei beiden Typen sind alle Bitmuster gültige Werte, also sollte das gehen.

```
let bits: u32 = unsafe { mem::transmute(6.283185f32) };

println!(
    "{:b} {:b} {:b}",
    bits >> 31,
    (bits >> 23) & 0xFF,
    bits & 0x7F_FF_FF,
)
```

Ausgabe:

```
0 10000001 10010010000111111011010
```

Zusammenfassung Unsafe Rust

+ Erlaubt Implementierung von Low-Level Datenstrukturen
+ Erlaubt es, Features zu nutzen, die über das Typsystem hinaus gehen
+ Erlaubt die Interaktion mit OS / „unsicheren“ Libraries
+ Direktes Ansprechen von Hardware (Mikrocontroller)
- Schwer zu meistern (fast so schwer wie C++)
- Rust hat (noch) kein Memory Model (Was genau ist gültiger `unsafe`-Code?)
- Auswirkungen auf sicheren Code möglich (`set_length`)

Wie nutzt man `unsafe` richtig? Rustonomicon!

Rust's Foreign Function Interface (FFI)

Fremden Code in Rust nutzen

- Gegen fremde Bibliotheken linken
- Datentypen in Rust nutzbar machen
- Funktionen aus Rust aufrufen

Rust aus anderen Sprachen nutzen

- C-kompatibles `struct`-Layout erzwingen
- Funktionen aus Fremdsprachen aufrufbar machen
- Symbolnamen festlegen

Motivation

Warum mit den Anderen reden?

- Integration in bestehende Codebase: Code kann stückweise nach Rust portiert werden
- Nutzung bestehender Libraries aus Rust
- Nutzung von Rust-Library aus anderer Sprache (C, C++, Python, Lua, ...; Alles mit C-kompatiblem FFI)

Fremde Bibliotheken nutzen

```c
/**
 * \brief Calculates the n-th fibonacci number
 * \param n Number to calculate
 * \returns The calculated fibonacci number
 */
uint64_t libfoo_fibonacci(uint64_t n);
```

Fremde Bibliotheken nutzen

```rust
#[link(name = "foo")]
extern {
    fn libfoo_fibonacci(n: u64) -> u64;
    // ...
}
```

- **`extern`**-Block deklariert Funktionen aus Library
- Alle Funktionen automatisch **`unsafe`**
- **`#[link]`**-Attribut enthält den Namen der Library
- **`gcc -lfoo rust_program.o -o rust_program`**
- Linkeraufruf anzeigen mit: **`rustc -Zprint-link-args`** (lang!)

Fremde Bibliotheken nutzen

```c
void libfoo_print_me(const char *s);
```

In Rust:

```rust
#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const char);
}
```

```
warning: found Rust type `char` in foreign module, while
`u32` or `libc::wchar_t` should be used,
#[warn(improper_ctypes)] on by default
```

Das wäre fast schiefgegangen (Undefined Behaviour)! Danke lieber Compiler!

(deinen Vorschlag lehnen wir in diesem Fall aber dankend ab)

Fremde Bibliotheken nutzen

```c
void libfoo_print_me(const char *s);
```

In Rust:

```rust
#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const char);
}
```

- **`char`** hier nicht richtig! C-chars sind 8 Bits groß, Rust-chars sind Unicode-Skalare/Codepoints (32 Bit)!

Fremde Bibliotheken nutzen

```c
void libfoo_print_me(const char *s);
```

In Rust:

```rust
#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const char);
}
```

- **`char`** hier nicht richtig! C-chars sind 8 Bits groß, Rust-chars sind Unicode-Skalare/Codepoints (32 Bit)!
- **`u8`** oder besser **`i8`**? C-Standard erlaubt beides!

Fremde Bibliotheken nutzen

```c
void libfoo_print_me(const char *s);
```

In Rust:

```rust
#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const char);
}
```

- `char` hier nicht richtig! C-chars sind 8 Bits groß, Rust-chars sind Unicode-Skalare/Codepoints (32 Bit)!
- `u8` oder besser `i8`? C-Standard erlaubt beides!
- Zum Glück sind wir nicht die Ersten mit diesem Problem...

extern crate libc;

`libc` stellt C-Typen als die richtigen `type`-Aliase bereit (je nach Plattform):

- `libc::c_char` entspricht dem C-Typ `char`!
- Außerdem: `libc::{c_int, c_short, c_ulonglong}` = `{int, short, unsigned long long}`
- (`libc` macht noch viele andere tolle Sachen, dazu später mehr)

Fremde Bibliotheken nutzen

```
void libfoo_print_me(const char *s);
```

In Rust:

```
extern crate libc;

#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const libc::c_char);
}
```

Besser!

Strings

```
extern crate libc;

#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const libc::c_char);
}
```

Falsche Verwendung:

```
let s = "This program might eat your laundry";
unsafe {
    libfoo_print_me(s.as_ptr());
}
```

`str::as_ptr()` gibt einen `*const u8` auf den String zurück. Aber: Rust-Strings können 0-Bytes enthalten und sind nicht NULL-Terminiert (enden nicht immer mit 0-Byte).

Strings

```
extern crate libc;

#[link(name = "foo")]
extern {
    fn libfoo_print_me(s: *const libc::c_char);
}
```

Richtige Verwendung:

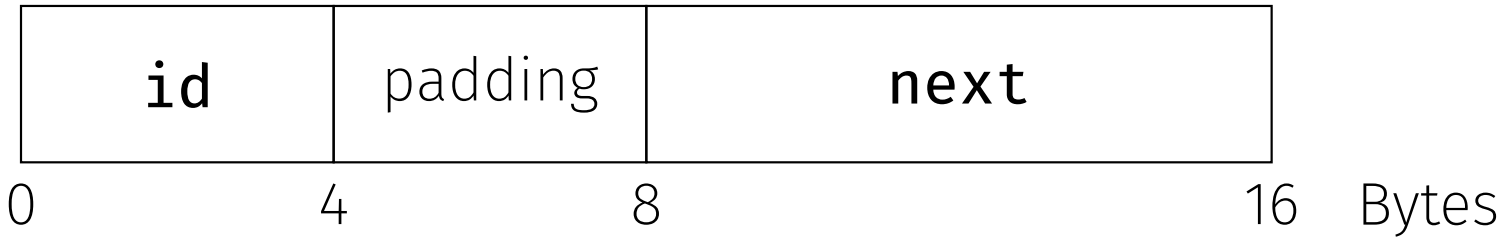
```
use std::ffi::CString;

let s = "This program might eat your laundry";
let cstring = CString::new(s).unwrap();
unsafe {
    libfoo_print_me(cstring.as_ptr());
}
```

structs

```
struct Foo {
    uint32_t id;
    struct Foo* next;
};
```

Wie sieht **Foo** im Speicher aus? (Plattformabhängig! Annahme: **x86_64** + Linux)

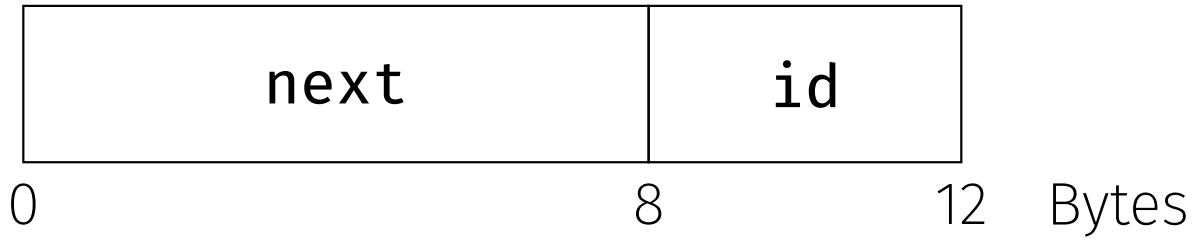


In Rust ist das Speicherlayout nicht festgelegt! Der Compiler darf Felder umsortieren und kombinieren (Optimierung).

structs

```
struct Foo {
    id: u32,
    next: *const Foo,
}
```

Mögliche Layout-Optimierung:



Rust kann durch Tauschen der Felder Speicher sparen (wird noch nicht standardmäßig gemacht).

#[repr(C)]

```
#[repr(C)]
struct Foo {
    id: u32,
    next: *const Foo,
}
```

`#[repr(C)]` deaktiviert Layout-Optimierungen und erzwingt C-kompatibles Layout.

bindgen

`bindgen` ist ein Tool zum automatischen Generieren von FFI-Bindings aus C/C++-Headern.

- Aufruf meist aus Build-Skript via Cargo (Command line geht auch)
- Generierten Rust-Code mit **`include!`** in Modul einfügen
- Code nutzt C-Namen (Warnungen deaktivieren hier okay)
- Unterstützung vieler C++-Features (!!!)
- Sogar die Doku wird mitkopiert :-)
- **Aber**: Immer noch nur Bindings, kein schönes und sicheres Rust-Interface

Konvention: Mit **`bindgen`** eine **`*-sys`** Crate erstellen, die nur die FFI-Bindings enthält, dann schöne und sichere Rust-API designen.

Wrapper-Design

Wie designe ich einen sicheren und idiomatischen Rust-Wrapper?

- Schwer. Schwerer als `unsafe` zu meistern (behaupte ich).
- Möglichst viel `unsafe` Zeug loswerden
- Alle `CString`s und Raw Pointer „wegabstrahieren“ (Rust-Typen verwenden)
- Benötigt zuerst gutes Gefühl für Rust

Das sind nur ein paar generelle Tipps, je nach Library sieht alles evtl. anders aus!

Rust aus anderen Sprachen nutzen

Ziel: Integration in andere Codebase

```
gcc -lmy_rust_lib ...
```

Symbolnamen

- Am Ende der Kompilierung: Linker löst Symbolreferenzen auf
- Symbol in C: `libfoo_create_foo`
- Symbol in C++: `_Z7b2Alloci` (entspricht Funktion `void* b2Alloc(int)`)
- Symbol in Rust: `_ZN8rust_out4main17ha208b69ccbc11839E` (entspricht `fn main()`)

⇒ C++ und Rust verwenden „Name Mangling“

Wenn Symbolnamen nicht stimmen: Linker-Fehler!

```
Undefined reference to `my_rust_function'.
```

#[no_mangle]

Als Funktionsattribut deaktiviert **#[no_mangle]** Rust's Name Mangling:

```
#[no_mangle]
pub fn my_rust_function() { /* ... */ }
```

ABIs

- Application Binary Interface, definiert:
  - Stack-Frame Layout
  - Calling Convention (Welche Argumente in welchen CPU-Registern)
  - Name-Mangling (`#[no_mangle]`)
  - Layout von Datentypen (`#[repr(C)]`)
- ABI ist plattform- und sprachenabhängig
- C-ABI ist kleinster gemeinsamer Nenner (voll spezifiziert)
- Rust-ABI unspezifiziert (Optimierungen möglich)
- Oft ist mit ABI die Calling Convention gemeint

`extern "ABI" fn`

Um die ABI / Calling Convention einer Funktion zu ändern, gibt es die **`extern "ABI" fn`** Syntax:

```
#[no_mangle]
pub extern "C" fn my_rust_function() { /* ... */ }
```

- **`"C"`** = Standard-C-ABI der Zielplattform
- Am häufigsten benutzt, daher default (entspricht **`extern fn`**)
- Richtig fürs Linken gegen C-Libraries (und C-kompatible)

Endlich ist unsere Funktion aus C nutzbar!

Crate-Typen

- Rust kennt mehrere Arten von Crates
- Bisher benutzt: **`rlib`** (Libraries), und **`bin`** (Binaries)
- Fürs Linken gegen C wichtig: **`dylib`** und **`staticlib`**
- Produktion einer dynamisch oder statisch gelinkten C-Library
- Man erhält eine **`.so`**, **`.dll`**, **`.dylib`** (dynamisch), oder **`.a`**, **`.lib`** (statisch)

In **`Cargo.toml`**:

```
[lib]
crate-type = ["dylib"]  # array allows multiple types
```

Danke für eure Aufmerksamkeit!

https://is.gd/XuGXr3