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 see most of the languages **AND** build systems shortcomings.
- Writing complex mutimedia code you need performance **AND** keep the code **robust** at the same time.

Introduction

- Picking a language for a project

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.

> And this is why **most** of the multimedia software is written in C.

- 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.

Simple examples

Before delving in the details of actual projects let's start with simplified examples of increasing complexity, all around **Hello word**.

- Writing C-compatible code in Rust
- Using C-compatible code in Rust
- Using a C-compatible dynamic library in Rust
- Making a C-compatible dynamic library written in Rust
    - Making it proper

Writing C-compatible code in Rust

- We have `lib.rs` that contains `hello_rust()`.
- We have `main.c` with an hand-crafted reference to it and a `main()` using it.
- We want to produce a single executable out of it.

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** has 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

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

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

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

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

Writing C-compatible code in Rust

Example 1 - hello rust

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

# Produce the link line see rust-lang/rust#61089
$ NATIVE_LIBS=`rustc --crate-type staticlib \
  --print native-static-libs 2>&1 - < /dev/null | \
  grep native-static-libs | cut -d ':' -f 3`

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

$ ./main
Hello from Rust!
```

Concerns and hurdles

Replace small internal components

- The hard parts are easy:
  - ABI compatibility
  - Object code generation (static archive)
- Getting the correct link line is more complex than it should
  - The way we obtain the `native-static-libs` is brittle.
- The actual integration looks *simple* (sort of), but is **not**
  - You **link** a static library as **usual**, but how to produce it?
    > You normally do not use just `rustc` alone
  - You would not like to hand-craft the exported symbols list.

Concerns and hurdles

Build system support

- No build system support rust and C **equally well** at the same time. (meson is far from being useful and bazel is usually the **wrong** solution)
- Calling cargo from an host build system is usually the path most consolidated project take.
  - many details can be controlled by env vars.
  - cargo metadata and cargo build --build-plan can provide to the caller plenty of information.
- Using cargo to build the C code is **feasible** but makes more sense when you are importing C code in a rust project.

From C to Rust: Concerns and hurdles

Build system support - one build system

- The **meson** native rust support is complete
    - It is not crate-aware.
    - It is dequate if you are writing something tiny and `std-only`.
    - Help in integrating the cargo build plan system to overcome those limitations is probably welcome.
- Do **not** use bazel
    - **Really no.**

Concerns and hurdles

Build system support

- Call cargo from your original build system (as seen in librsvg)

Pros	Cons
Easy to start having something working and build from there	Maintaining the project requires knowing the **two different** build systems.
You can **copy** what others did for their project and be happy as they are.	The two toolchain share the **least** amount of information about one another.
	Getting cross compiling requires a **decent** amount of skill

Concerns and hurdles

Making things nicer

- Use cbindgen to generate the C headers.
- Use cargo-vendor to *optionally* provide all the source dependencies.
- Never be tempted to reinvent the wheel and **duplicate** what cargo does.

Using C-compatible code in Rust

- We have a `lib.c` with two symbols we want to use:
  - A pointer to a constant `NULL-terminated` array of `char`.
  - A function, `hello_c` that calls `printf`.
- We have a `main.rs` that refers-to and uses them.
- We want to build an executable.

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

```
// 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.so
$ cc lib.c -c -o lib.o && ar rcs liblib.a lib.o

# Assume the libc link-line is implicit and correct
# There is no portable way to discover it short of
# trial and error anyway.

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

$ ./main
Hello from C!
```

Concerns and hurdles

- It is still pretty simple for the hard parts
  - `extern "C"` let us expose the C symbols to the compiler.
  - `unsafe` let us use them.
- Some steps are slightly different
  - There is an additional call to `ar` for symmetry.
  - The link-line is implicit, otherwise we'd have to get **creative**.
- There is still a lot that could be automated regarding symbol importing and build systems.

Solutions

Use C-ABI symbols from Rust

- bindgen can parse the **C** headers to expose the symbols to **Rust**.
- Calling **C** functions from **Rust** is as easy as calling any other `unsafe` code.
  - Bare pointers (`*mut ptr`, `*const ptr`) can be wrapped in normal structs and `Drop` can be implemented on them to make the memory management simple.
- Building **foreign** code from `cargo` is simple thanks to cc-rs, nasm-rs and, if the needs arise is feasible to use cmake or autotools with minimal hurdle.
- metadeps and pkg-config make even easier link external libraries.

Using a C-ABI dylib in Rust

- Assume we have a `libhello` providing its **platform-specifically-named** library.

  > It requires quite a bit of platform knowledge to produce a **correct** dynamic library

- We want to link it to our `main.rs` as before.

Using a C-ABI dylib in Rust

```
# Produce the dynamic library
# Depending on the platform you could have
#   args="-shared -Wl,-soname,liblib.so.1"
#   libprefix="lib"
#   ext=".so"
# or
#   args="-shared"
#   args+=" -Wl,-install_name,${p}/liblib.1.2.3.dylib"
#   args+=" -Wl,-current_version,1.2.3 "
#   args+=" -Wl,-compatibility_version,1 "
#   libprefix="lib"
#   ext=".dylib"
# or ...
$ cc ${args} lib.c -c -o ${p}/{$libprefix}lib.${ext}

# Produce the binary
$ rustc --crate-type bin main.rs -L${p} -l dylib=lib

$ ./main
Hello from C!
```

Concerns and hurdles

Using a C-ABI dylib in Rust

- It is no different from the static library situation
- There are no **Rust**-specific problems, and at least few platform-specific issues are well hidden
  - Hi **Windows**!

Using a C-ABI dylib in Rust

- The code remains the same as before.
- Building it for this purpose is getting more complex and with many platform specific nuances.
  - I avoided **windows** on purpose since it gets even more complex
- The **Rust** side remains unchanged.
  - The concerns about the runtime linker search paths and ABI version are the **usual** that come with the concept itself of dynamic library.

Making a C-ABI dylib written in Rust

- It is as non-straightforward as it is for C
  - You need to pass `platform-specific` flags
  - There is no `--print cdylib-link-line` to spare us some manual work.
- The way `rustc` interact with the linker is slightly more verbose

> This is where we can improve by leaps.

Making a C-ABI dylib written in Rust

```
# Produce the dynamic library
# Depending on the platform you could have
#   args="-shared -Wl,-soname,liblib.so.1"
#   libprefix="lib"
#   ext=".so"
# or
#   args="-shared"
#   args+=" -Wl,-install_name,${p}/liblib.1.2.3.dylib"
#   args+=" -Wl,-current_version,1.2.3 "
#   args+=" -Wl,-compatibility_version,1 "
#   libprefix="lib"
#   ext=".dylib"
$ rustc -C link-arg=${args} --crate-type cdylib lib.rs
$ cp target/debug/{$libprefix}lib.${ext} ${p}

# Produce the binary
$ cc main.c -L${p} -llib -o main

$ ./main
Hello from Rust!
```

Concerns and hurdles

Making a C-ABI dylib written in Rust

- Crafting a **proper** dynamic library is non-trivial once you want to support more than 1 platform.
  - Linux and most *BSD are nearly straightforward
  - macOS is a little more verbose
  - I omitted how to deal with **Windows** because it won't fit the slides...
- Omitted from the example but needed in real-life:
  - We should provide a pkg-config
  - We should provide a proper header file.

Tools and integrations to make our life simpler

Because nobody wants to call `rustc` and `cc` directly

Nor edit the binary sections to add the correct version information.

Nor hand-craft symbol lists (`.rs` <-> `.h`)

Nor hand write pkg-config `.pc` files

From C to Rust (and Back)

Integration options as used in the real world

- 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 production pipeline (crav1e at Vimeo)

From C to Rust (and Back)

librsvg - Moving code from C to Rust

- Making `cargo` and `autootools` talk to each other somehow

rav1e - Moving code from Rust to Assembly

- Integrate nasm in `cargo` to build **x86_64**-specific SIMD.

crav1e - Give rav1e a C interface

- Produce a correct dynamic library, header and pkg-config file

rav1e - Integrate back *crav1e* using `cargo-c`

- Do all `crav1e` does, but in a less invasive and more straightforward way

librsvg - autotools integration

- autoconf
    - use `AC_CHECK_PROGS` to check for `cargo` (and `rustc`)
    - Check for the correct minimum version
- automake
    - Keep a list of rust sources in the `Makefile.am`
    - Bind the staticlib building to make custom targets
        - Make sure to use the correct target path
    - Use `cargo` `build.rs` to pass through variables set in the `Makefile.am`.
    - Link the static libraries together and have `libtool` deal with the dynamic library problem.

librsvg

`configure.ac`

```
AC_CHECK_PROGS(CARGO, [cargo], [no])
AS_IF(test x$CARGO = xno,
    AC_MSG_ERROR([cargo is required.  Please install the 
)
...
```

`Makefile.am`

```
$(RUST_LIB): $(RUST_SRC)
        +cd $(top_srcdir)/rsvg_internals &&        \
        PKG_CONFIG_ALLOW_CROSS=1                   \
        PKG_CONFIG='$(PKG_CONFIG)'                 \
        CARGO_TARGET_DIR=$(CARGO_TARGET_DIR)       \
        $(CARGO) --locked build $(CARGO_VERBOSE)   \
        $(CARGO_TARGET_ARGS) $(CARGO_RELEASE_ARGS) \
        --features "c-library"
```

librsvg

- The symbol lists are hand-crafted
- Relies on `gtk-rs` crates to access `glib`, `cairo`, etc...
- Uses `cargo-vendor` to provide the source snapshots including the dependencies.

rav1e - `build.rs`

- Manually import the assembly symbols and use the combination of `feature="nasm"` and `target_arch = "x86_64"`.
- Leverage nasm-rs to build the a static library with the assembly code.
- Use `cargo:rustc-link-lib=static` to link it.
- The decoding tests leverage `aom-rs` and `dav1d-rs` to keep everything tidy.

rav1e

```
// build.rs
#[cfg(feature = "nasm")]
fn build_nasm_files() {
```

```
let dest_path = Path::new(&out_dir).join("config.asm");
let mut config_file = File::create(dest_path).unwrap();
config_file.write(b"  %define private_prefix rav1e\n")?;
```

```
nasm_rs::compile_library_args(
      "rav1easm",
      &[
          "src/x86/data.asm",
```

```
println!("cargo:rustc-link-lib=static=rav1easm");
rerun_dir("src/x86");
rerun_dir("src/ext/x86");
```

rav1e

```
// predict.rs
#[cfg(all(target_arch = "x86_64", feature = "nasm"))]
macro_rules! decl_angular_ipred_fn {
  ($f:ident) => {
    extern {
      fn $f(
        dst: *mut u8, stride: libc::ptrdiff_t,
        topleft: *const u8,
        width: libc::c_int, height: libc::c_int,
        angle: libc::c_int
      );
    }
  };
}
```

```
#[cfg(all(target_arch = "x86_64", feature = "nasm"))]
decl_angular_ipred_fn!(rav1e_ipred_dc_avx2);
```

rav1e

- The lack of an `asmbindgen` makes hand-crafting the symbol import a must.
    - In rav1e we automate it with some macros
- `nasm-rs` works decently even if it is basically rav1e-maintained nowadays.
    - Integrating it in `cc-rs` would avoid some effort duplication
- Leveraging external crates instead of integrating directly `aom` and `dav1d` made the `build.rs` incredibly simpler.
    - Before `cmake-rs` was used to build a custom `aom` .

crav1e - `build.rs` + `Makefile`

- `Cargo.toml`
  - Set the library crate type to `cdylib` and `staticlib`
- `build.rs`
  - Use cbindgen to generate the `.h`
  - Use cdylib-link-lines to link the library correctly
- `Makefile`
  - Produce the `.pc` file
    - Extract the `native-static-libs`
  - Install target
  - c-examples (for testing)

crav1e - `build.rs`

```
extern crate cbindgen;
extern crate cdylib_link_lines;

fn main() {
    let crate_dir =
        std::env::var("CARGO_MANIFEST_DIR").unwrap();
    let header_path: std::path::PathBuf =
        ["include", "rav1e.h"].iter().collect();

    cbindgen::generate(crate_dir)
        .unwrap()
        .write_to_file(header_path);

    println!("cargo:rerun-if-changed=src/lib.rs");
    println!("cargo:rerun-if-changed=cbindgen.toml");

    cdylib_link_lines::metabuild();
}
```

crav1e - `Makefile`

```
# Getting the native-static-libs
rav1e.pc: data/rav1e.pc.in Makefile Cargo.toml
        sed -e "s;@prefix@;$(prefix);" \
            -e "s;@libdir@;$(libdir);" \
            -e "s;@incdir@;$(incdir);" \
            -e "s;@VERSION@;$(VERSION);" \
            -e "s;@PRIVATE_LIBS@;$$(rustc --print \
             native-static-libs /dev/null \
             --crate-type staticlib 2>&1 | \
             grep native-static-libs | \
             cut -d ':' -f 3);" data/rav1e.pc.in > $@
```

crav1e - `Makefile`

```
# Horrible hack for Msys
#
# This only works natively building on Windows
# with a GNU toolchain Rust. Tested in MSYS2.
ifneq ($(OS),Linux)
ifneq ($(OS),Darwin)
OS=$(shell uname -o)
STATIC_NAME=rav1e.lib
endif
endif

SO_NAME_Darwin=librav1e.dylib
SO_NAME_MAJOR_Darwin=librav1e.$(VERSION_MAJOR).dylib
SO_NAME_INSTALL_Darwin=librav1e.$(VERSION).dylib
SO_NAME_Linux=librav1e.so
SO_NAME_MAJOR_Linux=librav1e.so.$(VERSION_MAJOR)
SO_NAME_INSTALL_Linux=librav1e.so.$(VERSION)
SO_NAME_INSTALL_Msys=rav1e.dll
SO_NAME_Msys=rav1e.dll
```

crav1e - `Makefile`

```
install: target/$(build_mode)/$(STATIC_NAME) rav1e.pc incl
        -mkdir -p $(INCDIR) $(LIBDIR)/pkgconfig
        cp rav1e.pc $(LIBDIR)/pkgconfig
        cp target/$(build_mode)/$(STATIC_NAME) $(LIBDIR)
        cp target/$(build_mode)/$(SO_NAME_$(OS)) $(LIBDIR
ifneq ($(OS),Msys)
        ln -sf $(LIBDIR)/$(SO_NAME_INSTALL_$(OS)) $(LIBDI
        ln -sf $(LIBDIR)/$(SO_NAME_INSTALL_$(OS)) $(LIBDI
else
        cp target/$(build_mode)/$(SO_NAME_$(OS)).a $(LIBD
        cp target/$(build_mode)/rav1e.def $(LIBDIR)/rav1e
endif
        cp include/rav1e.h $(INCDIR)
```

crav1e

- We have the build logic scattered in at least 2 places: `build.rs` and `Makefile`
    - The OS-specific logic is scattered around multiple and it makes supporting cross compiling more **convoluted** than it should.
- Most of the `build.rs` lives in **reusable** crates already
    - `cdylib-link-line` is simple enough and lightweight.
    - `cbindgen` adds up a lot for the build process time, calling it as executable from the `Makefile` could improve the build time while adding a preparation step for the user.

crav1e

- The `pkg-config` -generation can be improved.
  - Extracting the `native-static-libs` is fairly brittle, but there is no way out of it for now.
  - Extracting information from `Cargo.toml` in the `Makefile` requires parsing `toml` using `grep` and `sed` or adding the dependency for a json query tool.
- The C-API bindings are quite small, yet they **need** a separate crate since the build machinery is too cumbersome to live in the main crate.
  - Some wannabe users are confused by the two crates existence
  - API evolution is non-atomic

cargo-c

- A cargo applet that integrates `cbindgen`, `cdylib-link-line` and `pkg-config-gen`.
- Builds and **installs** `.h`, `.pc` and libraries with a simple command
- Supports **macOS**, **Linux** and **Windows** (only msys2 for now)
- All the information required to work is provided by `Cargo.toml` (and `cbindgen.toml`)
- Requires **minimal** changes to the main crate to build if any.

cargo-c

Requirements

- The `crate` must have a `lib` target
  - And the target crate **must** be the first of the workspace due a `cargo-metadata` limitation.
- The C-API code can be guarded using `#[cfg(cargo_c)]`
  - It **must** the first module in the library due `cbindgen` limitations.
- Only a single library and header can be produced per crate (`cbindgen` and `cargo-c` limitation)

cargo-c use-case

- Less invasive
  - No complex `build.rs`
  - No duplicated logic in multiple places
- All the complexity is hidden
  - All the `cdylib`-craft is embedded in `cargo-c`
- No additional build systems and non-rust dependencies:
  - All you need is `cargo install cargo-c`
- The install phase is packager-friendly:
  - The usual `destdir`, `libdir`, `includedir` overrides are easy to pass.

Moving crav1e back into rav1e - Why

- `crav1e` works and it is used already by Vimeo with not many issues
- **But** keeping the two crates in sync when we change the API is additional work
  - Double review
  - Double release churn
- The `make` -based build system works
- **But** is not straightforward.
  - `DESTDIR` , `LIBDIR` , etc are pseudo-standard, but not discoverable.
  - Extracting information from `cargo` using `make` is cumbersome.

Moving crav1e back into rav1e

- Add a `src/capi.rs` with the C-API bindings.
- Reference to this module as the **first** in `src/lib.rs`
- Use `#[cfg(cargo_c)]` to conditionally build it.
- Copy the `cbindgen.toml`
- ...
- That's all!

TL;DR

```
$ git clone https://github.com/xiph/rav1e
$ cd rav1e
$ cargo cinstall --prefix=/usr --destdir=/tmp/rav1e-out
$ lsd --tree /tmp/rav1e-out
rav1e-out
└── usr
    ├── include
    │   └── rav1e
    │       └── rav1e.h
    └── lib
        ├── librav1e.0.1.0.dylib
        ├── librav1e.0.dylib
        ├── librav1e.a
        ├── librav1e.dylib
        └── pkg-config
            └── rav1e.pc
```

Questions?