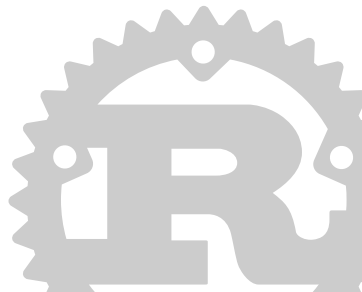


Calling Rust from C and Java

Danilo Bargaen (@dbrgn)

2017-10-31

Rust Zürichsee Meetup



```
println!(":?", Self)
```

Hi! I'm Danilo (@dbrgn).



```
println!(":?", Self)
```

Hi! I'm Danilo (@dbrgn).

I live in Rapperswil ([instagram.com/visitrapperswil](https://www.instagram.com/visitrapperswil)).



```
println!(":?", Self)
```

Hi! I'm Danilo (@dbrgn).

I live in Rapperswil ([instagram.com/visitrapperswil](https://www.instagram.com/visitrapperswil)).

I work at Threema (threema.ch).



```
println!(":?", Self)
```

Hi! I'm Danilo (@dbrgn).

I live in Rapperswil ([instagram.com/visitrapperswil](https://www.instagram.com/visitrapperswil)).

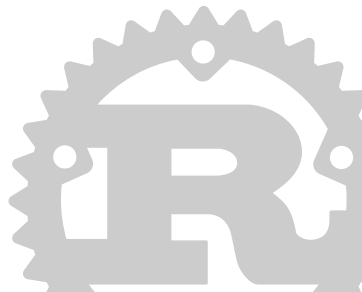
I work at Threema (threema.ch).

I'm a founding member of Coredump
hackerspace (coredump.ch).



Outline

1. FFI
2. Parsing ICE Candidates
3. Rust \rightleftharpoons C
4. Rust \rightleftharpoons Java
5. Questions?



FFI



What is FFI?

FFI stands for «Foreign Function Interface».

It's a way to call functions written in one programming language from another one.

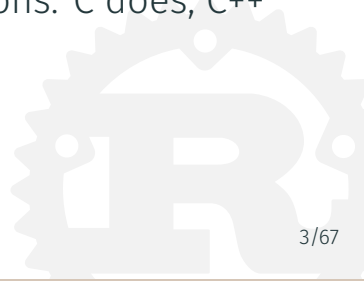


How does it work?

FFI works if there are known binary calling conventions that both sides adhere to.

Think of it as a «communication protocol».

Not all languages have fixed calling conventions. C does, C++ does not.

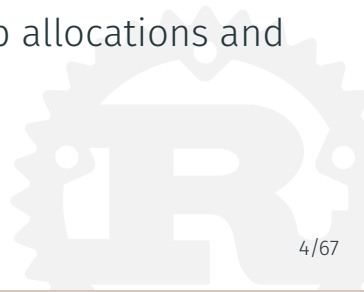


FFI Is Easy!!!...?

Most FFI examples / intros do something like adding two integers.

That is a totally useless example, since reality is much more complex.

Biggest pain point once you get started: Heap allocations and pointers.

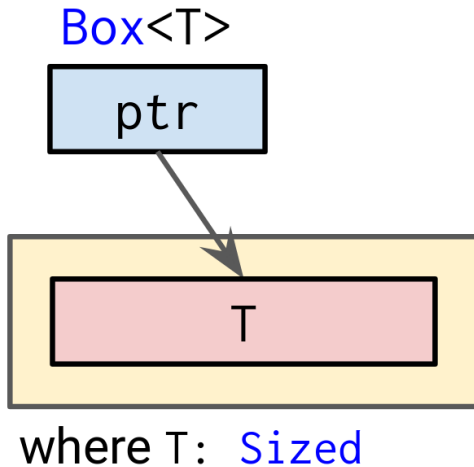


Memory Ownership

If you know Rust, you have probably acquired an intuitive understanding of the concept called «Memory Ownership». The owner of an object owns its memory.

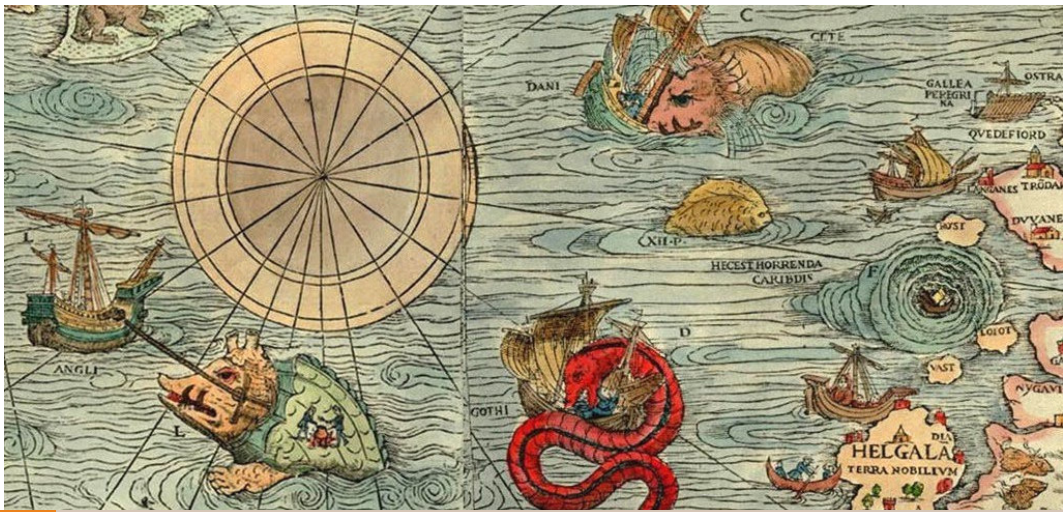


Let's Talk About Boxes



Here Be Dragons

Rust ownership guarantees only cover memory allocated by Rust. For all other memory, we cannot make any assumptions.



Rust: Beware the Drop



When returning raw (unsafe) pointers from Rust, remember that the memory owned by Rust will be freed when the corresponding value is dropped.

C: Beware Other Allocators



By default, Rust uses the jemalloc memory allocator and C does not.

When handling memory allocated by Rust, do not try to free it in a C program.

Java: Beware the GC



When holding on to a Java reference in Rust, the Java runtime must be notified about that. Otherwise the memory may be collected by the garbage collector.

It's Dangerous



doc.rust-lang.org/nomicon/ffi.html

jakegoulding.com/rust-ffi-omnibus/

valgrind.org/

Parsing ICE Candidates



ICE Candidate Parsing

In order to have a practical example in this talk, we'll take a look at a simple library I've written.

That library is a parser for ICE candidates with bindings for C and Java.

Source: <https://github.com/dbrgn/candidateparser>

WTF are ICE Candidates?



WTF are ICE Candidates?



No, not that ice.



WTF are ICE Candidates?



No, not that ice.

ICE stands for «Interactive Connectivity Establishment».

It's a protocol used in peer-to-peer networks to establish a connection.

WTF are ICE Candidates?

This is what an ICE candidate looks like:

```
candidate:842163049 1 udp 1686052607  
1.2.3.4 46154 typ srflx  
raddr 10.0.0.17 rport 46154 generation 0  
ufrag EEtu network-id 3 network-cost 10
```

Parsing

Since this talk is about FFI, I won't cover the parsing in detail. The parser is written in Rust using nom¹. It provides a single function as entry point:

```
pub fn parse(sdp: &[u8]) -> Option<IceCandidate>
```

¹<https://crates.io/crates/nom>

IceCandidate struct

This is the type returned by the parsing function:

```
pub struct IceCandidate {  
    pub foundation: String,  
    pub component_id: u32,  
    pub transport: Transport,  
    pub priority: u64,  
    pub connection_address: IpAddr,  
    pub port: u16,  
    pub candidate_type: CandidateType,  
    pub rel_addr: Option<IpAddr>,  
    pub rel_port: Option<u16>,  
    pub extensions: Option<HashMap<Vec<u8>, Vec<u8>>>,  
}
```

IceCandidate struct


This is the type returned by the parsing function:

```
pub struct IceCandidate {  
    pub foundation: String, Allocation  
    pub component_id: u32, ?  
    pub transport: Transport, ?  
    pub priority: u64, ?  
    pub connection_address: IpAddr, ?  
    pub port: u16, ?  
    pub candidate_type: CandidateType, ?  
    pub rel_addr: Option<IpAddr>,  
    pub rel_port: Option<u16>,  
    pub extensions: Option<HashMap<Vec<u8>, Vec<u8>>>, Allocation  
}
```

Enums


Inside the `IceCandidate` struct, two enums are being used.

```
pub enum CandidateType {  
    Host, Srflx, Prflx, Relay, Token(String)  
}
```



Allocation

```
pub enum Transport {  
    Udp, Extension(String)  
}
```



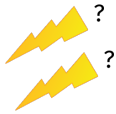
Allocation

Note that both of them contain associated data.

External Types

The `connection_address` and the `rel_addr` keys contain an `std::net::IpAddr`.

```
pub enum IpAddr {  
    V4(Ipv4Addr),  
    V6(Ipv6Addr),  
}
```



Other Complex Types

The `extensions` key type:

 Allocation  Allocation  Allocation

`Option<HashMap<Vec<u8>, Vec<u8>>>>.`

Rust \rightleftharpoons C



Rust Types in C

To be able to call Rust from C, we need to:

- Make sure that all involved data types are `#[repr(C)]` (simplifying Rust specific types)



Rust Types in C

To be able to call Rust from C, we need to:

- Make sure that all involved data types are `#[repr(C)]` (simplifying Rust specific types)
- Mark all exposed functions with `extern "C"` and `#[no_mangle]`



Rust Types in C

To be able to call Rust from C, we need to:

- Make sure that all involved data types are `#[repr(C)]` (simplifying Rust specific types)
- Mark all exposed functions with `extern "C"` and `#[no_mangle]`
- Compile the crate as a `cdylib`



Making Rust `#[repr(C)]`

If we want to be able to call Rust from C, then all involved data types need to use C representation as memory layout.

By default, the memory layout in Rust is unspecified. Rust is free to optimize and reorder fields.



IceCandidate: Rusty

```
[derive(Debug, PartialEq, Eq, Clone)]
pub struct IceCandidate {
    pub foundation: String,
    pub component_id: u32,
    pub transport: Transport,
    pub priority: u64,
    pub connection_address: IpAddr,
    pub port: u16,
    pub candidate_type: CandidateType,
    pub rel_addr: Option<IpAddr>,
    pub rel_port: Option<u16>,
    pub extensions: Option<HashMap<Vec<u8>, Vec<u8>>>,
}
```

IceCandidate: C-like

```
#[repr(C)]
pub struct IceCandidateFFI {
    pub foundation: *const c_char,
    pub component_id: u32,
    pub transport: *const c_char,
    pub priority: u64,
    pub connection_address: *const c_char,
    pub port: u16,
    pub candidate_type: *const c_char,
    pub rel_addr: *const c_char, // Optional (nullptr)
    pub rel_port: u16, // Optional (0)
    pub extensions: KeyValueTypeMap,
}
```

CStr and CString

There are two wrapper types to handle C strings:

- `std::ffi::CStr` (borrowed)
- `std::ffi::CString` (owned)

String to *const c_char

A Rust `String` can be converted to a `*const c_char` through `CString`:

```
use std::ffi::CString;
use libc::c_char;

let s: String = "Hello".to_string();
let cs: CString = CString::new(s).unwrap();
let ptr: *const c_char = cs.into_raw();
```

⚠ **Note:** `CString` enables C compatibility but should not be exposed directly through FFI!

⚠ **Note:** `CString::into_raw()` transfers memory ownership to a C caller!

(The alternative would be `CString::as_ptr()`)

Custom types to `*const c_char`

Our library generates some enums with associated data that cannot be represented directly as a C type. Return it as a C string instead!

```
pub enum Transport { Udp, Extension(String) }

impl Into<CString> for Transport {
    fn into(self) -> CString {
        match self {
            Transport::Udp => CString::new("udp").unwrap(),
            Transport::Extension(e) => CString::new(e).unwrap(),
        }
    }
}
```


Custom types to `*const c_char`

We also return some external types like `IpAddr`. We cannot impl `Into<CString>` for those due to the orphan rule².

Instead, convert them to a C string using the `ToString` trait!

```
let addr = CString::new(parsed.addr.to_string())  
    .unwrap()  
    .into_raw();
```

²You can write an impl only if either your crate defined the trait or defined one of the types the impl is for.

Optional types to C

C does not have a type directly corresponding to `Option<T>`. Instead, when dealing with heap allocated types, use (yuck!) null pointers.

```
let optional_ip = match parsed.rel_addr {  
    Some(addr) => {  
        CString::new(addr.to_string()).unwrap().into_raw()  
    },  
    None => std::ptr::null(),  
}
```

Optional types to C

C does not have a type directly corresponding to `Option<T>`. Instead, when dealing with heap allocated types, use (yuck!) null pointers.

```
let optional_ip = match parsed.rel_addr {  
    Some(addr) => {  
        CString::new(addr.to_string()).unwrap().into_raw()  
    },  
    None => std::ptr::null(),  
}
```

For simpler types, use an "empty" value.

```
let optional_port = parsed.rel_port.unwrap_or(0);
```

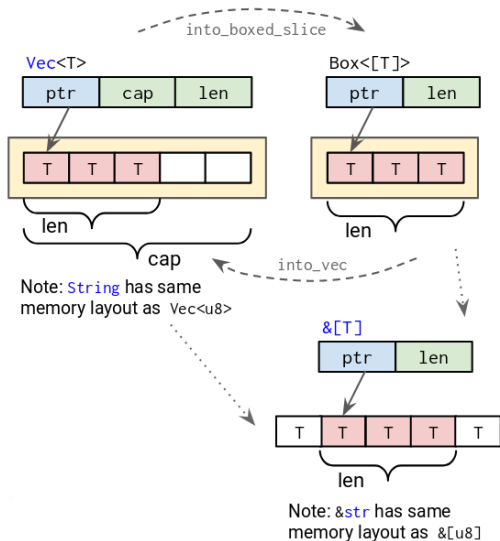
HashMap to C

Now for some more complex types. Our `extensions` field has the type `Option<HashMap<Vec<u8>, Vec<u8>>>`.



Vec to C

Let's start with `Vec<T>`.



Vec to C: Option 1

Option 1: Shrink `Vec`, get a pointer, then forget the memory.

```
let mut v: Vec<u8> = vec![1, 2, 3, 4];  
v.shrink_to_fit(); // assert_eq!(v.len(), v.capacity());  
let ptr: *const uint8_t = v.as_ptr();  
std::mem::forget(v);
```

Vec to C: Option 2

Option 2: Use `into_boxed_slice` and `into_raw`.

```
let v: Vec<u8> = vec![1, 2, 3, 4];  
let v_box: Box<[u8]> = v.into_boxed_slice();  
let ptr: *const [uint8_t] = Box::into_raw(v_box);
```

Passing Vec to C

When passing a **Vec** to C, it is passed as a pointer to the first element.

C also needs to know how long our vector is!

```
let v: Vec<u8> = vec![1, 2, 3, 4];  
let v_len: usize = v.len();  
let v_ptr: Box<u8> = Box::into_raw(v.into_boxed_slice());  
let raw_parts = (v_ptr, v_len);
```

In C:

```
for (size_t i = 0; i < rustvec.len; i++) {  
    handle_byte(rustvec.ptr[i]);  
}
```


Passing HashMap<Vec<u8>, Vec<u8>> to C

Pass a HashMap to C using a KeyValuePair type!

```
#[repr(C)]
pub struct KeyValueMap {
    pub values: *const KeyValuePair,
    pub len: size_t,
}
```

```
#[repr(C)]
pub struct KeyValuePair {
    pub key: *const uint8_t,
    pub key_len: size_t,
    pub val: *const uint8_t,
    pub val_len: size_t,
}
```

The Parsing Function

Phew! That was quite a lot. Now how do we actually expose this to C?

...using an `extern "C"` function.

```
#[no_mangle]
pub unsafe extern "C" fn parse_ice_candidate_sdp(
    sdp: *const c_char
) -> *const IceCandidateFFI {
    // ...
}
```

The Parsing Function: Reading C strings

Inside that function, we first need to convert the C char pointer to a Rust byte slice.

```
// `sdp` is a *const c_char
if sdp.is_null() {
    return std::ptr::null();
}
let cstr_sdp = CStr::from_ptr(sdp);
```

Note that we're using **CStr**, not **CString**!

The Parsing Function: Reading C strings

Next, we parse the ICE candidate bytes using the regular Rust parsing function.

```
// Parse
let bytes = cstr_sdp.to_bytes();
let parsed: IceCandidate =
    match candidateparser::parse(bytes) {
        Some(candidate) => candidate,
        None => return ptr::null(),
    };

```

The Parsing Function: Reading C strings

Finally we convert the Rust type to the FFI type (using the techniques explained previously) and return a pointer to that.

```
// Convert to FFI representation
let ffi_candidate: IceCandidateFFI = ...;

// Return a pointer
Box::into_raw(Box::new(ffi_candidate))
```

Compiling as a C Library

To compile the Rust crate as a C compatible shared library, put this in your `Cargo.toml`:

```
[lib]
name = "candidateparser_ffi"
crate-type = ["cdylib"]
```

This will result in a `candidateparser_ffi.so` file.

Generating a Header File

To be able to use the library from C, you also need a header file.

You can write such a header file by hand, or you can generate it at compile time using the **cbindgen** crate³.

³<https://github.com/eqrion/cbindgen>

Calling the Parser from C

Include the header file and simply call the function:

```
#include "candidateparser.h"
const IceCandidateFFI *candidate =
    parse_ice_candidate_sdp(sdp);
```

Then link against the shared library when compiling:

```
$ clang example.c -o example \
    -L ../target/debug -l candidateparser_ffi \
    -Wall -Wextra -g
```

A full example is available in the `candidateparser-ffi` crate on Github.

Cleaning up

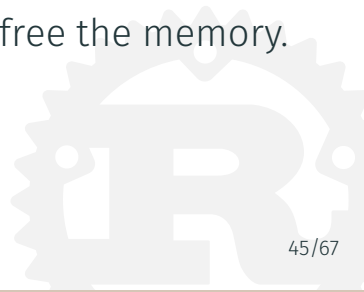


Cleaning up

Since we passed pointers from Rust to C, that memory cannot be freed by C!

If we don't free it, we end up with memory leaks.

We need to pass the pointers back to Rust to free the memory.



Cleaning up

First, create another function that accepts a pointer to an `IceCandidateFFI` struct.

```
#[no_mangle]
pub unsafe extern "C" fn free_ice_candidate(
    ptr: *const IceCandidateFFI
) {
    if ptr.is_null() { return; }
    // ...
}
```

Cleaning up

Now we create an owned **Box** from the pointer.

```
// Cast `*const T` to `*mut T`  
let ptr: ptr as *mut IceCandidateFFI;  
  
// Reconstruct box  
let candidate: Box<IceCandidateFFI> = Box::from_raw(ptr);
```

Cleaning up Strings

Because the struct also contains pointers, we reconstruct Rust owned types from these pointers. The memory is freed as soon as those objects go out of scope!

For strings:

```
CString::from_raw(candidate.foundation as *mut c_char);
```

For nullable strings:

```
if !candidate.rel_addr.is_null() {  
    CString::from_raw(candidate.rel_addr as *mut c_char);  
}
```

Cleaning up Vec / KeyValueType

Reclaiming the memory for our KeyValueType is a bit more complex:

```
let e = candidate.extensions;
let pairs = Vec::from_raw_parts(e.values as *mut KeyValuePair,
                                e.len as usize, e.len as usize);

for p in pairs {
    Vec::from_raw_parts(p.key as *mut uint8_t, // Start
                        p.key_len as usize,    // Length
                        p.key_len as usize);   // Capacity
    Vec::from_raw_parts(p.val as *mut uint8_t, // Start
                        p.val_len as usize,    // Length
                        p.val_len as usize);   // Capacity
}
```

We Did It!

Whew, that was a bumpy ride!



Rust \rightleftharpoons Java

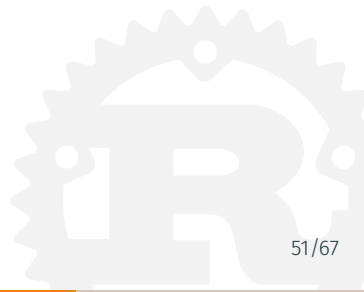


Hello Java

Ok, now for Java.

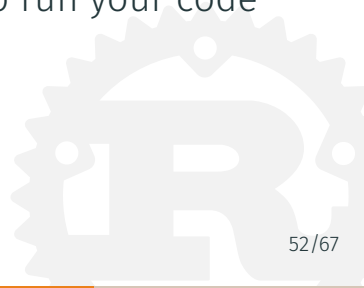
Unfortunately we can't reuse the code we wrote for C.

But we can reuse some of the concepts!



The "classic" way to talk to Java from external languages is through JNI (Java Native Interface).

There are newer options by now (namely JNA), but as far as I know there are issues with that if you want to run your code on Android.



Preparations

First, we have to write classes for all Java types we're going to use. Since it's Java, it's a bit verbose.

```
package ch.dbrgn.candidateparser;
import java.util.HashMap;

public class IceCandidate {
    // Non-null fields
    private String foundation;
    private long componentId;
    private String transport;
    private long priority;
    private String connectionAddress;
    private int port;
    private String candidateType;
```

Preparations

```
// Extensions
private HashMap<String, String> extensions = new HashMap<>();

// Nullable fields
private String relAddr = null;
private Integer relPort = null;

public IceCandidate(String foundation, long componentId,
                    String transport, long priority,
                    String connectionAddress, int port,
                    String candidateType) {
    this.foundation = foundation;
    this.componentId = componentId;
    // ...
}

// ...
```

Preparations

Next, we'll write the "interface" for the parser class.

```
package ch.dbrgn.candidateparser;

public class CandidateParser {
    static {
        System.loadLibrary("candidateparser_jni");
    }

    public static native IceCandidate parseSdp(String sdp);
}
```

Note the `native` modifier.

Generating JNI Headers

To generate the JNI headers, we first compile the `.java` files:

```
$ javac -classpath app/src/main/java/ \
    app/src/main/java/ch/dbrgn/candidateparser/IceCandidate.java
$ javac -classpath app/src/main/java/ \
    app/src/main/java/ch/dbrgn/candidateparser/CandidateParser.java
```

Then use the `javah` tool to generate the headerfile.

```
$ javah -classpath app/src/main/java/ \
    -o CandidateParserJNI.h \
    ch.dbrgn.candidateparser.CandidateParser
```

Generating JNI Headers

The header file (minus some boilerplate):

```
#include <jni.h>
```

```
/*  
 * Class:      ch_dbrgn_candidateparser_CandidateParser  
 * Method:     parseSdp  
 * Signature:  (Ljava/lang/String;)Lch/dbrgn/candidateparser  
 *                                     /IceCandidate;  
 */  
JNIEXPORT jobject JNICALL  
    Java_ch_dbrgn_candidateparser_CandidateParser_parseSdp  
    (JNIEnv *, jclass, jstring);
```

Rust Bindings for JNI

Create a new library and add the `jni`⁴ crate as dependency.

```
[dependencies]
```

```
jni = "0.6"
```

```
[lib]
```

```
crate_type = ["dylib"]
```

⁴<https://github.com/prevoty/jni-rs>

lib.rs

In `lib.rs`, create a function with the same name as the function in the JNI header.

```
#[no_mangle]
#[allow(non_snake_case)]
pub extern "system"
fn Java_ch_dbrgn_candidateparser_CandidateParser_parseSdp(
    env: JNIEnv,
    _class: JClass,
    input: JString)
-> jobject {

    // ...

}
```

Converting parameters

To get a reference to a Java String passed in as an argument we need to access it through the **JNIEnv** instance and convert it to a Rust **String**.

```
let sdp: String = env.get_string(input).unwrap().into();
```

Now we can simply pass it to the regular Rust function!

```
let candidate = match candidateparser::parse(sdp.as_bytes()) {  
    Some(cand) => cand,  
    None => return std::ptr::null_mut() as *mut _jobject, // hack  
};
```

Creating New Java Objects

Since we want to return the parsed candidate to Java, we want to instantiate the Java `IceCandidate` class.

```
let obj: JObject = env.new_object(  
  // Classpath  
  "ch/dbrgn/candidateparser/IceCandidate",  
  // Signature  
  "(Ljava/lang/String;JLjava/lang/String;J  
    Ljava/lang/String;ILjava/lang/String;)V",  
  // Argument slice containing `JValue`s  
  &args  
) .unwrap();
```

JNI signature syntax:

<https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/types.html>

Creating New Java Objects

The arguments need to be wrapped in JNI wrapper types. This makes sure that the JVM GC knows about them (memory ownership!). Two examples:

```
let component_id = JValue::Long(  
    candidate.component_id as jlong  
);
```

```
let foundation = JValue::Object(  
    env.new_string(&candidate.foundation).unwrap().into()  
);
```

Inspecting Classfiles

Hint: You can use `javap` to find the signature descriptor for a method.

```
$ javap -s -classpath app/src/main/java \
    ch.dbrgn.candidateparser.IceCandidate
Compiled from "IceCandidate.java"
public class ch.dbrgn.candidateparser.IceCandidate {
    public ch.dbrgn.candidateparser.IceCandidate();
        descriptor: ()V

    public ch.dbrgn.candidateparser.IceCandidate(java.lang.String, long,
        descriptor: (Ljava/lang/String;JLjava/lang/String;JLjava/lang/String;

    public java.lang.String getFoundation();
        descriptor: ()Ljava/lang/String;
    ...
```

Calling Java Methods

You can also call methods on Java objects through the `JNIEnv`:

```
let call_result = env.call_method(  
  // Object containing the method  
  obj,  
  // Method name  
  "setRelPort",  
  // Method signature  
  "(I)V",  
  // Arguments  
  &[JValue::Int(port as i32)]  
);
```

Memory Ownership

Since all allocated memory is created through the `JNIEnv`, the original Rust memory can be freed (on drop) and the Java memory is tracked by the GC.

We don't need an explicit `free_ice_candidate` function.

Questions?



Appendix: Android Logging

You can log directly to the Android adb log through standard Rust logging facilities:

Cargo.toml:

```
[dependencies]
log = "0.3"
android_logger = "0.3"
```

Appendix: Android Logging

You can log directly to the Android adb log through standard Rust logging facilities:

lib.rs:

```
#[macro_use]
extern crate log;
#[cfg(target_os = "android")]
extern crate android_logger;

// ...

#[cfg(target_os = "android")]
android_logger::init_once(log::LogLevel::Info);
```

Thank you!

`www.coredump.ch`

Slides: `github.com/rust-zurichsee/meetups/`

Candidateparser library: `github.com/dbrgn/candidateparser/`

