RUST 💖 C/C++

A memory-violating love story

WHOAMI(1)

Katharina Fey (`@spacekookie`)

- Active FOSS developer
- Avid tea drinker
- Hobbyist hardware maker

WHOAMI(2)

I do Rust things!

- Contributer to the CLI-WG
- Author of (too) many `use[ful|less]` crates
- Member of `berlin.rs`

WHY

WHY WOULD YOU DO THAT?

- Integrate into larger projects
- Replace application piece by piece
- Write plugins

WHY GIVE THIS TALK?

Rust promises efficient FFI to C code

What does this mean?

ABI

ABI

Application *Binary* Interface

- Defines the function signature & types
- Much like an API but for linkers

ABI

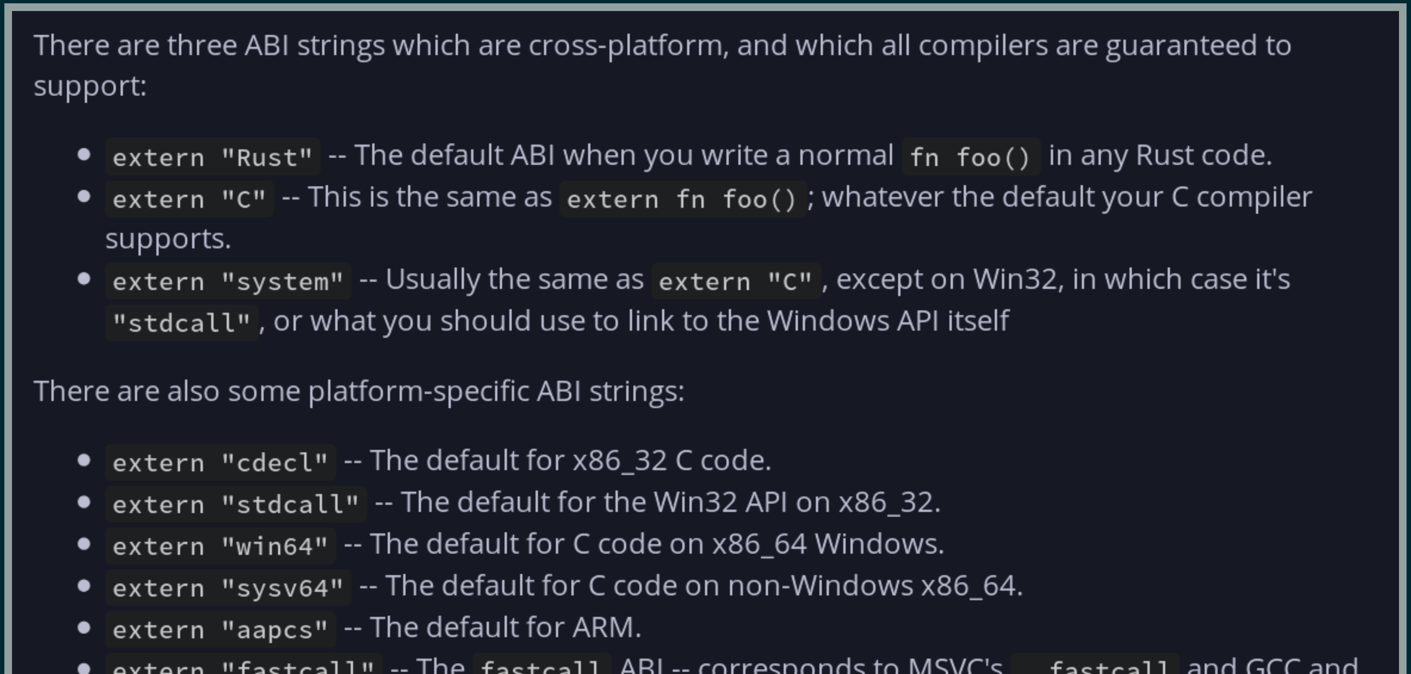
```
extern "C" fn foo() { /* ... */ }


#[repr(C)]
struct Bar { /* ... */ }

#[repr(C)]
enum Biz { /* ... */ }

union Fuzz { /* unions are just cool by default */ }
```

ABI



There are three ABI strings which are cross-platform, and which all compilers are guaranteed to support:

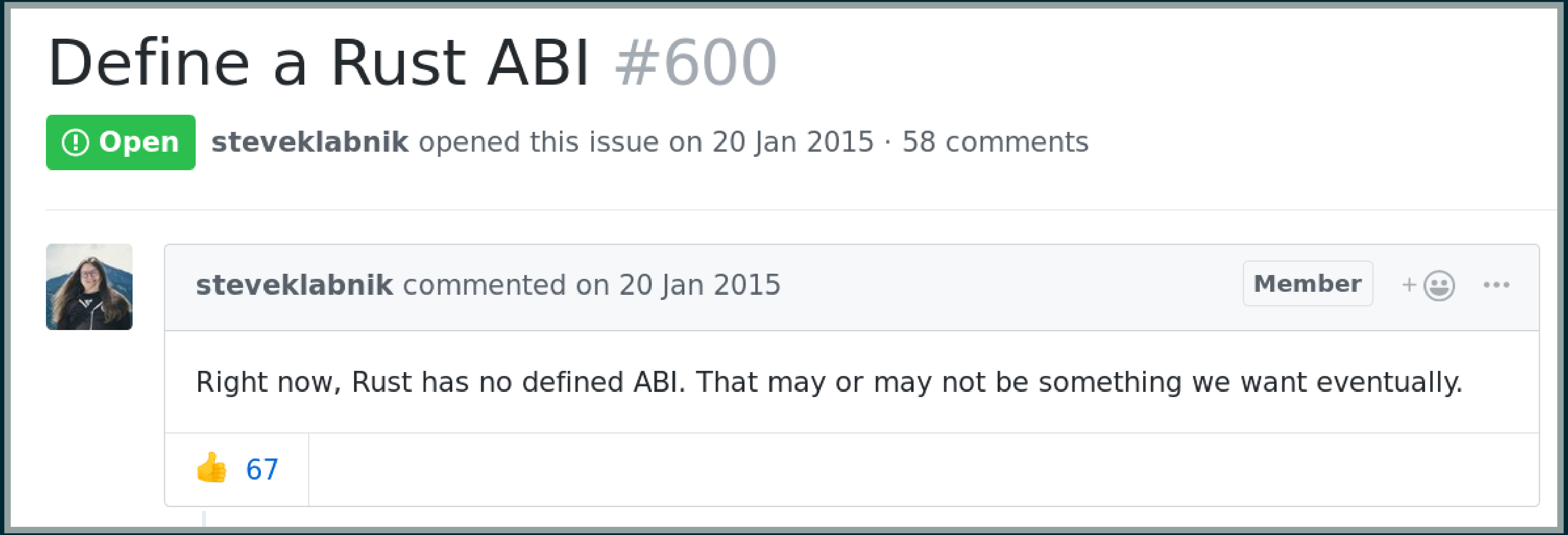
- `extern "Rust"` -- The default ABI when you write a normal `fn foo()` in any Rust code.
- `extern "C"` -- This is the same as `extern fn foo()`; whatever the default your C compiler supports.
- `extern "system"` -- Usually the same as `extern "C"`, except on Win32, in which case it's `"stdcall"`, or what you should use to link to the Windows API itself

There are also some platform-specific ABI strings:

- `extern "cdecl"` -- The default for x86_32 C code.
- `extern "stdcall"` -- The default for the Win32 API on x86_32.
- `extern "win64"` -- The default for C code on x86_64 Windows.
- `extern "sysv64"` -- The default for C code on non-Windows x86_64.
- `extern "aapcs"` -- The default for ARM.
- `extern "fastcall"` -- The `fastcall` ABI -- corresponds to MSVC's `__fastcall` and GCC and

ABI

Let's talk about stability



Define a Rust ABI #600

Open steveklabnik opened this issue on 20 Jan 2015 · 58 comments

steveklabnik commented on 20 Jan 2015 Member

Right now, Rust has no defined ABI. That may or may not be something we want eventually.

👍 67

Neither does C++

C doesn’t *have* an ABI

The operating system does

C CODE FROM RUST

BORING FFI

- Bind to native API with `extern` functions
- Wrap function calls in `unsafe`
- Make data C-compatible

```
extern "C" {
    fn reverse(const *c_char) -> const *c_char;
}

fn stuff(value: &str) {
    println!("{:?}",
        unsafe { reverse(CStr::from(value).unwrap()) }
    );
}
```

BORING FFI

`std::os::raw` & `std::ffi` contain FFI types

- (Rust) `String` becomes `CString`
- (Rust) `&str` becomes `CStr`
- `void` becomes `c_void`
- … etc …

TURNING TABLES

- Same `extern "C"` as before
- Take data in C-form
- Use `#[no_mangle]` to preserve the function name

```rust
#[no_mangle]
pub extern "C" fn reverse(word: *const c_char) -> *const c_char {
    /* ... implementation really not important right now ... */
}
```

TURNING TABLES

Some special fields in `Cargo.toml`

```
# ...

[lib]
name = "reverso"                # Practise my reversing. Ha-HAA!
crate-type = ["cdylib"]         #  dynamic library (.so)
#             ["staticlib"]        static library (.a)
```

TURNING TABLES

Integrating the Rust code into your build toolchain

```
├── CMakeLists.txt
├── reverso
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── reverso.h
└── main.c
```

Note the header `reverso.h`

```
// Safely reverse a unicode string
const char *reverse(const char *in);
```

TURNING TABLES

Calling this from C is easy

```c
#include "reverso.h"
void main() {
    char * greeting = "привет RustConf 👩‍💻";
    printf("'%s' reversed: '%s' \n", greeting, reverse(greeting))
}
```

```
'ривет RustConf 👩‍💻' reversed: '💻👩 fnoCtsuR тевирп'
```

THANK YOU

Tweet at me @spacekookie

Like, Share & Subscribe…

ALRIGHT, NOT QUITE

SOME PROBLEMS

- I don’t want to write headers
- How to deal with anything going wrong?
- Oh god, *real* memory management! 😨
- How to build pretty APIs?

TOOLING

CBINDGEN

Don’t write headers yourself. Use `cbindgen`

- Like bindgen, but in reverse
- Can generate `.h` files at compile-time

BUILD SYSTEM SUPPORT

😏

MEMORY MANAGEMENT

MEMORY MANAGEMENT

Put your troubles in a box ✨

```
#[repr(C)]
struct MyThing {
    /* ... */

}

#[no_mangle]
extern "C" fn make_thing() -> Box<MyThing> {
    Box::new(MyThing {
        /* ... */
    })
}
```

📦 BOXES 📦

```
let ptr: c_void = /* ... */;

let thing: &mut MyThing = unsafe {
        &mut *ctx as &mut MyThing
    };

thing.foo();
```

Remember: C is now responsible for the memory.

You can’t make the native code memory safe

```
void main() {
    MyThing *t = make_thing();

    free(t);

    printf("%s", t.value); // kaboom!
}
```

COMMUNICATING ERRORS

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

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

- Errors in C
- Errors in C++

Emulate Result<T, E> with a structure

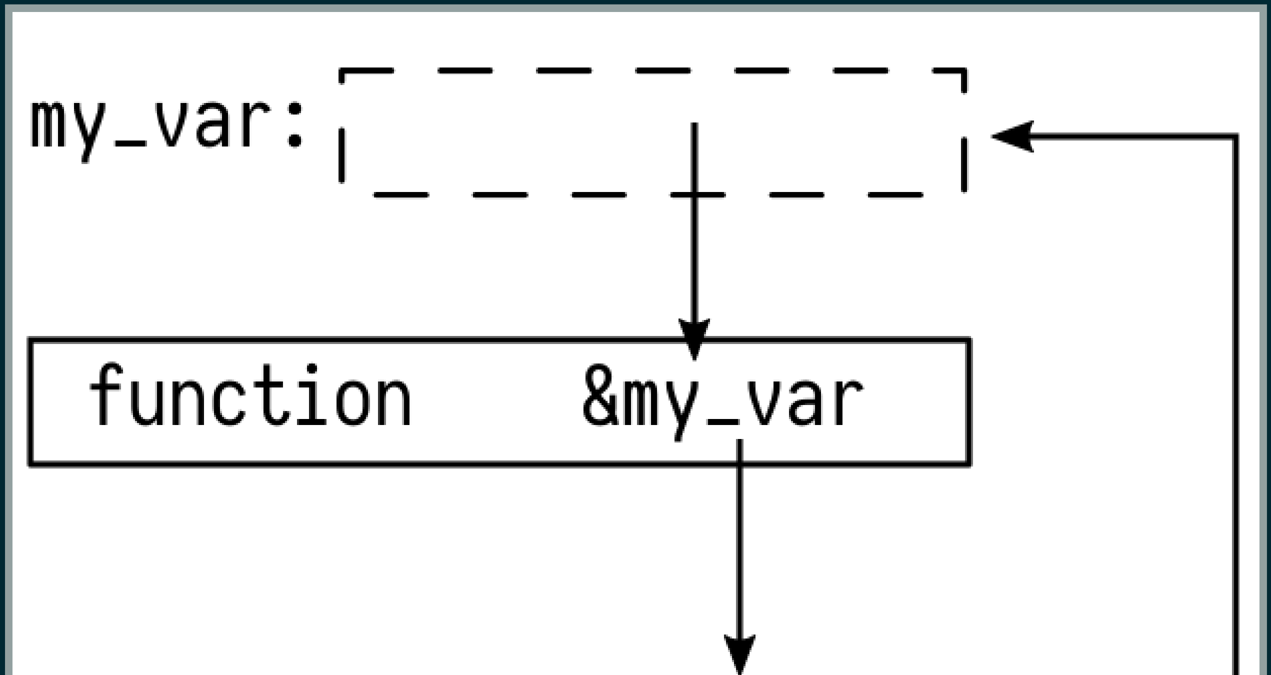
```rust
#[repr(C)]
pub struct rvalue_t<T> {
    thing: Box<Option<T>>,
    code: u32,
}
```

C

```c
struct rvalue_t {
    void         *ignore_me;
    unsigned int code;
};
```

```
rvalue_t val = myfunction();
if (val.code) {
    // Handle errors
}
```

☛ POINTERS



ERRORS IN C

```c
uint32_t get_client(server_t *ctx, client_t **client);

/* ... */

client_t *client;
ret = get_client(ctx, &client);
if(ret) {
    // Handle errors
}
```

ERRORS IN C

```c
uint32_t initialise(server_t **ctx, uint16_t port);

/* ... */

server_t *server;
ret = initialise(&server, 1337);
if(ret) {
    // Handle errors
}
```

ERRORS IN & RUST

```
/// Initialise <thing>
#[no_mangle]
pub extern "C" fn initialise(ctx: *mut *mut c_void,

                             port: c_uint) -> c_uint {

    /* ... check if port valid ... */

    let server = Box::new(server_t { port });
    unsafe { *ctx = Box::into_raw(server) as *mut c_void };
    return 0;
}
```

```c
/* rust.h */
struct server_t {
    uint16_t port;
};
uint32_t initialise(struct server_t **, uint16_t);


/* main.c */

struct server_t *server;
uint32_t ret = initialise(&server, 8080);
if(ret) {
    // Handle errors
}
```

ERRORS IN C++

Well...

ERRORS IN C++

Wrap C-errors in exception throwing code

```
extern "C" {
    #include "cbindgen-made-this.h"
}

class MyRustModule {
    void do_something_dangerous() {
        auto ret = do_rust_things();
        if(ret) throw CorporateExceptionSeven(ret);
    }
}
```

ERRORS IN C++

```
namespace Rust {
    extern "C" {
        #include "no_i_made_this.h"
    }
}

/* ... */

auto ret = Rust::do_something_dangerous();
if(ret) return new MyResultNine(ret, "Oh no!");
```

CAN YOU THROW A C++ EXCEPTION FROM RUST?

YES!

EXCEPTIONS

`try` - `throw` – `catch`

`try` creates a “landing pad”

`throw` walks up the stack

Then calls `catch`

TRY

Landing pad determines how to continue

CATCH

But which one? Filter or rethrow!

THROW

Replaced with calls into `libc++`

THIS IS A TALK ABOUT RUST

EXCEPTION.RS

```
extern crate exception_rs as exception;

pub extern "C" fn oh_no() {
    exception::throw(RustException { text: "Oh noes!" });
}
```

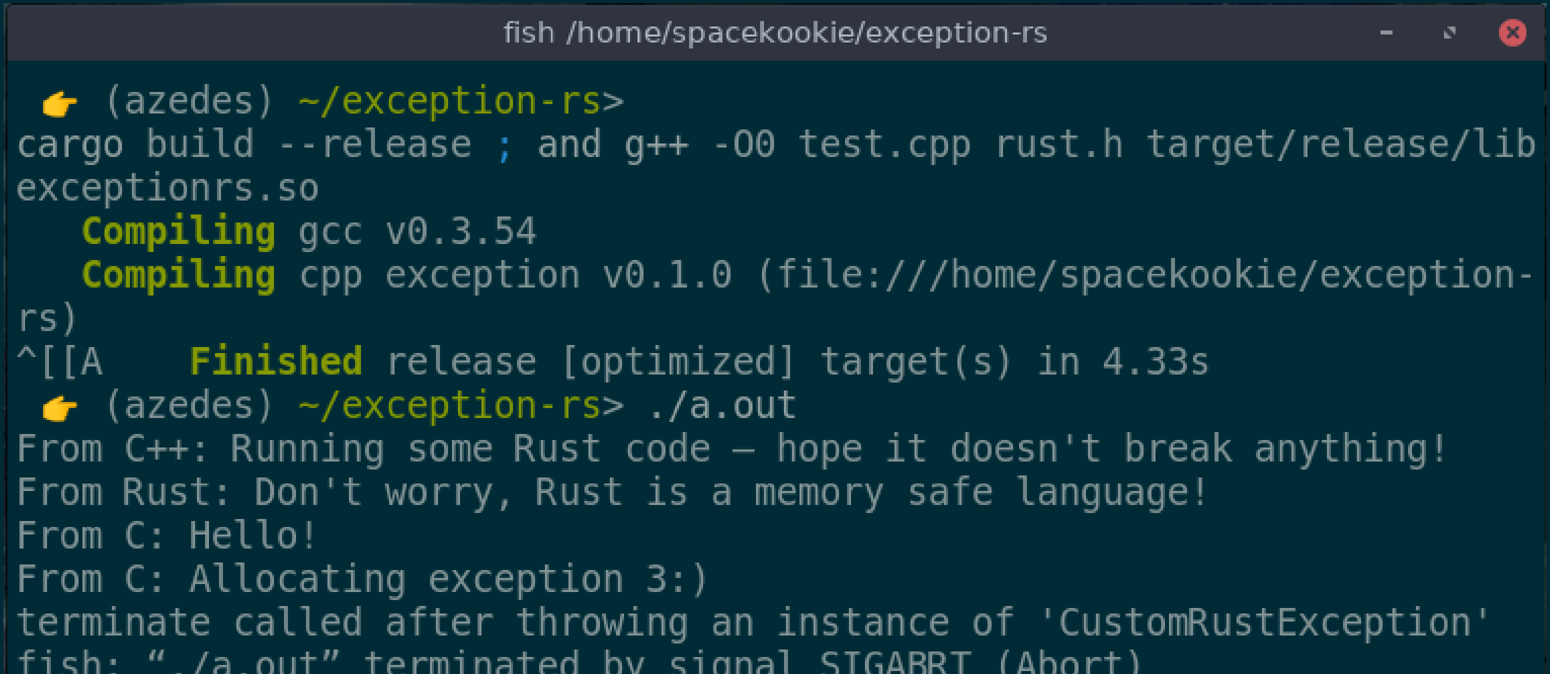
Oh god please don't use this! (soon™ on crates.io)

No `libc++` bindings in Rust

Invoke apropriate functions via C shim layer

```c
extern void *__cxa_allocate_exception(size_t thrown_size);
extern void __cxa_throw(void *e, void **t, void (*dest)(void *));
```

Functions are linked when C++ project is compiled



```
 👉 (azedes) ~/exception-rs>
cargo build --release ; and g++ -O0 test.cpp rust.h target/release/lib
exceptionrs.so
   Compiling gcc v0.3.54
   Compiling cpp exception v0.1.0 (file:///home/spacekookie/exception-
rs)
^[[A    Finished release [optimized] target(s) in 4.33s
 👉 (azedes) ~/exception-rs> ./a.out
From C++: Running some Rust code — hope it doesn't break anything!
From Rust: Don't worry, Rust is a memory safe language!
From C: Hello!
From C: Allocating exception 3:)
terminate called after throwing an instance of 'CustomRustException'
fish: “./a.out” terminated by signal SIGABRT (Abort)
```

CAN YOU *CATCH* A C++ EXCEPTION IN RUST?

Yes. But not today

THANK YOU (FOR REAL)

Follow me on twitter `@spacekookie`

Or: `kookie@spacekookie.de`

- ♡ My employer: **Ferrous Systems**
- ♡ Mozilla
- ♡ All of you