Using Rust Code in a C/C++ Project with CMake

Devolutions

A NEW SERIES CALLED THE INSIDER SERIES

As you hopefully know, we publish different series in our blog. For example, The Basics covers essential Remote Desktop Manager functions, while Case Studies highlight examples of IT pros using our products to solve their problems. And for fun, there's always our Monthly Poll and various quizzes (like this recent one that measures your Geek Level).

Today, I'm happy to introduce you to a new series called the **Insider Series**. As you might guess from the title, this series features articles by Devolutions' Software Developers and explores the tools we use to develop our products. We hope that you find this new series interesting and that you find something for your own projects.

To get started, here's my article called "Using Rust Code in a C/C++ Project with CMake".

I am currently working on Wayk Now written in C for the most part. We recently started writing new code for it in Rust which I'm super excited about.

As we needed a way to integrate this code in our CMake build1, we wrote a set of modules for CMake which can found on the following repository:

https://github.com/wayk/CMakeRust

CMakeRust makes it possible to generate a static library with cargo and use it as a dependency. We currently handle the following:

- automatically find rustc and cargo
- build for Windows, Linux, macOS, Android and iOS
- build for 32 or 64 bit
- build in release or debug depending on the project configuration

A use-case we do not currently cover is adding a library as a dependency directly from crates.io, but rust libraries in the project can use dependencies as usual.

HOW TO USE IT

Let's go over a small example where we will generate a rust library and call its functions from C++. You can find the completed example here.

We will start with a basic C++ project containing two files, CMakeLists.txt and main.cpp.

```
1  cmake_minimum_required(VERSION 3.0)
2
3  project(CMakeRustSample)
4
5  add_executable(helloworld main.cpp)
```

CMakeLists.txt hosted with ❤ by GitHub view raw

```
1  int main(int argc, char* argv[]) {
2          return 0;
3  }
```

main.cpp hosted with ❤ by GitHub view raw

First we copy the folder CMakeRust/cmake/ to our project and add the following lines to CMakeLists.txt to use the modules.

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/")

enable_language(Rust)
include(CMakeCargo)
```

CMakeLists.txt hosted with ❤ by GitHub view raw

Next we add a rust library, here I used the name test-lib. An important detail is that we need to set the crate type to staticlib in the Cargo.toml.

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

gistfile1.txt hosted with ❤ by GitHub view raw

To make functions accessible from C/C++ we annotate them with #[no_mangle] and declare them pub extern "C". Note that we only need to do this for the exported functions, it is not required for the code that is only used inside the library.

```
#[no_mangle]
pub extern "C" fn print_hello() {
    println!("hello world!");
}
```

hello.rs hosted with ❤ by GitHub view raw

In our library folder we create a CMakeLists.txt file with the following content. This tells CMake to generate the library and make it available as a dependency.

```
cargo_build(NAME test-lib)
```

CMakeLists.txt hosted with ❤ by GitHub view raw

We create a header file to declare the library functions and include it in main.cpp.

```
#ifndef TEST_LIB
#define TEST_LIB

#ifdef __cplusplus
extern "C" {
#endif

void print_hello();

#ifdef __cplusplus
}
#endif

#endif /* TEST_LIB */
```

test_lib.h hosted with ❤ by GitHub view raw

The last step is to add test-lib as a dependency to the executable. On Windows, we also need to add ws2_32 and userenv as the Rust runtime depends on them. Here is the final CMakeLists.txt.

```
cmake_minimum_required(VERSION 3.0)

project(CMakeRustSample)

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/")

enable_language(Rust)
include(CMakeCargo)

include_directories("include")
add_subdirectory(test-lib)

add_executable(helloworld main.cpp)
target_link_libraries(helloworld test-lib ws2_32 userenv)


```

We're all set to call our function from our C++ program.

```
#include "test_lib.h"

int main(int argc, char* argv[]) {
    print_hello();
}
```

main.cpp hosted with ❤ by GitHub view raw

We can now generate the project and build our program.

```
ekse@eclipse:~/rust/cmake_rust_sample/build$ cmake ..
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Cargo Prefix: /home/ekse/.cargo
-- Rust Compiler Version: 1.25.0
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ekse/rust/cmake_rust_sample/build
ekse@eclipse:~/rust/cmake_rust_sample/build$ make
Scanning dependencies of target test-lib_target
[ 33%] running cargo
   Compiling test-lib v0.1.0 (file:///home/ekse/rust/cmake_rust_sample/src/test-lib)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
[ 33%] Built target test-lib_target
Scanning dependencies of target helloworld
[ 66%] Building CXX object CMakeFiles/helloworld.dir/src/main.cpp.o
[100%] Linking CXX executable helloworld
[100%] Built target helloworld
ekse@eclipse:~/rust/cmake_rust_sample/build$ ./helloworld
hello world!
```

[1] found a really helpful answer on Stack Overflow that shows how to achieve this using ExternalProject. We also came across RustCMake which does not seem to be actively maintained.