From C to Rust and Back

Luca Barbato

RustLab 2019

Intro - Who am I?

- Luca Barbato
  - lu_zero@gentoo.org
  - lu_zero@videolan.org
  - @lu-zero on GitHub
- Packaging software in Gentoo let you experience about every possible pitfalls and shortcomings in languages **and** their build systems.
- Writing complex mutimedia code is always interesting, you have some heavy constraints since you need performance **AND** keep the code robust at the same time.

Picking a language for a project

Rules of thumb

- How much time I'm going to spend in tasks **unrelated** to implementing features?
- How much code already written can I **reuse** if I use this language for the new project?
- How hard will be for 3rd-parties to use my projects?

Picking a language for a project

New or mature?

- A **new** language could solve some well known problems in innovative ways
- A **mature** language usually has a good number of rough edges smoothed out with time
- One or another might provide better tools to solve your problems.

From C to Rust

- **C** is a **mature** language with plenty of high performing and battle tested software available.
- **Rust** is a relatively **new** language that is touted to provide speed and safety at the same time.
- Why moving from **C** to **Rust**?

From C to Rust

C - Pros

- **C** is an established language, plenty of amazing software everybody uses is written in it.
- It let you have **nearly** predictable performance while giving you enough levels of abstractions to express concepts in an effective way, most of the times.
- It's *ABI* is simple enough that nearly every language can map to it.

From C to Rust

C - Cons

- The language, and its compilers, have nearly nothing to prevent you to make **mistakes** that lead to memory **corruption**
  - Even if you are careful the odds are **always** non-zero
- You **pay** for the abstractions
  - The boilerplate code you have to write is large
    - *preprocessor* macros can **hide** some, and even more effectively **hide bugs** within it!
  - The compiler usually cannot optimize away all of it.

From C to Rust

Rust - Pros

- **Rust** actively prevents you from make a **large** class of mistakes.
    - You cannot have memory hazards in safe rust: the compiler will **stop you**.
- In **Rust** higher level abstractions usually can lead to better runtime execution
    - If the compiler has **better information** on how the code should behave it could apply optimizations it cannot consider otherwise, e.g. the *autovectorizer works much better!*
- A growing number of high performance libraries is being produced, mainly thanks to the fact rust let you write **robust code** that is also fast to **execute**.

From C to Rust

Rust - Cons

- **Rust** is a relatively young language
  - The **ABI** is not set in stone
  - You have some good software written with it, but not ALL you need
- You **could** use it everywhere, but that does not mean you **should** rewrite everything with it.
  - Rust does **not** save you from logic mistakes
  - There is always a cost-opportunity tradeoff

From C to Rust

Ideally you'd like to use the best of both words:

- Use the **rust** robustness and speed to write complex code that would be otherwise **painful** to debug.
- Leverage battle-tested C (or **assembly**) routines that had been already optimized and known to work correctly.

From C to Rust (and Back)

Integration options

- Replace a small internal component from a large C project (e.g. librsvg)
- Share the **assembly-optimized** kernels across projects (e.g. ring or rav1e)
- Use a rust library from your C/Go production pipeline (crav1e at Vimeo
    - **BONUS TRACK**: Use rust to write your system **libc** since your whole operating system is written in Rust already.

From C to Rust: Concerns and hurdles

Replace small internal components

- The hard parts are easy:
  - ABI compatibility
  - Object code generation
- The devil is in the ~~detail~~**integration**

Writing C-compatible code in Rust

Language features

- `#[repr(c)]` for our data types
  - The default **Rust** memory representation is highly optimized.
  - You can tell the compiler to be wasteful and have structs
- `extern "C"` & `#[no_mangle]` for our functions
  - **Rust** a specific symbol mangling strategy to avoid collisions.
  - You can tell the compiler to not do that (and be more precise on what to do when the need arises).
- Use the std::os::raw and std::ffi type definitions (and the libc crate when needed)

Writing C-compatible code in Rust

Compiler features

- Use `--crate-type staticlib` to ask `rustc` to produce a normal archive.
- Use `--print native-static-libs` to ask `rustc` what are the system libraries that should be linked to (if any is needed).

Writing C-compatible code in Rust

Example 1 - hello rust

```c
// main.c
int hello_rust(void);

int main(void) {
    return hello_rust();
}
```

```rust
// lib.rs
use std::os::raw::*;

#[no_mangle]
extern "C" fn hello_rust() -> c_int {
    println!("Hello from Rust!");
    0
}
```

Writing C-compatible code in Rust

Example 1 - hello rust

```
# Produce liblib.a
$ rustc --crate-type staticlib lib.rs

# Produce the binary
$ cc main.c -L. -llib -o example1

$ ./example1
Hello from Rust!
```

Using C-compatible code in Rust

Example 2 - hello C

```c
// lib.c
#include <stdio.h>
char *hi = "from C!";

void hello_c(void) {
    printf("Hello ");
    fflush(stdout);
}
```

Using C-compatible code in Rust

Example 2 - hello C

```rust
// main.rs
use std::ffi::CStr;
use std::os::raw::c_char;

extern "C" {
    static hi: *const c_char;
    unsafe fn hello_c();
}

fn main() {
    unsafe {
        hello_c();
        let from_c = CStr::from_ptr(hi);
        println!("{}", from_c.to_string_lossy());
    }
}
```

Using C-compatible code in Rust

Example 2 - hello C

```
# Produce liblib.a
$ cc lib.c -c -o lib.o && ar rcs liblib.a lib.o

# Produce the binary
$ rustc --crate-type bin main.rs -L . -l static=lib

$ ./main
Hello from C!
```