Rust FFI

Interacting with other languages using the Foreign Function Interface

https://github.com/distil/rust_meetup_ffi

Motivation

- Legacy code written in another language.
- Interface with existing libraries.
- Standard APIs like **WinAPI**, **OpenGL**, etc.

Concerns

- Code duplication
  - Diverging duplicated code might cause undefined behavior!
  - Type marshalling
- Explicit memory management
- `unsafe`
- Calling conventions: `cdecl`, `stdcall`, `fastcall`

Rust → C

C

```c
#include <stdint.h>

int32_t squared(int32_t x) {
    return x * x;
}
```

Rust

```rust
#[link(name = "calling_c")]
extern {
    fn squared(x: i32) -> i32;
}

fn main() {
    let squared = unsafe {
        squared(5)
    };

    println!("5 * 5 = {}", squared);
}
```

C → Rust

C

```c
#include <stdint.h>
#include <stdio.h>

int32_t squared(int32_t x);

int main(int argc, char **argv) {
    printf("5*5 = %d\n",
squared(5));
    return 0;
}
```

Rust

```rust
#[no_mangle]
pub extern fn squared(x: i32) -> i32 {
    x * x
}
```

Memory management

C

```c
#include <stdint.h>
#include <string.h>

int32_t string_length(const char *str)
{
    return strlen(string);
}
```

Rust

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

#[link(name = "memory_management")]
extern {
    fn string_length(
        string: *const c_char) -> i32;
}

fn main() {
    let c_string = CString::new(
        "Hello, world!").unwrap();

    let length = unsafe {
       string_length(c_string.as_ptr())
    };
    println!("length: {}", length);
}
```

Ownership

C

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

char *stringify(int32_t x);
void release_string(char *string);

int main(int argc, char **argv) {
    char *string = stringify(42);
    printf("stringify: %s\n", string);

    release_string(string);
    return 0;
}
```

Rust

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

#[no_mangle]
pub unsafe extern fn stringify(
        x: i32) -> *mut c_char {
    CString::new(format!(
        "x is equal to {}", x))
        .unwrap().into_raw()
}

#[no_mangle]
pub unsafe extern fn release_string(
        string: *mut c_char) {
    CString::from_raw(string);
}
```

Pitfalls

- `panic!()`
  - Use `std::panic::catch_unwind()`
- Memory allocators
  - `malloc`, `free` for **C**
  - **`new`**, **`delete`** for **C++**
- Calling conventions
  - **`cdecl`**, **`stdcall`**, **`fastcall`**

Asynchronous callbacks (I)

Rust

```rust
fn perform_task<F: Fn(i32)+'static>(
    value: i32,
    f: F
);

fn main() {
    perform_task(
        5,
        |result| { /* ... */ }
    );
}
```

C++

```cpp
typedef void (*callback_t)(void *,int32_t);

extern "C" void c_perform_task(
    int32_t value,
    callback_t callback,
    void *userdata
) {
    thread([=]() {
        this_thread::sleep_for(seconds( 3))
;
        callback(userdata, value * value);
    }).join();
}
```

Asynchronous callbacks (II)

```
fn perform_task<F: Fn(i32)+'static>(
        value: i32, f: F) {
    let userdata = Box::into_raw(Box::new(
        Userdata{ callback: Box::new(f)}
    )) as *mut c_void;

    unsafe { c_perform_task(
        value, callback, userdata)
    }
}
```

```
struct Userdata {
    callback: Box<Fn(i32)>,
}

unsafe extern fn callback(
        userdata: *mut c_void,
        result: i32) {
    (Box::from_raw(
        userdata as *mut Userdata
    ).callback)(result);
}
```

LuaJIT FFI

```
local ffi = require('ffi')
ffi.cdef[[
    int32_t squared(int32_t x);
]]

local sdk = ffi.load('my_rust_library')

sdk.squared(5)
```

Summary

- Lots of room for typos and other mistakes
- Powerful tool when used correctly
- Can be extended to work with almost any language