

Hands-on Workshop: Incremental Migration of C-code to Rust

Alicia Andries, Jorn Lapon, Stijn Volckaert
firstname.name@kuleuven.be
KU Leuven, DistriNet@Gent

1 Introduction

More and more companies and open-source projects are introducing Rust to their existing code base with the aim to make their code more secure. Linux, Mozilla and Google are a few examples. This workshop is made to help you get started on following in their footsteps. Different projects have different aims when introducing Rust. Some projects simply want new features to be written in Rust as it lowers the chance of introducing memory errors into an otherwise stable project. Other projects want to translate as much of their code base as possible to Rust. Another option is to simply translate the most critical parts in your code base. As Linux is open-source it is an interesting project to follow. They have chosen to introduce the use of Rust when creating new drivers, but they also translate some of their existing C code to support those new Rust drivers. In this workshop we will be introducing Rust to a C project by translating some preexisting C code to Rust so you get both experience in coupling your Rust code to your C code and experience in translating C to Rust. If you end up in a project where only new code is written in Rust you can skip quite a few steps in this workshop, however, there are quite a few tips on best practices on C and Rust interoperability throughout the whole workshop.

The workshop starts with simply setting up and building the project as is. In chapter 2 the real work begins, here we start by translating a C file, `vector.c`, that contains a lot of memory manipulations and is used in multiple parts of the project. Because of the many memory manipulations the file is more likely to contain memory errors. The translation to Rust is done automatically with the help of `C2rust`. This automatic translation is a pure syntactical translation, therefore, the code will have just as few memory safety guarantees as before. Once we have this new Rust code we still need to connect it to the rest of the project which is written in C. This step is done in chapter 3. Once the Rust code can compile and interact with the C code we will take a closer look at the Rust code. The automatically generated Rust code is rather difficult to read and it uses quite a lot of non-idiomatic functions. We will replace these in chapter 4. In chapter 5 we go through the Rust code step by step to look at how we can remove the use of *unsafe* Rust (i.e., Rust code the compiler can't check for memory safety) as much as possible.

i If you want to learn more about the interoperability between C and Rust, take a look at these links:
[A Little Rust With Your C](#)
[A Little C With Your Rust](#)
Adding Rust to your C++ project? This will help you along:
<https://cxx.rs/index.html>

1.1 The Project used in this tutorial

The C program that we will introduce Rust to in this tutorial emulates an IoT socket server that is set up to receive and process packets from IoT temperature sensors. To facilitate high packet rates the server's three main responsibilities are parallelized by using dedicated threads that communicate using a shared buffer.

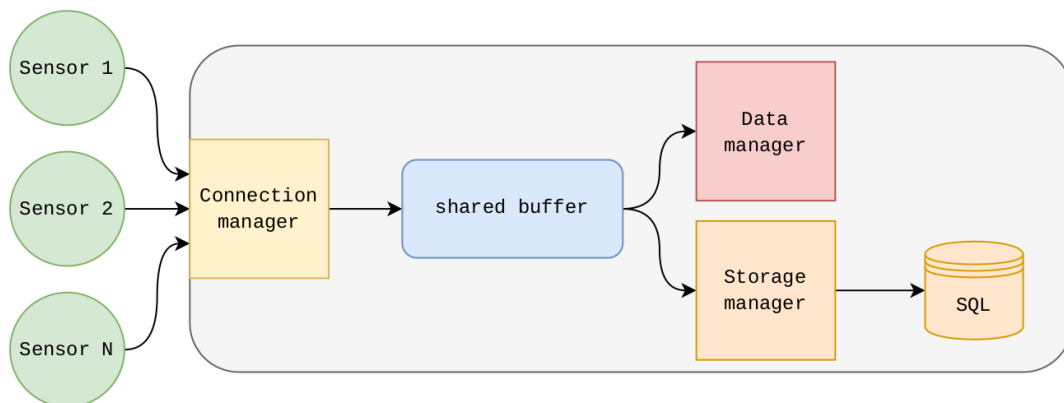


Figure 1: Server Architecture.

The three aforementioned responsibilities can be seen in the figure above. Firstly, the *Connection manager* receives data from the various sensors and writes it into the shared buffer. The *Storage manager* and *Data manager* both read from the shared buffer and, respectively, write the temperature data into a database and log concerning temperature values.

You can recognise the major components in the file structure of the project (the storage manager is contained in `sensor_db`). The `.h` files are in the same directory as their corresponding `.c` files, there is no separate include directory. The file we will be translating is `vector.c` inside the `lib` directory. This vector implementation is used by the *Connection manager*, *Data manager* and *shared buffer*. As this file defines an implementation of a vector, it contains a lot of memory manipulations, this means there is a high chance of introducing memory errors and, therefore, is the perfect candidate.

```
root
├── lib
│   ├── CMakeLists.txt
│   ├── tcpsock.c
│   ├── vector.c
│   └── CMakeLists.txt
├── connmgr.c
├── datamgr.c
├── sbuffer.c
├── sensor_db.c
├── sensor_node.c
└── config.h
```

As you can deduce from the two `CMakeLists.txt` file, this project uses CMake. CMake is a build system that can generate build files for a couple of languages, most notable C and C++. It does not have support for Rust. CMake does not build your project on its own, it generates *Makefiles* so `make` can build the project. The `CMakeLists.txt` contain the build configuration of your project.

1.2 Layout of the Workshop


This handout can be read as one flowing text, however, you might notice there are some colourful boxes throughout. There are two types of boxes, the action box, and the info box. If you get to an action box, it simply means it is time to do something like changing some code or installing a tool.

Action boxes look like this:

 *Do something ...*

Info boxes give some extra background information such as where you can find extra information on tools used in this workshop.

Info boxes look like this:

 *Background information ...*

1.3 Setting Up

The source code of the project can be found on GitHub: https://github.com/AliciaAndries/rustiec_workshop_steps. The repository contains two branches, the main branch and the solutions branch. The main branch contains the starting point of this workshop while the solutions branch contains all the transformations implemented on the code step by step. The different steps are marked with tags. When an action corresponds to such a tag, the action will have the tag's name as title.

This workshop will be using Docker as the project only works on a Linux system. The action below explains how to set up the provided Docker step by step. In the GitHub `README` you can follow the full setup guide should you not want to use Docker. Be forewarned, we might not be able to help you if you have system related issues when running the project.

We will be using VS Code as it has handy Docker, Rust and C extensions. Specifically we will be using the following extensions: rust-analyzer, clangd, Docker, and Dev Containers. The rust-analyzer extension adds Rust language support to VS Code and Clangd adds C/C++ language

support. The Docker extension allows you to see the available Docker images and containers while the Dev Containers extension allows you to connect a VS Code window to a running Docker container. These Docker extensions make interacting with code inside a Docker container easier.

\$ We will start with installing all the tools used in this project.

- The installation steps for Docker on Windows, Linux, and macOS can be found here: <https://docs.docker.com/get-docker/>
- If you don't have VS Code installed you can find the downloads for Windows, Linux, and macOS here: <https://code.visualstudio.com/download>
- Install the following extensions in VS Code:
 - Docker
 - Dev Containers

You can add extensions by selecting the extensions icon on the left-hand bar in the VS Code window.

- Linux only:
To make sure you can use Docker from VS Code you need to add allow docker to run without sudo instructions can be found here: <https://docs.docker.com/engine/install/linux-postinstall/>

\$ The Docker image that we will be using is build from the Dockerfile found in the GitHub of this workshop: https://github.com/AliciaAndries/rustiec_workshop_steps. Save the Dockerfile into a new directory.

To build the Dockerfile, run the following command from inside the new directory:

```
$ docker build -t c2rust_workshop:v1 .
```

Open VS Code. On the left-hand side there should be a Docker icon. When you click on it, you can see all the Docker containers and images that currently exist on your computer. Under images you should find an image with the name c2rust_workshop as can be seen in the image below.

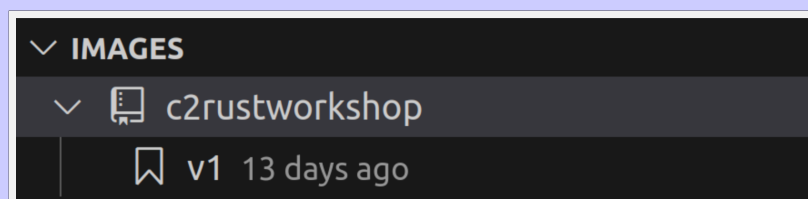


Figure 2: VS Code Docker extension.

Right click on v1 and select **Run interactive**. This will start a container, you should see it listed under the tab **containers**.

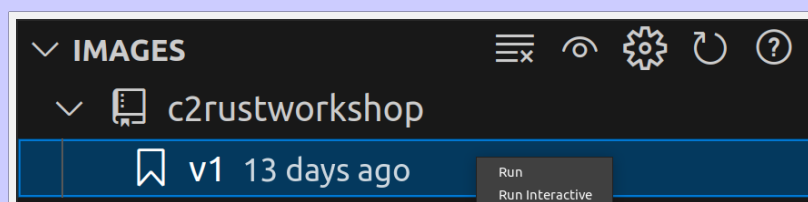


Figure 3: VS Code Docker extension.

Right click the container and select **Attach Visual Studio Code**. This will open a new

VS Code window, it should have opened the correct folder, `/rustiec_workshop_steps/`, automatically.

Under terminal in the top bar you can open new terminals connected to the Docker container.

Once you are connected with the Docker container, install the following VS Code extensions:

- [rust-analyzer](#)
- [clangd](#)

\$ To make sure you have the latest version of the project run:

```
$ git pull
```

Now that the docker is set up try and build the project.

```
$ mkdir build
$ cd build
$ cmake ../
$ mv compile_commands.json ../
$ make
```

i The clangd extension we installed in VS Code only works if it has access to the current project's `compile_commands.json`. By moving the `compile_commands.json` file to the root folder of the project the extension can locate it.

To run the project start the socket server with the command below, this command must be run from the `/build` folder.

```
$ ./server <port>
```

You can choose any non-reserved port, for example 1234. It is now possible to connect one or more sensors. The server will reach a timeout if no sensors have been added in about 10 seconds. To add a sensor run the command below in a different terminal.

```
$ ./sensor <ID> <sleep time> <server IP> <server port>
```

The IP we shall be using is `127.0.0.1`.

If you run the server or sensor without the command line options, the program will tell you which info it needs.

```
alicia@pandora:~/rustiec_workshop/sysprog/vector_translated/build$ ./sensor
Use this program with 4 command line options:
'ID'           : a unique sensor node ID
'sleep time'   : node sleep time (in sec) between two measurements
'server IP'    : TCP server IP address
'server port'  : TCP server port number
```

Figure 4: Command line options.

If you want to use gdb for debugging you can run the command below and simply attach a sensor in the same way as before.

```
$ gdb --args ./server <port>
```

The root `CMakeLists.txt` file also provides two handy debugging tools, thread sanitizer and address sanitizer. Only one can be activated at a time. If you uncomment one or the other it is not necessary to rerun `cmake`, `make` will suffice.

```
# add_compile_options(-fsanitize=address)
# add_link_options(-fsanitize=address)

# add_compile_options(-fsanitize=thread)
# add_link_options(-fsanitize=thread)
```


2 Translate the Vector Code to Rust

We will use the help of the tool C2Rust to translate the `vector.c` file to Rust. It is important to note that C2Rust is currently only capable of doing syntactical translations, not a semantic ones. Consequently, any Rust code produced by the code can give no greater memory safety assurances than the original C code. All the generated code will be marked `unsafe` and it will contain raw pointers, libc calls and any memory errors present in the original code.

Despite these limitations, C2Rust can still speed up the translation time, particularly if the sole desire is to obtain compilable Rust code. If idiomatic Rust is the goal, manual effort is still required.


In this section our only goal is to generate a Rust version of `vector.c` without without concerning ourselves with whether the code is idiomatic or not. This means we will simply run C2Rust and leave any manual translation for a later section.

To translate code with C2Rust all we need is a `compile_commands.json` file containing the necessary information on how to compile `vector.c`.

 All information on how to install and use C2Rust can be found in the [manual](#). The manual also provides information about the current limitations of C2Rust and more.

Step 1

To translated a C file to Rust using C2Rust we need the `compile_commands.json` file for that particular C file. To create such a file, it is handy to have the `compile_commands.json` file of your entire project, which we already generated in the previous action.

 If you are working on a different project where this file is not automatically generated at build time, there are tools that will create the file for you. Some of the tools you can use are listed in the [C2Rust manual under Quickstart](#).

Inside the project's `compile_commands.json` file there will be a listing pertaining to `vector.c`. This part can be copy-pasted into a new file inside of the build folder, we will call it `compile_commands_vector.json`. The file contents should look like the json code below, the absolute paths will be different on your computer.

```
1 [
2 {
3   "directory": "/home/alicia/workshop_start_point/build/lib",
4   "command": "/usr/bin/cc -Dvector_EXPORTS -fPIC -O0 -Wall
5             -Wextra -ggdb -o CMakeFiles/vector.dir/vector.c.o
6             -c /home/alicia/workshop_start_point/lib/vector.c",
7   "file": "/home/alicia/workshop_start_point/lib/vector.c"
8 }
9 ]
```

Generate `vector.rs` with C2Rust.

(If you have left the build directory, navigate to the build directory.)

```
$ c2rust transpile compile_commands_vector.json
```

The translated file is created in the same directory as the original file, so `vector.rs` can be found in the `/lib` directory.

Once we have the Rust code, we must ensure it can be compiled with cargo. Cargo expects a `cargo.toml` file, with metadata on the Rust package, and a `main.rs` or `lib.rs` file in a `src` directory. In this case, we will need a `lib.rs` file as we are building a dynamic library.

- Create a standard Cargo.toml file for building a dynamic library in the root of the project:

```

1  [package]
2  name = "vector"
3  version = "0.1.0"
4  edition = "2021"
5
6  [lib]
7  crate-type = ["dylib"]

```

- cargo expects the source files to be under the `/src` directory

```

$ cd ..
$ mkdir src
$ mv lib/vector.rs src

```

Cargo will expect a `lib.rs` file with references to all the source files that belong to the library. We must create such a file in the `src` directory with the following contents:

```

1 mod vector;

```

If there are other Rust files you would like to use in your C code you can add them to the `lib.rs` file.

- Run cargo to test if the current setup works.

```

$ cargo build

```

Multiple occurrences of the error *use of undeclared crate or module 'libc'* appear when you try to compile:

```

error[E0433]: failed to resolve: use of undeclared crate or module `libc`
--> src/vector.rs:454:22
454 |     free(vec as *mut libc::c_void);
    |                        ^^^^^ use of undeclared crate or module `libc`

```

Figure 5: `libc` is unknown.

To remedy this error we must add the `libc` crate dependency to the Cargo.toml file:

```

1  [dependencies]
2  libc = "0.2.149"

```

This time when compiling, the compilation doesn't fail. There are some warnings, but we will ignore the ones about the logic of the code. However, the warning in the figure below pertains to how the crate is compiled. According to the warning we have to move the line of code `#![feature(label_break_value)]` to the root of the module, in other words, it must be moved to `lib.rs`.

```

warning: crate-level attribute should be in the root module
--> src/vector.rs:2:1
2 | #![feature(label_break_value)]
  | ~~~~~~
= note: `#[warn(unused_attributes)]` on by default

```

Figure 6: Warning of crate-level attribute generated by C2Rust.

When the attribute is moved and we recompile, an error appears. It can be seen in the figure below.


```
error[E0554]: '#![feature]' may not be used on the stable release channel
--> src/lib.rs:1:1
1 | #![feature(label_break_value)]
  | ~~~~~ help: remove the attribute
= help: the feature 'label_break_value' has been stable since '1.65.0' and no longer requires an attribute to enable
```

Figure 7: Error after moving crate-level attribute.

The attribute `#![feature(...)]` cannot be built with the stable compiler as it is used to unlock unstable features which can only be compiled with the nightly compiler. In this case, `#![feature(...)]` is enabling the `label_break_value` feature, however, this feature has been made stable. This means we don't have to switch to the nightly compiler and all we have to do is remove the feature attribute (in other words, remove the line of code). Beware, C2Rust can still generate code that cannot be built with the stable compiler. [Tracking Issue: Translation nightly features](#) contains all the unstable features that may be generated.



If you would have to change to nightly mode, you can install the nightly toolchain with the command below.

```
$ rustup toolchain install nightly
```

Then you can activate nightly:


```
$ rustup default nightly
```

There is a new nightly version created every single day. If you want to learn more about how rust is made you can take a look at the following link: <https://doc.rust-lang.org/book/appendix-07-nightly-rust.html>

Try compiling again, all the warnings should now pertain to code logic.

3 Integrate Rust into CMake

Now we have Rust code that compiles to a shared library, it would be handy if we could compile the Rust code from our original build system. In this tutorial we only cover how to build Rust with CMake.

 Extra information about building Rust via CMake:

- [Using Unsafe for Fun and Profit](#)
- [Corrosion](#)

On the other hand, the [cmake crate](#) enables running cmake with cargo.

To integrate our new Rust code with CMake, CMake must run cargo, which will build a shared library from `vector.rs`. This shared library must replace the original library that contained `vector.c`. We may have the correct library linked to the C code, but we also have to call the Rust code from C. To make this possible, the Rust code must have a C binding. In other words, the Rust code must have a corresponding C header. As C2Rust creates an exact translation, the existing `vector.h` file used for the C implementation can be reused to call the Rust implementation.

Step 2

First, remove the code that compiles `vector.c` in `lib/CMakeLists.txt`.

```
add_library(vector SHARED vector.c)
target_compile_options(vector PRIVATE ${COMMON_FLAGS})
```

Now, in place of the compilation instructions of `vector.c`, we must add the code that compiles `vector.rs`. In other words, we must add a call to `cargo build` inside `lib/CMakeLists.txt`.

```
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
    set(CARGO_CMD cargo build)
    set(OUT_LOCATION "debug")
else ()
    set(CARGO_CMD cargo build --release)
    set(OUT_LOCATION "release")
endif()


set(RUST_CODE_SO
    "${CMAKE_CURRENT_BINARY_DIR}/${OUT_LOCATION}/libvector.so")

add_custom_target(vector ALL
    COMMENT "Compiling vector"
    COMMAND CARGO_TARGET_DIR=${CMAKE_CURRENT_BINARY_DIR} ${CARGO_CMD}
    COMMAND cp ${RUST_CODE_SO} ${CMAKE_CURRENT_BINARY_DIR}
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})

set_target_properties(vector PROPERTIES
    LOCATION ${CMAKE_CURRENT_BINARY_DIR})
```

There are a couple of steps entailed in using cargo via CMake.

The if block determines whether cargo builds in release mode or debug mode. Depending on the mode the shared library will be compiled to a different location, namely, `build/lib/debug` or `build/lib/release`.

 If you want to use gdb for debugging, build the Rust code in debug mode to get the necessary debug symbols.

Set defines where the shared object file can be found.

Add custom target defines the new target `vector`.

Set target property defines the property location of the target as the location of the shared object file. This is used later.

Watch out when copy-pasting the CMake code as there are some line breaks that have to be removed so CMake can parse the code correctly. The transgressors are the line breaks in `set` and in `set_target_properties`.

If you try to compile the project now, the following error appears:

```
CMake Error at CMakeLists.txt:25 (target_link_libraries):
  Target "vector" of type UTILITY may not be linked into another target.  One
  may link only to INTERFACE, OBJECT, STATIC or SHARED libraries, or to
  executables with the ENABLE_EXPORTS property set.
```

Figure 8: CMake error.

Due to the changes above, `vector` is no longer a library target, which means you can't link it directly like before as seen in the code below. To fix this we will simply use the location property defined above to link the shared object file directly. Concretely the code below must be replaced inside the root `CMakeLists.txt`:

```
target_link_libraries(users vector tcpsock "-lsqlite3")
```

With:

```
get_target_property(RUST_CODE_DIR vector LOCATION)
target_link_libraries(users ${RUST_CODE_DIR}/libvector.so
                        tcpsock "-lsqlite3")
```

Once again, watch out when copy-pasting the code above. Remove the line break in `target_link_libraries`.

Build the project:

```
$ cd build
$ make
```

Run the project to make sure it still works after linking the Rust code.

4 Make the Rust Code Prettier

If you open `vector.rs` and compare it to the original `vector.c` file, you will see that the number of lines of code has gone from a mere 73 to 427. Our first order of business will be to get rid of some unnecessary verbosity and improve readability.

4.1 Trim Code Size

To trim the code size down, we first have to find the main contributors to the code size. Looking through `vector.rs` you can notice a repeating pattern:

```
1 if !vec.is_null() {} else {
2     __assert_fail(
3         b"vec\0" as *const u8 as *const libc::c_char,
4         b"/home/alicia/rustiec_workshop/sysprog/vector_translated
5         /lib/vector_og.c\0"
6         as *const u8 as *const libc::c_char,
7         14 as libc::c_int as libc::c_uint,
8         (*::core::mem::transmute::<
9             &[u8; 38],
10            &[libc::c_char; 38],
11            >(b"void vector_set_size(vector_t *, int)\0"))
12         .as_ptr(),
13     );
14 }
15 'c_1316: {
16     if !vec.is_null() {} else {
17         __assert_fail(
18             b"vec\0" as *const u8 as *const libc::c_char,
19             b"/home/alicia/rustiec_workshop/sysprog
20             /vector_translated/lib/vector_og.c\0"
21             as *const u8 as *const libc::c_char,
22             14 as libc::c_int as libc::c_uint,
23             (*::core::mem::transmute::<
24                 &[u8; 38],
25                 &[libc::c_char; 38],
26                 >(b"void vector_set_size(vector_t *, int)\0"))
27             .as_ptr(),
28         );
29     }
30 };
```

This pattern is visible at the start of nearly every function. Compare the Rust and the C code, which C code corresponds to the listing above?

Solutions: The repeating Rust code corresponds with an `assert()` call. One of C2Rust's main goals is to retain the exact semantics of the C code. To ensure this, a C `assert` is not translated to a Rust `assert` and is instead translated to the code above.

\$ Step 3

As we have found a major contributor to the code size, we can trim it down by replacing the literal translation of `assert()` with the Rust `assert`. The example listing above can be replaced with:

```
1 assert!(!vec.is_null());
```

It is up to you to replace all instances of literal `assert` translations in `vector.rs`. There are 8 of these in total.

4.2 Improve readability and debuggability

The code size is now more manageable but it is still hard to read. This is in part because asserts are not the only operations that look strange when literally translated from C to Rust. Arithmetic operations fall into the same boat.

\$ Step 4


In the code snippet below you can see that all the arithmetic operations have been translated to wrapping functions. For example, the use of a `wrapping_add` instead of `+` in line 8.

```
1 #[no_mangle]
2 pub unsafe extern "C" fn vector_add(
3     mut vec: *mut vector_t,
4     mut element: *mut libc::c_void,
5 ) {
6     assert(!vec.is_null());
7
8     (*vec).size = ((*vec).size).wrapping_add(1);
9     (*vec).size;
10    (*vec)
11        .elements = realloc(
12            (*vec).elements as *mut libc::c_void,
13            ((*vec).size)
14                .wrapping_mul(::core::mem::size_of::<*mut libc::c_void>()
15                    as libc::c_ulong),
16        ) as *mut *mut libc::c_void;
17    let ref mut fresh0 = ((*vec).elements)
18        .offset((*vec).size).wrapping_sub(1 as libc::c_int
19            as libc::c_ulong) as isize);
20    *fresh0 = element;
21 }
```

We continue our quest to improve the quality of the generated Rust code. We will do this by removing `wrapping.*` functions and replacing them with standard arithmetic operations where desirable. Replacing wrapping arithmetic functions is not only advantageous because they are verbose, they can also hinder debugging.

When a program is compiled in release mode, multiplications, additions, subtractions, and all other arithmetic will wrap at run time. However, when a program is compiled in debug mode, checks on integer overflows are added as efficiency is not the primary goal. If an integer overflow is detected, it will cause a panic. Therefore, if the program you are translating is not meant to have integer overflows debug mode can help you detect any unwanted wrapping.

Remember to check the program logic before switching out wrapping functions for standard arithmetic operations. It may be that the program logic relies on the wrapping of an integer overflow.

 C2Rust aims to have an identical program before and after translation. To fulfill this aim wrapping Rust functions must be used as in C integer overflows are always wrapping. More information about integer overflow behaviour in Rust can be found in the [Rust documentation on types](#).

We will start by removing the three wrapping functions in `vector_add`. The first is a `wrapping_add` (line 8 in the listing above), the second a `wrapping_mul` (lines 14-15) and lastly a `wrapping_sub` (lines 18-19). If we replace these with the standard arithmetic operations we get the code below.

```
1 #[no_mangle]
2 pub unsafe extern "C" fn vector_add(
```

```

3     mut vec: *mut vector_t,
4     mut element: *mut libc::c_void,
5 ) {
6     assert!(!vec.is_null());
7
8     (*vec).size = (*vec).size + 1;
9     (*vec).size;
10    (*vec)
11        .elements = realloc(
12            (*vec).elements as *mut libc::c_void,
13            (*vec).size
14                * ::core::mem::size_of::<*mut libc::c_void>() as libc::c_ulong
15        ) as *mut *mut libc::c_void;
16    let ref mut fresh0 = ((*vec).elements)
17        .offset(((*vec).size - 1 as libc::c_int as libc::c_ulong) as isize);
18    *fresh0 = element;
19 }

```

Now it is up to you to replace the rest of the wrapping functions where advisable.

4.3 Use Standard libc Functions

The original C code uses some libc functions, such as `calloc`. C2Rust generates Rust bindings for these functions at the top of the file. Notably, these functions do not have the exact same arguments as their counterparts inside the `libc` crate, which is the standard Rust crate to use when you want to access any underlying platform types, functions, or constants. For example, it provides functions like `malloc` and `C` types.

This is the C2Rust version of `calloc`:

```
1 fn calloc(_: libc::c_ulong, _: libc::c_ulong) -> *mut libc::c_void;
```

This is the libc version:

```
1 pub unsafe extern "C" fn calloc(  
2     nobj: size_t,  
3     size: size_t  
4 ) -> *mut c_void
```

The different argument types are due to the difference in the definition of the `size_t` type. This discord exists because `size_t` is not the same as `usize` and different projects handle this discrepancy in their own way.



usize Is Not size_t

Once upon a time the assumption reigned that `size_t` was the same as `usize`. This meant that any Rust bindings created for C headers with the type `size_t` were translated into Rust code with the type `usize`. However, the revelation came that `usize` is not defined in the same way as `size_t`. `usize` is defined as ‘The pointer-sized unsigned integer type’. Whereas `size_t` is defined as ‘size_t can store the maximum size of a theoretically possible object of any type’. The true equivalent of `usize` is therefore not the more obvious `size_t`, but `uintptr_t`. There is no true equivalent of `size_t` currently available in Rust.

Therefore, when translating from Rust to C, in some projects the compile target is used to deduce what the type definition of `size_t` is in that particular case and then the Rust type is chosen accordingly. This can significantly harm the portability of your code.

There is a silver lining, in most architectures the `size_t` is the same as `uintptr_t` and accordingly, `size_t` is equivalent to `usize`. However, in both CHERI and the w65 abi this is not the case.

This problem will be reduced when `c_size_t` becomes a stable type in `core::ffi`.

You can read more about `size_t` vs `usize` [here](#).

C2Rust looks at the headers of the compilation target and deduces to which C type `size_t` is aliased. The Rust counterpart of that type will be given the `size_t` alias in the generated code. On the other hand, the `libc` crate assumes that `size_t` is always `usize`. We will be following this assumption as it results in code that is more platform-independent. If you are creating a project for a platform where `size_t` is not the same as `uintptr_t` either use the unstable `core::ffi::c_size_t` or set `size_t` to the correct type for your platform.



Step 5

As we are following the assumption of the `libc` crate regarding types, we will replace the C2Rust generated bindings with their `libc` crate counterparts. The upkeep of the correct bindings is now in the hands of a trusted and frequently used crate.

First of all we shall change the type definition of `size_t`.

Replace this:

```
1 pub type size_t = libc::c_ulong;
```

With the following:

```
1 pub type size_t = usize;
```

It is also possible to simply remove the type definition and replace `size_t` everywhere with `usize`.

If you are creating a project for an architecture where `size_t` is not the same as

`uintptr_t`, you can define `size_t` to be a different type.

You will get a lot of `mismatched type` compiler errors after this change.

Before dealing with these errors, remove the C2Rust generated bindings from the file. The code block that you should remove is shown below.

```
1 extern "C" {
2     fn memset(
3         _: *mut libc::c_void,
4         _: libc::c_int,
5         _: libc::c_ulong,
6     ) -> *mut libc::c_void;
7     fn memmove(
8         _: *mut libc::c_void,
9         _: *const libc::c_void,
10        _: libc::c_ulong,
11    ) -> *mut libc::c_void;
12    fn calloc(_: libc::c_ulong, _: libc::c_ulong) -> *mut libc::c_void;
13    fn realloc(
14        _: *mut libc::c_void,
15        _: libc::c_ulong
16    ) -> *mut libc::c_void;
17    fn free(_: *mut libc::c_void);
18    fn __assert_fail(
19        __assertion: *const libc::c_char,
20        __file: *const libc::c_char,
21        __line: libc::c_uint,
22        __function: *const libc::c_char,
23    ) -> !;
24 }
```

Now the compiler will tell us that there are a lot of functions used that are not defined. We will replace all the uses of the removed bindings with the use of their corresponding functions in the `libc` crate. For example, replace a call to the `calloc` function defined by the bindings:

```
1     let mut vector: *mut vector_t = calloc(
2         1 as libc::c_int as libc::c_ulong,
3         ::core::mem::size_of::<vector_t>() as libc::c_ulong,
4     ) as *mut vector_t;
```

With a call to the `calloc` function from the `libc` crate:

```
1     let mut vector: *mut vector_t = libc::calloc(
2         1 as libc::c_int as libc::c_ulong,
3         ::core::mem::size_of::<vector_t>() as libc::c_ulong,
4     ) as *mut vector_t;
```

After changing which function is called, we must also change the arguments passed to the function as they will now have different types. Mostly this comes down to removing no longer necessary type casts. The compiler will inform you on the types that must be changed.

Applying these transformations means that `vector_create` is transformed from this:

```
1 #[no_mangle]
2 pub unsafe extern "C" fn vector_create(
3     mut size: size_t
4 ) -> *mut vector_t {
5     let mut vector: *mut vector_t = calloc(
6         1 as libc::c_int as libc::c_ulong,
```



```

7     ::core::mem::size_of::<vector_t>() as libc::c_ulong,
8     ) as *mut vector_t;
9     if size != 0 {
10         vector_set_size(vector, size as libc::c_int);
11     }
12     return vector;
13 }

```

To this:

```

1  #[no_mangle]
2  pub unsafe extern "C" fn vector_create(
3      mut size: size_t
4  ) -> *mut vector_t {
5      let mut vector: *mut vector_t = libc::calloc(
6          1,
7          ::core::mem::size_of::<vector_t>(),
8      ) as *mut vector_t;
9      if size != 0 {
10         vector_set_size(vector, size as libc::c_int);
11     }
12     return vector;
13 }

```

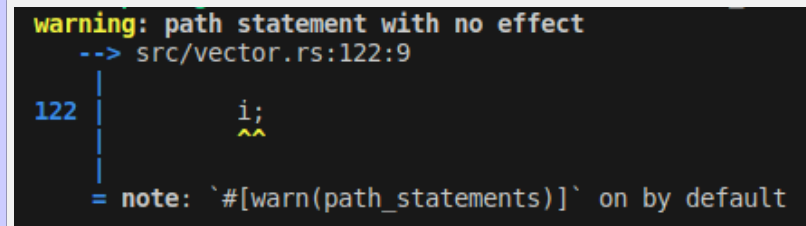
Note the call to `calloc`, lines 5 to 8. The function call is switched out for a call to the `libc` crate `calloc`. The type casts are also removed from the constant '1', which was never necessary, and from the second argument.

4.4 Remove Compiler Warnings

The biggest transformations on the code are now done, so we will now take a closer look at the warnings that remain. Currently when compiling the project only one warning remains.

\$ Step 6

If you compile the program at this point only one warning should appear, which is shown below:



```
warning: path statement with no effect
--> src/vector.rs:122:9
122 |         i;
    |         ^^
= note: `[warn(path_statements)]` on by default
```

Figure 9: `i;` has no effect warning.

This warning can easily be rectified by removing the relevant line of code.

This is not the last of the compiler warnings (and errors) we will have to fix. Currently there are quite a few errors and warnings hidden because of an allow list added by C2Rust to the top of the generated file. When adding certain lints (tools the compiler uses to help improve source code) to the allow list you can deactivate any errors or warnings they would normally create. For example the `unused_variables` lint will emit a warning when there are unused variables in your code.

\$ Step 6, Continuation

The allow list generated by C2Rust for `vector.rs` looks like this:

```
1  #![allow(
2      dead_code,
3      mutable_transmutes,
4      non_camel_case_types,
5      non_snake_case,
6      non_upper_case_globals,
7      unused_assignments,
8      unused_mut
9  )]
```

The first lint in the allow list is `dead_code`, which detects unused items and is a warn-by-default lint. When removing this item, no extra warnings appear.

The second lint, `mutable_transmutes` would normally error on transmutes from a immutable to a mutable type. This can also be removed without incurring any extra errors.

The next three lints are stylistic warnings. We will remove both `non_snake_case` and `non_upper_case_globals` with no effect. We will keep `non_camel_case_types` as this would create warnings for both `vector_t` and `size_t`.

`unused_assignments` can also be removed with no additional warnings.

The last lint is `unused_mut`, which in this case is the most important one. This lint would normally warn when any unused `mut` keywords are present. When removing this lint from the allow list quite a few warnings appear. Resolve these one by one.

Compile and run the program to make sure no new bugs were accidentally created.

4.5 Apply cosmetic changes

The initial steps to make the Rust code easier to debug and make sure it uses more standard Rust functions are nearly done. Just three small aesthetic steps will still be applied so the code is more pleasant to read.

Step 7

The first of these steps is to run the [Rust formatter](#), `rustfmt`, on the code. It can easily be downloaded by running:

```
$ rustup component add rustfmt
```

To run `rustfmt` on your code simply call the command below in the root of your Rust project.

```
$ cargo fmt
```

The second step is to remove any casts applied to literals that still remain as they are not useful. For example:

```
1 let mut i: size_t = 0 as libc::c_int as size_t;
```

Will become:

```
1 let mut i: size_t = 0;
```

The only exception is when a 0 is cast to `*mut libc::c_void` as this is done to create a null value. This can be replaced with `core::ptr::null_mut()` or, if it does not have to be mutable, `core::ptr::null()`.

The third step is removing the autogenerated `fresh_` variables or renaming them to something more descriptive. For example, in the `vector_add`:


```
1 let ref mut fresh0 = ((*vec).elements)
2   .offset(((vec).size - 1) as isize);
3 *fresh0 = element;
```

This code becomes after the removal of the `fresh_` variable:

```
1 ((*vec).elements).offset(((vec).size - 1) as isize) = element;
```

Or if you prefer renaming the variable:

```
1 let ref mut element_at_index = ((*vec).elements)
2   .offset(((vec).size - 1) as isize);
3 *element_at_index = element;
```

 Before moving on to the next part, check the compiler warnings and see if there is anything that still needs cleaning up or if there is a more serious issue. If you have any other cosmetic changes you want to apply, go for it. You can also rerun `rustfmt`.

5 Create a Safer Vector Implementation

If changing the interface that the Rust code exposes to the C code is not a problem, it is possible to change the implementation of the `vector.c` function into a wrapper for the Rust vector.

i It is also possible to make your vector implementation safer without API changes. If you would like to try this out, you can take a look at <https://stackoverflow.com/questions/39224904/how-to-expose-a-rust-vec-to-ffi> as a starting point.

As we are planning to change the interface, we will use `cbindgen` to automatically generate the corresponding `vector.h` file for our Rust implementation.

i `cbindgen` is a tool created by Mozilla, it generates C header files based on a Rust file. Such a header file makes it possible to call Rust functions from C. More information about the installation process and `cbindgen`'s features can be found on their [github](#).

Step 9

It is possible to simply call `cbindgen` via the command line on a Rust file to create the corresponding C header. However, the better option is to integrate `cbindgen` into the buildsystem, then any changes to your Rust interface will be automatically updated in its corresponding C binding. This can be done by adding a `build.rs` file, with the contents seen below, to the root of the project. The file will be noticed automatically by cargo.

```
1 // sourced from: https://rust-lang.github.io/rust-bindgen/
2
3 use std::env;
4
5 fn main() {
6     let crate_dir = env::var("CARGO_MANIFEST_DIR").unwrap();
7
8     cbindgen::Builder::new()
9         .with_crate(crate_dir)
10        .with_language(cbindgen::Language::C)
11        .generate()
12        .expect("Unable to generate bindings")
13        .write_to_file("lib/vector.h");
14 }
```

Note that the language is set to C, the default is C++. There are more `cbindgen` options available, they are all listed in the [cbindgen::Builder documentation](#). `cbindgen` will now be used at compile time and, therefore, must be added to the build dependencies in `Cargo.toml`.

```
1 [build-dependencies]
2 cbindgen = "0.26.0"
```

If you build the project `vector.h` should update, to test this remove `vector.h` and see if a new one appears. It should be identical to the original one, which is remarkable because we changed the type definition of `size_t` from `libc::u_long` to `usize`. This change is therefore not reflected in the binding even though the `cbindgen` documentation states that `usize` is mapped to its true equivalent, `uintptr_t`. When you define a type with the same name as any other C type in Rust, `cbindgen` will not ignore the type definition. For example, if you were to add the following to your Rust code:

```
1 pub type int = i32;
```

You will get the following in your C binding:

```
1 using int = uint32_t;
```

If you want to test this yourself and see what output cbindgen generates for `vector.rs`, run the command below and the output will appear in the terminal.

```
$ cbindgen src/vector.rs
```

It seems that cbindgen has opted to specifically ignore any type casts of `size_t` to enable mapping `size_t` to `usize`. We will be using this bug/feature. However, in your own code you should be cautious when using this because it may not be stable as we have found no documentation on this phenomenon.

The C header for `vector.rs` will be updated automatically now, so let's get started on making `vector.rs` safer.

\$ Step 10

As we are replacing the use of `vector_t`, its struct definition can be removed.

```
1 #[derive(Copy, Clone)]
2 #[repr(C)]
3 pub struct vector {
4     pub elements: *mut *mut libc::c_void,
5     pub size: size_t,
6 }
7 pub type vector_t = vector;
```

We shall, once again, start by transforming `vector_create`. However, this time we will return a pointer to a Rust vector, thereby changing the interface.

The current state of `vector_create` is:


```
1 pub unsafe extern "C" fn vector_create(
2     mut size: size_t
3 ) -> *mut vector_t {
4     let mut vector: *mut vector_t =
5         libc::calloc(
6             1,
7             ::core::mem::size_of::<vector_t>()
8         ) as *mut vector_t;
9     if size != 0 {
10         vector_set_size(vector, size as libc::c_int);
11     }
12     return vector;
13 }
```

We will replace the current logic for creating a vector with a call to the Rust standard library, `vec!`. We create a vector with its members initialized to `null` and its size set to the argument `size`. `vector_set_size` has been rendered unnecessary.

```
1 let vector: Vec<*mut libc::c_void> = vec![0 as *mut libc::c_void; size];
```

We can't simply return a Rust vector struct because C code and Rust code do not have the same struct layout. The order, size, and alignment of `Vec`'s fields are not guaranteed to be the same as C code would expect. If you were to attempt passing a Rust struct to a function marked as a foreign function by `extern "C"`, the Rust compiler would tell you that this is not allowed as it is not FFI (Foreign Function Interface) safe. Any Rust concept that does not map directly to something C understands is considered FFI-unsafe and is not allowed to be used in the header of a Rust function that is marked as a foreign function. To make passing a struct from Rust to C FFI-safe, one must add the attribute `#[repr(C)]` to the definition of the struct in question. This attribute makes sure the struct's layout is exactly the same as were it written in C. As we cannot make any changes to the standard library `Vec` struct, we will allocate `Vec` using a `Box`. This means that the vector struct is now allocated on the heap, which means it is accessible with a pointer. Unlike a Rust struct without `#[repr(C)]`, a pointer is considered FFI

safe. The call `into_raw` transforms the Rust idea of a pointer, a `Box`, into a raw pointer that any C code can work with.

 There is a way to get around the fact we can't change the struct definition of `Vec`. The crate `safer_ffi` provides a `Vec` type that does adhere to the C size, alignment and, padding of structs. This implementation can be found here: https://docs.rs/safer-ffi/latest/safer_ffi/struct.Vec.html.

```
1 let boxed_vec: *mut Vec<*mut libc::c_void> =
2     Box::into_raw(Box::new(vector));
```

Putting everything together we get the following function:

```
1 #[no_mangle]
2 pub unsafe extern "C" fn vector_create(
3     mut size: size_t
4 ) -> *mut Vec<*mut libc::c_void> {
5     let vector: Vec<*mut libc::c_void> =
6         vec![0 as *mut libc::c_void; size];
7     let boxed_vec: *mut Vec<*mut libc::c_void> =
8         Box::into_raw(Box::new(vector));
9     boxed_vec
10 }
```

As not a single piece of code in the function above is unsafe, the `unsafe` keyword can be removed.

```
1 #[no_mangle]
2 pub extern "C" fn vector_create(
3     mut size: size_t
4 ) -> *mut Vec<*mut libc::c_void> {
5     let vector: Vec<*mut libc::c_void> =
6         vec![0 as *mut libc::c_void; size];
7     let boxed_vec: *mut Vec<*mut libc::c_void> =
8         Box::into_raw(Box::new(vector));
9     boxed_vec
10 }
```

The next function, `vector_add`, is a function that appends an element to a vector, the current state of this function can be seen below. The logic corresponds with that of the `Vec` method `push`.

```
1 #[no_mangle]
2 pub unsafe extern "C" fn vector_add(
3     mut vec: *mut vector_t,
4     mut element: *mut libc::c_void
5 ) {
6     assert!(!vec.is_null());
7     (*vec).size = (*vec).size + 1;
8     (*vec).size;
9     (*vec).elements = libc::realloc(
10         (*vec).elements as *mut libc::c_void,
11         (*vec).size * ::core::mem::size_of::<*mut libc::c_void>(),
12     ) as *mut *mut libc::c_void;
13     ((*vec).elements).offset(((*vec).size - 1) as isize) = element;
14 }
```

The `assert` should stay, the rest of the code ought to be replaced. From the C side we shall get a pointer to a Rust vector. This pointer must be changed back to a `Box`, for this purpose `from_raw` is used. This function is considered unsafe as you are in essence casting a pointer to a `Box`.

```
1 let mut boxed_vec = Box::from_raw(vec);
```

To ensure this transformation is memory-safe, the C side must never pass a pointer to `vector_add` that did not originate from `vector_create`. There is no method to check this from the Rust side, so it should be added to the function's documentation.

Once we have a boxed Rust vector we can call `push` on it. The `Box` is automatically dereferenced.

```
1 boxed_vec.push(element);
```


Put this all together and you get the following function:

```
1 pub unsafe extern "C" fn vector_add(  
2     mut vec: *mut Vec<*mut libc::c_void>,  
3     mut element: *mut libc::c_void,  
4 ) {  
5     assert!(!vec.is_null());  
6     let mut boxed_vec = Box::from_raw(vec);  
7     boxed_vec.push(element);  
8 }
```

There is an issue in this code related to lifetimes. The variable `boxed_vec` is created from a raw pointer and is then dropped at the end of the function. The function `Box::from_raw` moves the ownership of the passed pointer to the newly created `Box`. Since Rust is unaware of any other references to the vector, the allocated memory is freed when `boxed_vec` reaches the end of its lifetime at the end of the function. To prevent the Rust compiler from deallocating the vector and thereby causing a use-after-free later on, we must call `std::mem::forget(boxed_vec)`. This function can be interpreted as a way to move ownership of a value back to your C code.

Below have the final code for `vector_add`.

```
1 #[no_mangle]  
2 pub unsafe extern "C" fn vector_add(  
3     mut vec: *mut Vec<*mut libc::c_void>,  
4     mut element: *mut libc::c_void,  
5 ) {  
6     assert!(!vec.is_null());  
7     let mut boxed_vec = unsafe{Box::from_raw(vec)};  
8     boxed_vec.push(element);  
9     std::mem::forget(boxed_vec);  
10 }
```

 When using `std::mem::forget` one must always be very careful that the value that was intended to be forgotten by Rust is actually forgotten. We will now introduce you to a particularly temperamental bug. Cast your eyes on the following code:

```
1 let mut boxed_vec = unsafe{Box::from_raw(vec)};  
2 let mut vec = *boxed_vec;  
3 vec.push(element);  
4 std::mem::forget(vec);
```

There is only a small change, `boxed_vec` is first dereferenced and the ownership of has been moved to `vec`. `push` is called on `vec` and subsequently `vec` is forgotten.

In this case the size and capacity of the vector were still allocated in the box, but my vector elements had been freed.

Lastly, this function must remain marked unsafe, not because there is a piece of unsafe code, but because it is not possible to implement safe guards to ensure that the pointer

passed to `vec` is valid. The only check that can be done, is to check whether or not the pointer is null. In other words, as the Rust Documentation states, there is a safety contract that the compiler cannot enforce. It is important to document this contract. Adding documentation to your code can be simply done with the help of `///`. Rustdoc will recognise these markings and automatically generate documentation with the provided information for your code. Best practice is to add documentation on unsafety as follows:

```
1  /// # Safety
2  /// insert what you should do to not trigger undefined behaviour.
```

It is helpful for debugging to add an unsafe marking to any unsafe Rust code. This can help you locate the possible origin of a memory error. It is no problem to add unsafe blocks to code that is already marked unsafe.

When applying these best practices you get the code below.

```
1  /// # Safety
2  /// The pointer passed to vec must originate from vector_create.
3  pub unsafe extern "C" fn vector_add(
4      mut vec: *mut Vec<*mut libc::c_void>,
5      mut element: *mut libc::c_void,
6  ) {
7      assert!(!vec.is_null());
8      let mut boxed_vec = unsafe{Box::from_raw(vec)};
9      boxed_vec.push(element);
10     std::mem::forget(boxed_vec);
11 }
```

These same practices should be applied to all the other vector functions. When you are done it is always a good idea to run `cargo fmt`.

The interface of `vector.rs` has changed so it will be necessary to change some of the types in the C files.

Compile and run the project to make sure all your changes are correct. If everything works, you have reached the end of this workshop.

6 Extra Time

If you have more time, take a look at the file `sbuffer.c`. You will see there are some `mutexes` present. If you translate this file with the help of C2Rust, the generated Rust file will not use Rust `mutexes`. It will simply call the C implementation of a `mutex`. Try and replace the C `mutexes` with the idiomatic Rust ones, you will notice that the Rust and C implementations of are really quite different.

The documentation on Rust `mutexes` is provided here: <https://doc.rust-lang.org/std/sync/struct.Mutex.html>. More information on concurrency in Rust can be found here: <https://doc.rust-lang.org/book/ch16-00-concurrency.html>.