17.4.2024

FFI

Programovanie v jazyku Rust

STU FIIT

Dušan Podmanický

Contents

Introduction

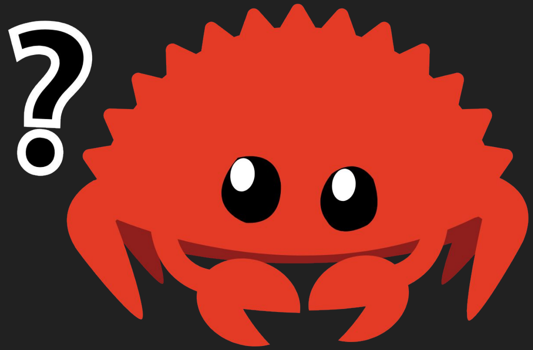
What is FFI?

- **Foreign Function Interface**
- Allows us to call functions from foreign sources
- In the context of Rust, “foreign” implies any language that is not Rust
- As a systems programming language, Rust mostly wants to interact with other low-level languages
- In this domain, C is the de-facto lingua franca

But…why?

- The ability to leverage existing code written in other languages
- You are, essentially, gaining an **entire ecosystem of libraries**
- This saves time and resources

- You may leverage capabilities that other languages possess
  - performance,
  - hardware access (GPU),
  - libraries …



Unsafe Rust

Unsafe Rust

- The compiler determines whether or not code upholds Rust guarantees
- It's **safer to reject valid programs** than to accept invalid programs
- Compiler cannot determine whether the foreign code you called is safe :(

…therefore, all FFI calls are inherently unsafe in Rust

- Unsafe superpowers:
  - **Dereference a raw pointer**
  - **Call an unsafe function or method**
  - Access or modify a mutable static variable
  - Implement an unsafe trait
  - Access fields of unions

```
unsafe {
      ...
}
```

Unsafe Rust

“unsafe” block does not mean we are doing everything ourselves:

- It does not turn off the borrow checker
- It does not turn off any other of Rust’s safety checks

If you use a reference in unsafe code, it will still be checked. The unsafe keyword only gives you access to the five features that are then not checked by the compiler for memory safety

By requiring unsafe operations to be inside annotated blocks **you’ll know where to look for any errors related to memory safety.**

Approaches to FFI

Approaches to FFI

- Some languages may leverage common runtimes:
  - JRE: Java, Scala, Kotlin
  - BEAM VM: Erlang, Elixir
- Other approaches:
  - Nim can compile into C directly
  - Zig natively supports importing of C libraries
  - D maintains their own C compiler, can be linked with c obj files and dlls, but there is overhead

In the world of systems programming, C is the lingua franca

Rust and C

Rust and C

- Rust cannot just “import” C code

Why?

- **Idiomatic C** and **idiomatic Rust** are different
- C **cannot provide the guarantees** Rust can
- Maintaining half a C compiler is **not fun**

Rust is only really valuable, if its **compiler runs on your code**.
If it were just a thin layer of unsafe wrappers around C code, the guarantees given by the compiler would be much less interesting.

Looser coupling

- We need to make C and Rust **talk to each other**
- We must go a **level lower**, and make them speak similar in **assembly**
- The **linker** will then stitch everything together
- Rust is really the one that needs to adapt
- We need to make Rust generate assembly similar to what C generates

This makes the coupling prone to errors:

- If there's a typo in your function name, you will get a nasty, unhelpful linker error
- If you use different types by accident, I am so so sorry

Disagreements

- Like any couple, C and Rust disagree

Points of contention:

- **Calling conventions**
- **Name mangling**
- **Memory layout**

Solutions:

- extern “C”
- #[no_mangle]
- #[repr(C/transparent)]

Forcing Rust into C ABI

What is ABI

- **Application Binary Interface**
- A set of conventions, it dictates how various components of computer programs **interact at the binary level**
- This can also extend to how programs interact with the OS at the binary level

- This includes:
    - **Data Type Sizes and Layouts**
    - **Function Calling Conventions**
    - Register Usage
    - System Call Interfaces
    - **Name Mangling**
    - **Exception Handling**
    - Object File Formats

Calling conventions

Calling conventions

- Rust and C disagree on function calling
- Rust generally uses more registers for argument passing

What do we mean by that?

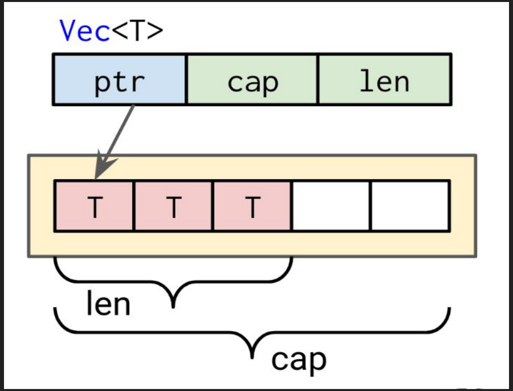
- Suppose you pass a vec to a function in Rust
    - What is a vec, really?
- In assembly, there are no vecs, just registers

What does it mean to pass a vec into a function

- A vec is essentially just a block of memory
- What we need to know is:
  a. Where does this memory start
  b. How big the block is
  c. How much of it is still free
- We can represent this with 3 pointer-sized values
- We can either:
  a. **Pass** these **3 values separately** as arguments (this is called **fastcall**)
  b. Use indirection - **pass a pointer** and let the machine figure it out (this is called **cdecl**)

  (There are other ways but they are unimportant now)

- C and Rust use different ways



Calling conventions

```
fn foo(vec: Vec<u8>) -> usize {
      vec.len()
}
```

What we’ve written

```
                    1              2              3
fn foo(ptr: *const u8, len: usize, cap: usize) -> usize {
     len
}
```

How Rust describes it

```
 fn foo(vec: *const (usize, usize, usize)) -> usize {
      vec.1
 }
```

How C describes it

Extern "C"

- Allows Rust programs to declare **C functions**
- Allows us to tell Rust to **use the C calling convention** on functions within this block
- Fast calling convention vs. cdecl calling convention

```
extern "C" {
    do_stuff_with_long(input: i64);
}
```

There will exist a function, with this name, and this type.
I haven't defined it yet, but it will exist in the final binary.
Trust me bro.

Name mangling

Name mangling

- Technique used to **resolve naming conflicts**, and **allow function overloading**
- Name mangling modifies the names of functions/variables/classes to make them **unique** across the whole program
- Rust uses name mangling, C **does not**
- This means that the actual function name is a **mangled** version of the name we assigned
- Declaring a function **#[no_mangle]** disables this feature in Rust

Cpp: example of name mangling

```
namespace A {
    void function(int x);
}
```

Mangled name for A::function(int)

_ZN1A8functionEi

```
namespace B {
    void function(int x);
}
```

Mangled name for B::function(int)

_ZN1B8functionEi

Rust: solution

```
#[no_mangle]
fn function(data: u8) {
    ...
}
```

Memory layout

What is memory layout

```c
typedef struct {
    char id;
    int moneys;
    short num_family_members;
} Container;

int main(int arc, char *argv[]) {
    printf("Size of struct in memory: %d\n
            Size of int: %d\n
            Size of char: %d\n
            Size of short: %d\n",
            sizeof(Container), sizeof(int),
            sizeof(char), sizeof(short));
    client.id = 5;
    client.num_family_members = 3;
    client.moneys = 1000;
    *((int*)&client.id) = 1000;
    return;
}
```

```
Size of struct in memory: 12
Size of int: 4
Size of char: 1
Size of short: 2
```

Assume that &client == 0x0113FD18, and all fields are set to 0:

0x0113FD18 00 cc cc cc 00 00 00 00 00 00 cc cc

After "`client.id = 5;`":

0x0113FD18 05 cc cc cc 00 00 00 00 00 00 cc cc

After "`client.num_family_members = 3;`":

0x0113FD18 05 cc cc cc 00 00 00 00 03 00 cc cc

After "`client.moneys = 1000;`":

0x0113FD18 05 cc cc cc e8 03 00 00 03 00 cc cc

After "`*((int*)&client.id) = 1000;`":

0x0113FD18 e8 03 00 00 e8 03 00 00 03 00 cc cc

Memory layout tells us about the structure of our memory and how things are kept inside it.

Memory layout

“The order in which struct members are allocated in memory is the same as the order in which they are declared”:

- Rust **does not** guarantee this
- C **does** guarantee this

Padding between fields in struct:

0x0113FD18 00 cc cc cc 00 00 00 00 00 00 cc cc

Differences between types:

- C types != Rust types (in general)
- C types != C types → architecture/compiler/OS dependent

Some types are the same

- **Integers**
  are the same, if we use proper size
- **Booleans**
- **Enums**
  basic Rust enums, with no payloads,
  represented as just numbers
  the same for Rust and C
- **Unions**

```
extern "C" {
    // integers
    fn is_even(x: i32) -> bool;

    // pointers
    fn is_null(ptr: *const u32) -> bool;
}

#[repr(u8)]
enum Color { R, G, B }

extern "C" {
    // tag-only enums
    fn circle_with_me(c: Color) -> Color;
}

#[repr(C)]
union U { int: i64, float: f64 }
```

Some just need a bit of convincing

- **#[repr(C)]**
  forces Rust to use the same memory representation as C would

```
#[repr(C)]
struct Point { x: f32, y: f32 }

extern "C" {
    // repr(C) structs
    fn h(p: Point) -> bool;
}
```

- **#[repr(transparent)]**
  forces Rust to use the same memory representation as the underlying type

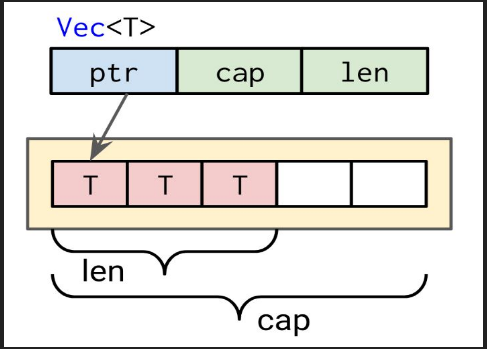
```
#[repr(transparent)]
struct Wrapper<T>(T);

extern "C" {
    // repr(transparent) structs,
    // if the inner type is repr(C)
    fn h(w: Wrapper<u64>) -> bool;
}
```

Some are just haters

- Enums like Result/Option
- Custom enums with payloads
- Owned collections like String, Vec<T>
- Fat pointers like &str, &[T]

- They require special, manual treatment
- We need to **destructure them**



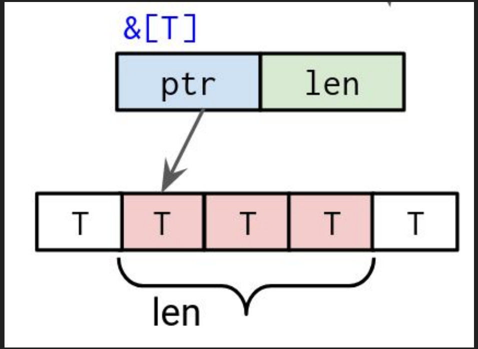
```
// Destructuring Vec
fn handle_vec(vector: Vec<i32>) {
    let ptr = vector.as_ptr();
    let cap = vector.capacity();
    let len = vector.len();

    c_vec_function(ptr, cap, len);
}
```

Some are just haters

- Enums like Result/Option
- Custom enums with payloads
- Owned collections like String, Vec<T>
- Fat pointers like &str, &[T]

- They require special, manual treatment
- We need to **destructure them**



&str has same memory layout as &[u8]

```
// Destructuring fat pointers like &str
fn handle_fptr(s: &str) {
   let ptr = s.as_ptr();
   let len = s.len();

   c_fptr_function(ptr as *const u8, len);
}
```

Summary so far

- C and Rust are both great
- They just don't speak the same language by default
- We need to create ways for them to come together and speak the same language in assembly
- These ways include:
  - Tell Rust the names and signatures of foreign C functions using `**extern "C"**`
  - Force Rust into the C calling convention using `**extern "C"**`
  - Force Rust to cease its name mangling ways using **#[no_mangle]**
  - Use only compatible types using either basic integers or **#[repr(C)]/#[repr(transparent)]**
- We need to create bindings for Rust functions in C and C functions in Rust

Cargo bindgen automates some of this process.

Fun facts - which of these is actually defined in C?

- The width of a byte
- The width of an integer
- Signed integer overflow
- The order of function parameter evaluation (f(x(), y(), z()))
- Subscripting with const char* : 2[“ABCDE”]

What is the problem?

```
void do_stuff_with_long(long input) {
    // work work
}
```

```
extern "C" {
    fn do_stuff_with_long(input: i64);
}
```

What is the problem?

```
void do_stuff_with_long(long input) {
    // work work
}


extern "C" {
    fn do_stuff_with_long(input: i64);
}
```

Solution:
Depends on whether
“long” is 64bit or 32bit.

What are libraries

Libraries

What is a library

- collection of pre-compiled routines, classes, or functions

Static libraries (.lib, .a)

- Code is stored in one or more archive files at development time
- Their contents are physically copied into the app's executable file by the linker
- This results in a larger executable file
- Executable contains all necessary functionality without external dependencies, improving portability

Dynamic libraries (.dll, .dylib, .so)

- Code is stored in separate files that are not part of the application
- Linked at run-time, only references to the library are included in the executable
- Smaller executables,
  multiple programs may share a library
- Updates to the library are possible without recompilation of the applications using it

Generating bindings

Generating proper bindings in FFI is crucial

How do we generate bindings?

- By hand (good luck)
- Tools that allow us to automate this process:
    - **bindgen - generate Rust FFI bindings to C libraries**
    - **cbindgen - generate C header (.h) files from rust code**

Binding generators - bindgen and cbindgen

Upsides:

- Doing god’s work
- No human error element

Downsides:

- The generated bindings are absolutely putrid
- We need to create nice wrappers around these
- If we don’t, we will get yelled at in the PR and nobody will want to work with us

It is our responsibility as good programmers to create nice wrappers around them.

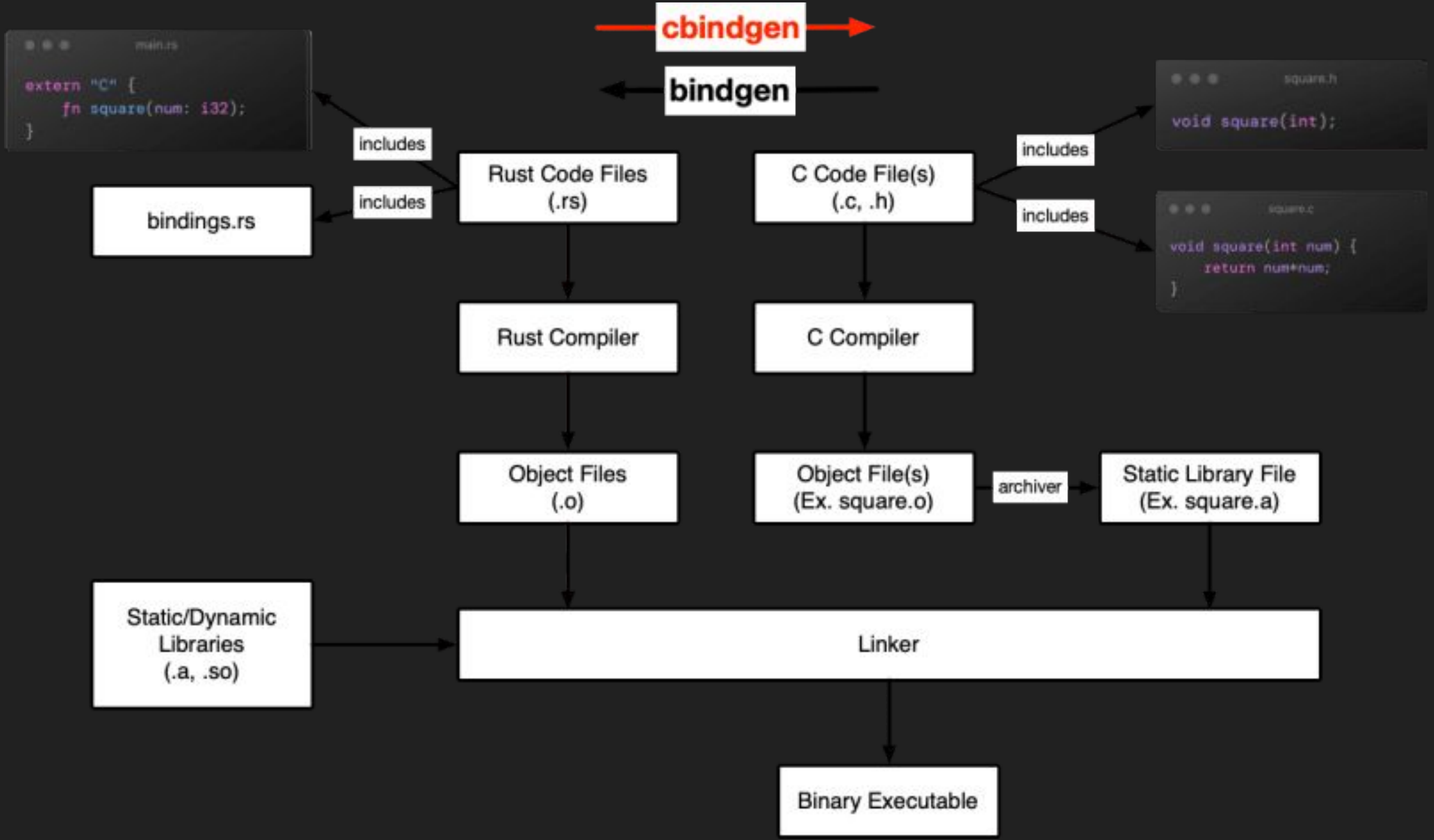
Examples of generated bindings

```
extern "C" {
    pub fn crypto_stream_salsa20_tweet_xor(
        arg1: *mut ::std::os::raw::c_uchar,
        arg2: *const ::std::os::raw::c_uchar,
        arg3: ::std::os::raw::c_ulonglong,
        arg4: *const ::std::os::raw::c_uchar,
        arg5: *const ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
```

You should react to this with some level of disgust

```
extern "C" {
    pub fn crypto_verify_16_tweet(
        arg1: *const ::std::os::raw::c_uchar,
        arg2: *const ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn crypto_verify_32_tweet(
        arg1: *const ::std::os::raw::c_uchar,
        arg2: *const ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
```



```rust
extern "C" {
    fn square(num: i32);
}
```

```c
void square(int);
```

```c
void square(int num) {
    return num*num;
}
```

Interacting with Rust from C

Interacting with Rust from C

We need to compile our Rust code as a library

- Compiling as a dynamic library by modifying .toml file:
  - .so on linux
  - .dll on windows
  - .dylib on apple

```toml
[lib]
crate-type = ["cdylib"]
```

- Compiling as a static library by modifying .toml file:
  - .a file on linux and apple
  - .lib file on windows

```toml
[lib]
crate-type = ["staticlib"]
```

You can also compile as a dynamic library with just “**dylib**”. This is intended for use with other Rust code, while “**cdylib**” is intended for interoperability with C ABI. However, if you did everything else correctly, just “**dylib**” **SHOULD** still suffice.

Interacting with C from Rust

Interacting with C from Rust

We need to compile our C code as a library

- We can use the “**cc**” crate which will do this for us (as a static lib)
- We can tell the builder where to find already compiled libraries and link them

build.rs

```rust
extern crate cc;

fn main() {
    println!("cargo:rerun-if-changed=crc32.h");
    println!("cargo:rerun-if-changed=crc32.c");
    cc::Build::new()
        .file("crc32.c")
        .compile("crc32");
}
```

build.rs

```rust
fn main() {
   println!("cargo:rustc-link-lib=static=mylibrary"); // for static libraries
   println!("cargo:rustc-link-lib=dylib=mylibrary");  // for dynamic libraries
   println!("cargo:rustc-link-search=native=/path/to/libraries");
}
```

Pitfalls, undefined behaviour

Rust memory safety

- Rust utilises memory ownership
- In Rust, the compiler ensures references are always valid

What if we use raw pointers?

- Rust will allow us to return a pointer to a variable that will be deallocated at the end of the function
- This make is very easy to return a dangling pointer when using Rust functions in other languages

Wrong

```rust
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

Correct

```rust
fn no_dangle() -> String {
    let s = String::from("hello");
    s
}
```

Panics are undefined behaviour

- If we call a Rust function from C code, and the Rust code panics, it is undefined behaviour
- If we call a C function from Rust, and the C code encounters an error, it is undefined behaviour

If we encounter undefined behaviour, the best case scenario is that the program crashes.

Miscellaneous

- W65 (and others) family of architectures - usize is not the same as size_t
- Size of long is different between linux/windows/mac
- Null-termination in strings
- Thread safety
- Passing memory allocated in one language and freeing it in other may also lead to undefined behaviour

The error messages, if they are even present, are often unhelpful when using FFI. We therefore have to be extra careful with everything we do.

End