safer-ffi-banner

# Introduction

> `safer_ffi` is a rust framework to generate a foreign function interface (or FFI) easily and safely.

This framework is primarily used to annotate rust functions and types to generate C headers without polluting your rust code with `unsafe`.

It's inspired by `#[wasm_bindgen]`. It's mainly expose Rust to C over the FFI (allowing C code calling into Rust code). However, it does have some usages for C to Rust over the FFI (callbacks or `extern { ... }` headers).

This chart shows the comparison of traditional FFI types vs ones using `safer_ffi`.

| | **Traditional FFI** | `safer_ffi` |
|---|---|---|
| Mutable pointer or NULL | `*mut T` | `Option<&mut T>` |
| Mutable pointer | `*mut T` | `&mut T` |
| Owned pointer or NULL | `*mut T` | `Option<repr_c::Box<T>>` |
| Owned pointer | `*mut T` | `repr_c::Box<T>` |

## Rust documentation

Link to the `rustdoc`-generated API documentation.

## Prerequisites

- Minimum Supported Rust Version: `1.60.0`

## Getting started

See the next chapter or the chapter on *Detailed Usage*.

> ⚠️ Warning: `safer_ffi` is still in an alpha stage. Some features may be missing, while others may be changed when further improving it.

# Quickstart

#### Small self-contained demo

You may try working with the `examples/point` example embedded in the repo:

```
git clone https://github.com/getditto/safer_ffi && cd safer_ffi
(cd examples/point && make)
```

Otherwise, to start using `::safer_ffi`, follow the following steps:

## Crate layout

#### Step 1: `Cargo.toml`

Edit your `Cargo.toml` like so:

```toml
[package]
name = "crate_name"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = [
    "staticlib",  # Ensure it gets compiled as a (static) C library
  # "cdylib",     # If you want a shared/dynamic C library (advanced)
    "lib",        # For `generate-headers`, `examples/`, `tests/` etc.
]

[[bin]]
name = "generate-headers"
required-features = ["headers"]  # Do not build unless generating headers.

[dependencies]
# Use `cargo add` or `cargo search` to find the latest values of x.y.z.
# For instance:
#   cargo add safer-ffi
safer-ffi.version = "x.y.z"
safer-ffi.features = [] # you may add some later on.

[features]
# If you want to generate the headers, use a feature-gate
# to opt into doing so:
headers = ["safer-ffi/headers"]
```

- Where `"x.y.z"` ought to be replaced by the last released version, which you can find by running `cargo search safer-ffi`.
- See the dedicated chapter on `Cargo.toml` for more info.

#### **Step 2:** `src/lib.rs`

Then, to export a Rust function to FFI, add the `#[derive_ReprC]` and `#[ffi_export]` attributes like so:

```rust
use ::safer_ffi::prelude::*;

/// A `struct` usable from both Rust and C
#[derive_ReprC]
#[repr(C)]
#[derive(Debug, Clone, Copy)]
pub struct Point {
    x: f64,
    y: f64,
}

/* Export a Rust function to the C world. */
/// Returns the middle point of `[a, b]`.
#[ffi_export]
fn mid_point(a: &Point, b: &Point) -> Point {
    Point {
        x: (a.x + b.x) / 2.,
        y: (a.y + b.y) / 2.,
    }
}

/// Pretty-prints a point using Rust's formatting logic.
#[ffi_export]
fn print_point(point: &Point) {
    println!("{:?}", point);
}

// The following function is only necessary for the header generation.
#[cfg(feature = "headers")] // c.f. the `Cargo.toml` section
pub fn generate_headers() -> ::std::io::Result<()> {
    ::safer_ffi::headers::builder()
        .to_file("rust_points.h")?
        .generate()
}
```

- See the dedicated chapter on `src/lib.rs` for more info.

### **Step 3:** `src/bin/generate-headers.rs`

```rust
fn main() -> ::std::io::Result<()> {
    ::crate_name::generate_headers()
}
```

## Compilation & header generation

```
# Compile the C library (in `target/{debug,release}/libcrate_name.ext`)
cargo build # --release

# Generate the C header
cargo run --features headers --bin generate-headers
```

- See the dedicated chapter on header generation for more info.

▶ Generated C header ( `rust_points.h` )

## Testing it from C

Here is a basic example to showcase FFI calling into our exported Rust functions:

`main.c`

```c
#include <stdlib.h>

#include "rust_points.h"

int
main (int argc, char const * const argv[])
{
    Point_t a = { .x = 84, .y = 45 };
    Point_t b = { .x = 0, .y = 39 };
    Point_t m = mid_point(&a, &b);
    print_point(&m);
    return EXIT_SUCCESS;
}
```

### Compilation command

```
cc -o main{,.c} -L target/debug -l crate_name -l{pthread,dl,m}

# Now feel free to run the compiled binary
./main
```

- ▶ Note regarding the extra `-l…` flags.

which does output:

```
Point { x: 42.0, y: 42.0 }
```

🚀🚀

# Usage

Using `safer_ffi` is pretty simple, provided one knows how C compilation works.

## TL,DR

### Cargo.toml

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

[dependencies]
safer-ffi = "..."

[features]
headers = ["safer-ffi/headers"]
```

### src/lib.rs

```
use ::safer_ffi::prelude::*;

#[ffi_export]
fn add(x: i32, y: i32) -> i32 {
    x.wrapping_add(y)
}

#[cfg(feature = "headers")]
pub fn generate_headers() -> ::std::io::Result<()> {
    ::safer_ffi::headers::builder()
        .to_file("filename.h")?
        .generate()
}
```

### src/bin/generate-headers.rs

```
fn main() -> ::std::io::Result<()> {
    ::crate_name::generate_headers()
}
```

- And run:

  ```
  cargo run --bin generate-headers --features headers
  ```

  to generate the headers.

# Cargo.toml

### `[lib] crate-type`

So, to ensure we compile a static or dynamic library containing the definitions of our `#[ffi_export]` functions (+ any code they transitively depend on), we need to tell to `cargo` that our crate is of that type:

```
# Cargo.toml

[lib]
crate-type = [
    "staticlib",  # Ensure it gets compiled as a (static) C library
                  # `target/{debug,release}/libcrate_name.a`
    # and/or:
    "cdylib",     # If you want a shared/dynamic C library (advanced)
                  # `target/{debug,release}/libcrate_name.{so,dylib}`

    "lib",        # For `generate-headers`, `examples/`, `tests/` etc.
]
```

### `[dependencies.safer_ffi]`

To get access to `safer_ffi` and its ergonomic attribute macros we add `safer_ffi` as a dependency, and enable the `proc_macros` feature:

```
[dependencies]
safer-ffi.version = "x.y.z"
```

- Where `"x.y.z"` ought to be replaced by the last released version, which you can find by running `cargo search safer-ffi` or `cargo add safer-ffi`
- If working in a `no_std` environment, you will need to disable the default `std` feature by adding `default-features = false`.

  ```
  [dependencies]
  safer-ffi.version = "x.y.z"
  safer-ffi.default-features = false  # <- Add this!
  ```

  - if, however, you still have access to an allocator, you can enable the `alloc` feature, to get the defintions of `safer_ffi::{Box, String, Vec}` *etc.*

```
[dependencies]
safer-ffi.version = "x.y.z"
safer-ffi.default-features = false
safer-ffi.features = [
    "alloc",  # <- Add this!
]
```

- You may also enable the `log` feature so that `safer_ffi` may log `error!`s when the semi-checked casts from raw C types into their Rust counterparts fail (*e.g.*, when receiving a `bool` that is nether `0` nor `1`).

```
[dependencies]
safer-ffi.version = "x.y.z"
safer-ffi.features = [
    "log",  # <- Add this!
]
```

### `[features]` headers

Finally, in order to alleviate the compile-time when not generating the headers (it is customary to bundle pre-generated headers when distributing an FFI-compatible Rust crate), the runtime C reflection and header generation machinery (the most heavyweight part of `safer_ffi`) is feature-gated away by default (behind the `safer_ffi/headers` feature).

However, when generating the headers, such machinery is needed. Thus the simplest solution is for the FFI crate to have a Cargo feature (flag) that transitively enables the `safer_ffi/headers` feature. You can name such feature however you want. In this guide, it is named `headers`.

```
[features]
headers = ["safer_ffi/headers"]
```

# src/lib.rs

### Export items to the FFI world

To do this, simply slap the `#[ffi_export]` attribute on any "item" that you wish to see exported to C.

The only currently supported such "item"s are:

- **function definitions** (main entry point of the crate!)
- `const`s
- type definitions (when not mentioned by an exported function).

> ⚠️ `static`s are not supported yet. This ought to be fixed soon.

If using non-primitive non-`safer_ffi`-provided types, then those must be `#[derive_ReprC]` annotated.

At which point the only thing remaining is to generate the header file.

## Header generation

```
// with the `safer-ffi/headers` feature enabled:
::safer_ffi::headers::builder()
    .to_file("….h")?
 // .other_optional_adapters()
    .generate()?
```

> ⚠️ Given how `safer_ffi` implements the C reflection logic as methods within a trait related to `ReprC`, the only way to generate the headers is to have that `.generate()` call be written *directly* in the library (a limitation that comes from the way the machinery currently operates), `.generate()` cannot be written in a downstream/dependent crate, such as a `bin` or an `example`.

On the other hand, it is perfectly possible to have the Rust library export a function which does this `.generate()` call.

That's why you'll end up with the following pattern:

Define a `cfg`-gated `pub fn` that calls into the `safer_ffi::headers::builder()` to `.generate()` the headers into the given file(name), or into the given `Write`-able / "write sink":

- Basic example:

```
//! src/lib.rs
#[cfg(feature = "headers")]
pub fn generate_headers() -> ::std::io::Result<()> {
    ::safer_ffi::headers::builder()
        .to_file("filename.h")?
        .generate()
}


//! src/bin/generate-headers.rs
fn main() -> ::std::io::Result<()> {
    ::crate_name::generate_headers()
}
```

- And run:

  ```
  cargo run --bin generate-headers --features headers
  ```

  to generate the headers.

▸ More advanced example (runtime-dependent header output)

# Custom Types

Custom types are also supported, as long as they:

- have a defined C layout;
- have a `#[derive_ReprC]` attribute.

### Usage with structs

```
#[derive_ReprC] // <- `::safer_ffi`'s attribute
#[repr(C)]      // <- defined C layout is mandatory!
struct Point {
    x: i32,
    y: i32,
}
```

- See the dedicated chapter on structs for more info.

### Usage with enums

```
#[derive_ReprC] // <- `::safer_ffi`'s attribute
#[repr(u8)]     // <- explicit integer `repr` is mandatory!
pub enum Direction {
    Up = 1,
    Down = -1,
}
```

- See the dedicated chapter on enums for more info.

# Motivation: safer types across FFI

In this chapter we will see why my company, Ditto, and I, chose to develop the `::safer_ffi` framework, which should help illustrate why using it can also be a good thing for you.

1. We will start with an overview of the traditional way to write Rust→C FFI,
2. We will then discuss about using idiomatic Rust types such as `Vec` and `[_]` slices in FFI, and how `::safer_ffi` helps in that regard.

# Why use `safer_ffi`?

Traditionally, to generate FFI from Rust to C developers would use `#[no_mangle]` and `cbindgen` like so:

```
#[repr(C)]
pub
struct Point {
    x: i32,
    y: i32,
}

#[no_mangle] pub extern "C"
fn origin () -> Point
{
    Point { x: 0, y: 0 }
}
```

And this is already quite good! For simple FFI projects (*e.g.*, exporting just one or two Rust functions to C), this pattern works wonderfully. So kudos to `cbindgen` authors for such a convenient, customizable and easy to use tool!

But it turns out that this can struggle with more complex scenarios. My company, Ditto, extensively uses FFI with Rust and has run into the limitations outlined below.

Learn more about Ditto's experience with FFI and Rust.

# `safer_ffi` features that traditional FFI struggles to support

- *These have been tested with `cbindgen v0.14.2`.*

## Correctly detecting `fn` pointers that use an incorrect ABI

As mentioned in the callbacks chapter, functions have an associated calling convention, and getting it wrong leads to Undefined Behavior.

▸ Example

By using `::safer_ffi`, such errors are caught (since only `extern "C" fn` pointers are `ReprC`):

```
error[E0277]: the trait bound `fn(): safer_ffi::layout::ReprC` is not
satisfied
 --> src/lib.rs:3:1
  |
3 | #[derive_ReprC]
  | ^^^^^^^^^^^^^^^ the trait `safer_ffi::layout::ReprC` is not implemented
for `fn()`
  |
  = help: the following implementations were found:
            <extern "C" fn() -> Ret as safer_ffi::layout::ReprC>
            <unsafe extern "C" fn() -> Ret as safer_ffi::layout::ReprC>
  = help: see issue #48214
  = note: this error originates in a macro (in Nightly builds, run with -Z
macro-backtrace for more info)
```

💪

## Support for complex types and respective layout or ABI semantics

Traditionally, if one were to write the following FFI definition:

```
#[no_mangle] pub extern "C"
fn my_free (ptr: Box<i32>)
{
    drop(ptr)
}
```

they would get:

```
typedef struct Box_i32 Box_i32;

void my_free(Box_i32 ptr);
```

which does not even compile.

This means that the moment you want to use types to express properties and invariants, you quickly stumble upon this limitation. This is why, traditional Rust→C FFI code uses "flat" raw pointers. **This results in `unsafe` implementations which are more error-prone**.

`::safer_ffi` solves this issue by using more evolved types:

| | Traditional FFI | `safer_ffi` |
|---|---|---|
| Mutable pointer or NULL | `*mut T` | `Option<&mut T>` |
| Mutable pointer | `*mut T` | `&mut T` |
| Owned pointer or NULL | `*mut T` | `Option<repr_c::Box<T>>` |
| Owned pointer | `*mut T` | `repr_c::Box<T>` |

▸ Example

For instance, what better way to guard against `NULL` pointer dereferences than to express nullability (or lack thereof) with `Option<_>`-wrapped pointer types?

▸ Example

## Consistent support for macro-generated definitions

Since `safer_ffi` is integrated within the compiler, it supports macros expanding to `#[ffi_export]` function definitions or `#[derive_ReprC]` type definitions.

▸ Example

## Support for shadowed paths

Since `safer_ffi` is integrated withing the compiler, the types the code refers to are unambiguously understood by both `#[derive_ReprC]` and `#[ffi_export]`.

▸ Example

This is another instance where

```
-   #[no_mangle] pub extern "C"
+   #[ffi_export]
```

saves the day 🙂

# Idiomatic Rust types in FFI signatures?

That was the main objective when creating and using `::safer_ffi`:

Why go through **the dangerously `unsafe` hassle** of:

- using `ptr: *const/mut T, len: usize` pairs when wanting to use slices?
- using `*const c_char` and `*mut c_char` and `CStr / CString / String` dances when wanting to use strings?
- losing all kind of ownership-borrow information with signatures such as:

```
// Is this taking ownership of `Foo` or "just" mutating it?
#[no_mangle] pub unsafe extern "C"
fn foo_stuff (foo: *mut Foo)
{
    /* ... */
}
```

---

> Can't we use our good ol' idiomatic `&/&mut/Box` trinity types? And some equivalent to `[_]` slices, `Vec`s and `String`s? And *quid* of closure types?

---

To which the answer is *yes!* All these types can be FFI-compatible, **provided they have a defined C layout**. And this is precisely what `safer_ffi` does:

---

> `safer_ffi` defines a bunch of idiomatic Rust types with a defined `#[repr(C)]` layout, to get both FFI compatibility and non-`unsafe` ergonomics.

---

That is, for any type `T` that has a defined C layout, *i.e.*, that is `ReprC` (and `Sized`):

- `&'_ T` and `&'_ mut T` are themselves `ReprC`!
    - Same goes for `repr_c::Box<T>`.
    - They all have the C layout of a (non-nullable) raw pointer.
    - And all three support being `Option`-wrapped (the layout remains that of a (now nullable) raw pointer, thanks to the enum layout optimization)

- `c_slice::Ref<'_, T>` and `c_slice::Mut<'_, T>` are also `ReprC` equivalents of `&'_ [T]` and `&'_ mut [T]`.
    - Same goes for `c_slice::Box<T>` (to represent `Box<[T]>`).
    - They all have the C layout of a `struct` with a `.ptr`, `.len` pair of fields (where `.ptr` is non-nullable).
    - And all three support being `Option`-wrapped too.
        1. In that case, the `.ptr` field becomes nullable;
        2. when it is `NULL`, the `.len` field can be uninitialized: ⚠️ it is thus then UB to read the `.len` field ⚠️ (type safety and encapsulation ensure this UB cannot be triggered from within Rust; only the encapsulation-deprived C side can do that).
- There is `repr_c::Vec<T>` as well (extra `.capacity` field *w.r.t.* generalization to `c_slice::Box<T>`),
- as well as `repr_c::String`!
    - with the slice versions (`.capacity`-stripped) too: `str::Box` and `str::Ref<'_>` (`Box<str>` and `&'_ str` respectively).
    - although these definitions are capable of representing any sequence of UTF-8 encoded strings (thus supporting NULL bytes), since the C world is not really capable of handling those (except as opaque blobs of bytes), `char*`-compatible null-terminated UTF-8 string types are available as well:
        - `char_p::Ref<'_>` for `char const *`: a temporary borrow of such string (useful as input parameter).
        - `char_p::Box` for `char *`: a pointer owning a `Box`-allocated such string (useful to *return* strings).

# Simple examples

To better grasp how easy and readable writing FFI code becomes, a few simple examples will be showcased here.

> ⚠️ Add more examples, mainly featuring very common requests from people that start doing FFI, such as *How to return a `Vec`*, *How to use `String`s across FFI*, *etc.*

# Simple examples: `string_concat`

```rust
#![deny(unsafe_code)] /* No `unsafe` needed! */

use ::safer_ffi::prelude::*;

/// Concatenate two input UTF-8 (_e.g._, ASCII) strings.
///
/// \remark The returned string must be freed with `rust_free_string`
#[ffi_export]
fn concat (fst: char_p::Ref<'_>, snd: char_p::Ref<'_>)
  -> char_p::Box
{
   let fst = fst.to_str(); // : &'_ str
   let snd = snd.to_str(); // : &'_ str
   format!("{}{}", fst, snd) // -------+
      .try_into() //                   |
      .unwrap() // <- no inner nulls --+
}

/// Frees a Rust-allocated string.
#[ffi_export]
fn rust_free_string (string: char_p::Box)
{
    drop(string)
}
```

▸ generates

# Simple examples: maximum member of an array

```rust
#![deny(unsafe_code)] /* No `unsafe` needed! */

use ::safer_ffi::prelude::*;

/// Returns a pointer to the maximum element of the slice
/// when it is not empty, and `NULL` otherwise.
#[ffi_export]
fn max<'xs> (xs: c_slice::Ref<'xs, i32>)
  -> Option<&'xs i32>
{
    xs  .as_slice() // : &'xs [i32]
        .iter()
        .max()
}
```

▸ generates

# The `ReprC` trait

- API Documentation

`ReprC` is the core trait around `safer_ffi` 's design.

- Feel free to look at the appendix to understand why and how.

Indeed,

---

> a function can only be marked `#[ffi_export]` if its parameters and returned value are all `ReprC` .

---

### When is a type `ReprC`?

A type is `ReprC` :

- when it is a primitive type with a C layout (integer types, floating point types, ~~`char`~~ , `bool` , non-zero-sized arrays, and `extern "C"` callbacks),

▸ Function pointers do not support lifetimes yet

- when it is a specialy-crafted type exported from the `safer_ffi` crate,
- or **when it is a custom type that is `#[derive_ReprC]` -annotated**.

## `#[derive_ReprC]`

You can (safely) make a custom type be `ReprC` by adding the `#[derive_ReprC]` attribute on it.

▸ Supported types

# Deriving `ReprC` for custom structs

## Usage

```rust
use ::safer_ffi::prelude::*;

#[derive_ReprC] // <- `::safer_ffi`'s attribute
#[repr(C)]      // <- defined C layout is mandatory!
pub
struct Point {
    x: i32,
    y: i32,
}

#[ffi_export]
fn get_origin ()
  -> Point
{
    Point { x: 0, y: 0 }
}
```

▸ Generated C header

## Usage with Generic Structs

`#[derive_ReprC]` supports generic structs:

```rust
use ::safer_ffi::prelude::*;

/// The struct can be generic...
#[derive_ReprC]
#[repr(C)]
pub
struct Point<Coordinate> {
    x: Coordinate,
    y: Coordinate,
}

/// ... but its usage within an `#[ffi_export]`-ed function must
/// no longer be generic (it must have been instanced with a concrete type)
#[ffi_export]
fn get_origin ()
  -> Point<i32>
{
    Point { x: 0, y: 0 }
}
```

▸ Generated C header

## Requirements

- All the fields must be `ReprC` or generic.
    - In the generic case, the struct is `ReprC` only when instanced with concrete

`ReprC` types.

- The struct must be non-empty (because ANSI C does not support empty structs)

## Opaque types (*forward declarations*)

Sometimes you may be dealing with a complex Rust type and you don't want to go through the hassle of recusrively changing each field to make it `ReprC`.

In that case, the type can be defined as an *opaque* object *w.r.t.* the C API, which will make it usable by C but only through a layer of pointer indirection and function abstraction:

```
#[derive_ReprC]
#[repr(opaque)] // <-- instead of `#[repr(C)]`
pub
struct ComplicatedStruct {
    path: PathBuf,
    cb: Rc<dyn 'static + Fn(&'_ Path)>,
    x: i32,
}
```

⚠️ Only braced struct definitions are currently supported. Opaque tuple structs and `enum`s ought to supported soon.

▸ Example

## Going further

▸ Transparent newtype wrapper

# Deriving `ReprC` for custom enums

## C enums

A C enum is a field-less enum, *i.e.*, an enum that only has *unit* variants.

▸ Examples

See the reference for more info about them.

### Usage

```
use ::safer_ffi::prelude:*;

#[derive_ReprC] // <- `::safer_ffi`'s attribute
#[repr(u8)]     // <- explicit integer `repr` is mandatory!
pub
enum LogLevel {
    Off = 0,    // <- explicit discriminants are supported
    Error,
    Warning,
    Info,
    Debug,
}
```

▸ Generated C header

### Layout of C enums

These enums are generally used to define a *closed* set of *distinct* integral constants in a *type-safe* fashion.

But when used from C, the type safety is kind of lost, given how loosely C converts back and forth between `enum`s and integers.

This leads to a very important point:

---

> What is the integer type of the enum discriminants?

---

With **no** `#[repr(...)]` annotation whatsoever, Rust reserves the right to choose whatever it wants: no defined C layout, so **not FFI-safe**.

With `#[repr(Int)]` (where `Int` can be `u8`, `i8`, `u32`, *etc.*) Rust is forced to use that very `Int` type.

With `#[repr(C)]`, Rust will pick what C would pick if it were given an equivalent definition.

⚠️ `#[repr(C)]` enums can cause UB when used across FFI ⚠️

▸ Click for more info

That's why `#[derive_ReprC]` makes the opinionated choice of **refusing to handle an `enum` definition that does not provide an explicit fixed-size integer representation**.

## More complex enums

⚠️ Are not supported yet.

# `#[ffi_export]`

This is a very simple attribute: simply slap it on an "item" that you wish to export to the FFI world (C), and *voilà*!

⚠️ The only currently supported such "item"s are function definitions: `const` and `static`s are not supported yet. This ought to be fixed soon.

```
use ::safer_ffi::prelude::*;

#[ffi_export]
fn adder (x: i32, y: i32) -> i32
{
    x.wrapping_add(y)
}
```

## Requirements

- **all the types used in the function signature need to be** `ReprC`

  This is the core property that ensures both the safety of exporting such functions to the FFI (contrary to the rather poor `improper_ctypes` lint and its false positives) and the associated C-header-generating logic.

- The only allowed generic parameters of the function are **lifetime parameters**.

    - That is, the following function definition is valid:

```rust
use ::safer_ffi::prelude::*;

#[ffi_export]
fn max<'xs> (xs: c_slice::Ref<'xs, i32>)
  -> Option<&'xs i32>
{
    xs  .as_slice()  // : &'xs [i32]
        .iter()
        .max()
}
```

- But the following one is **not**:

```rust
use ::safer_ffi::prelude::*;

#[derive_ReprC]
#[repr(C)]
#[derive(Default)]
pub
struct Point<Coordinate> {
    x: Coordinate,
    y: Coordinate,
}

#[ffi_export] // Error, generic _type_ parameter
fn origin<Coordinate> ()
  -> Point<Coordinate>
where
    Coordinate : Default,
{
    Point::default()
}
```

# `#[ffi_export]`

## Auto-generated sanity checks

The whole design of the `ReprC` trait, *i.e.*, a trait that expresses that a type has a C layout, *i.e.*, that it has an *associated* "raw" C type (types with no validity invariants whatsoever), means that the actual `#[no_mangle]`-exported function is one using the associated C

types in its function signature. This ensures that a foreign call to such functions (*i.e.*, C calling into that function) **will not directly trigger "instant UB"**, contrary to a hand-crafted definition.

- Indeed, if you were to export a function such as:

```
#[repr(C)]
enum LogLevel {
    Error,
    Warning,
    Info,
    Debug,
}

#[no_mangle] pub extern "C"
fn set_log_level (level: LogLevel)
{
    // ...
}
```

then C code calling `set_log_level` with a value different to the four only possible discriminants of `LogLevel` (`0, 1, 2, 3` in this case) would instantly trigger Undefined Behavior no matter what the body of `set_log_level` would be.

Instead, when using `safer_ffi`, the following code:

```
use ::safer_ffi::prelude::*;

#[derive_ReprC]
#[repr(u8)] // Associated CType: a plain `u8`
enum LogLevel {
    Error,
    Warning,
    Info,
    Debug,
}

#[ffi_export]
fn set_log_level (level: LogLevel)
{
    // ...
}
```

unsugars to (something along the lines of):

```
fn set_log_level (level: LogLevel)
{
    // ...
}

mod hidden {
    #[no_mangle] pub unsafe extern "C"
    fn set_log_level (level: u8)
    {
        match ::safer_ffi::layout::from_raw(level) {
            | Some(level /* : LogLevel */) => {
                super::set_log_level(level)
            },
            | None => {
                // Got an invalid `LogLevel` bit-pattern
                if compile_time_condition() {
                    eprintln!("Got an invalid `LogLevel` bit-pattern");
                    abort();
                } else {
                    use ::std::hint::unreachable_unchecked as UB;
                    UB()
                }
            },
        }
    }
}
```

So, basically, there is an attempt to `transmute` the input C type to the expected `ReprC` type, but such attempt can fail if the auto-generated sanity-check detects that so doing would not be safe (*e.g.*, input integer corresponding to no `enum` variant, NULL pointer when the `ReprC` type is guaranteed not to be it, unaligned pointer when the `ReprC` type is guaranteed to be aligned).

---

▸ Caveats

# `#[ffi_export]`

## Attributes

⚠️ These are not yet implemented

- **Non-`"C"` ABIs**

  Currently `#[ffi_export]` defaults to a `#[no_mangle] pub extern "C"` function definition, *i.e.*, it exports a function using the default C ABI of the platform it is compiled against (*target* platform).

  Sometimes a special ABI is required, in which case specifying the ABI is desirable.

  **Imagined syntax**: an optional `ABI = "<abi>"` attribute parameter:

  ```
  #[ffi_export(ABI = "system")]
  fn ...
  ```

- **Custom `export_name`.**

  To override the name (the *symbol*) the item is exported with (by virtue of the default `#[no_mangle]`, the item is exported with a symbol equal to the identifier used for its name), one could imagine someone wanting to develop their own namespacing tool / name mangling convention when controling both ends of the FFI, so they may want to provide an `export_name` override too.

  **Imagined syntax**: an optional `export_name = ...` attribute parameter.

- **`unsafe`-ly disabling the runtime sanity checks.**

  As mentioned in the sanity checks section, it is intended that all `#[ffi_export]`-ed functions perform some sanity checks on the raw inputs they receive, before transmuting those to the actual `ReprC` types. Still, for some functions where performance is critical and the caller of the `#[ffi_export]`-ed function is trusted not give invalid values, it will be possible to opt-out of such check when `debug_assertions` are disabled by marking each function where one wants to disable the checks with an `unsafe` parameter, such as:

  ```
  #[ffi_export]
  #[safer_ffi(unsafe { skip_sanity_checks() })]
  fn ...
  ```

  or on specific params:

  ```
  #[ffi_export]
  fn set_log_level (
      #[safer_ffi(unsafe { skip_sanity_checks() })]
      level: LogLevel,
      ...
  ) -> ...
  ```

# Callbacks

Bad pun

There are two kinds of callbacks:

- those "without captured environment", *i.e.*, functions, which can be represented as **simple function pointers**;
- and those with a captured environment, called *closures*.

# Function pointers

These are the most simple callback objects. They consist of a single (function) pointer, that is, the address of the (beginning of the) code that is to be called.

- In Rust, this is the family of `fn(...) -> _` types.

  To avoid (very bad) bugs when mixing calling conventions (ABIs), Rust includes the ABI within the type of the function pointer, granting additional type-level safety. When dealing with C, the calling convention that matches C's is almost always `extern "C"` and is *never* `extern "Rust"`.

  ⚠️ When unspecified, the calling convention defaults to `extern "Rust"`, which is different from `extern "C"`!

  This is why all function pointers involved in FFI need to be `extern`-annotated. Forgetting it results in code that triggers *Undefined Behavior* (and traditional FFI fails to guard against it)⚠️

- In C, these are written as `ret_t (*name)(function_args)`, where `name` is the name of a variable or parameter that has the function pointer type, or the name of the type being type-aliased to the function pointer type.

## Examples

| Rust | C |
| --- | --- |
| `cb: extern "C" fn()` | `void (*cb)(void)` |

| Rust | C |
|---|---|
| `f: extern "C" fn(arg1_t, arg2_t) -> ret_t` | `ret_t (*f)(arg1_t, arg2_t)` |
| `transmute::<_, extern "C" fn(arg_t) -> ret_t>(f)` | `(ret_t (*)(arg_t)) (f)` |
| `type cb_t = extern "C" fn(arg_t) -> ret_t;`<br>`let f: cb_t = ...;`<br>`transmute::<_, cb_t>(f)` | `typedef ret_t (*cb_t) (arg_t);`<br>`cb_t f = ...;`<br>`(cb_t) (f)` |

So, for instance,

```rust
#[ffi_export]
fn call (
    ctx: *mut c_void,
    cb: unsafe extern "C" fn(ctx: *mut c_void),
)
```

becomes

```c
void call (
    void * ctx,
    void (*cb)(void * ctx)
);
```

### Nullable function pointers

> ⚠️ A Rust `fn` pointer *cannot* possibly be NULL!

This means that when `NULL`-able function pointers are involved, **forgetting to `Option`-wrap them can lead to *Undefined Behavior***. Luckily, this is something that is easily caught by `::safer_ffi`'s sanity checks.

# Adding state: Closures

Since bare function pointers cannot carry any non-global instance-specific state, their usability is quite limited. For a callback-based API to be good, it must be able to support some associated state.

## Stateful callbacks in C

In C, the idiomatic way to achieve this is to carry an extra `void *` parameter (traditionally called `data`, `ctx`, `env` or `payload`), and have the function pointer receive it as one of its parameters:

▸ Example

This pattern is so pervasive that the natural thing to do is to bundle those two fields (data pointer, and function pointer) within a `struct`:

```
typedef struct MyCallback {
    void (*cb)(void * ctx);
    void * ctx;
} MyCallback_t;
```

▸ Example

## Back to Rust

In Rust, the situation is quite more subtle, since the properties of the closure are not wave-handed like they are in C. Instead, there are very rigurous things to take into account:

- `'static`

  Can the environment be held arbitrarily long, or is there some call frame / scope / lifetime it cannot outlive?

- `Send`

  Can the environment be accessed (non-concurrently) from another thread?

  - For the sake of sanity, non-`Send` closures are not supported.

- `Fn` ***vs.*** `FnMut`

  Both involve a callable API, but `FnMut` involves non-concurrent access whereas `Fn` allows concurrent access (*e.g.*, closure then has to be reentrant-safe and, when `Sync`, thread-safe too).

- `Sync` **(+** `Fn`**)**

  Is the closure thread-safe / can it be called in parallel?

> To get a better understanding of the `Fn*` traits and the `move? |...| ...` closure sugar in Rust I highly recommend reading the *Closures: Magic functions* blog post.

---

Such struct definitions are available, in a generic fashion, in `::safer_ffi`, under the `::safer_ffi::closure` module.

▸ Disclaimer about callbacks using lifetimes

For instance, `MyCallback_t` above is equivalent to using, within Rust, the `RefDynFnMut0 <'_, ()>` type, a `ReprC` version of `&'_ mut (dyn Send + FnMut())`:

▸ C layout

## Borrowed closures

More generally, when having to deal with a borrowed[?] stateful callback having `N` inputs of type `A1, A2, ..., An`, and a return type of `Ret`, *i.e.*, a `&'_ mut (dyn Send + FnMut(A1, ..., An) -> Ret)`, then the `ReprC` equivalent to use is:

```
RefDynFnMutN <'_, Ret, A1, ..., An>
```

- C layout:

```
typedef struct {
    // Cannot be NULL
    void * env_ptr; // &'_ mut TypeErased
    // Cannot be NULL
    Ret_t (*call)(void * env_ptr,
        A1_t arg_1,
        A2_t arg2,
        ...,
        An_t arg_n);
} RefDynFnMutN_Ret_A1_A2_..._An_t;
```

▸ Example: `call_n_times` in Rust

## Owned closures

When, instead, the closure may be held arbitrarily long (*e.g.*, in another thread), and may

have some destructor logic, *i.e.*, when dealing with a heap-allocation-agnostic generalization of:

```
Box<dyn 'static + Send + FnMut(A1, ..., An) -> Ret>
```

then, the `ReprC` equivalent type to use is:

```
BoxDynFnMutN <Ret, A1, ..., An>
```

▸ C layout

### Ref-counted thread-safe closures

And, finally, when, on top of the previous considerations, the closure may have multiple owners (requiring ref-counting) and/or may be called by concurrent ( `Fn` instead of `FnMut` ) and even *parallel* (added `Sync` bound) code, *i.e.*, when dealing with a heap-allocation-agnostic generalization of:

```
Arc<dyn 'static + Send + Sync + Fn(A1, ..., An) -> Ret>
```

then, the `ReprC` equivalent type to use is:

```
ArcDynFnN <Ret, A1, ..., An>
```

▸ C layout

# FFI-safe `dyn Traits` / Virtual objects

This is a generalization of callbacks. For instance, while you could model an `Iterator` as an `FnMut()` , and then make that closure FFI-compatible based on the previous chapter, you could also be tempted to instead actually use some FFI-safe version of `dyn Iterator` , right?

▸ Example (click to see)

This functionality is indeed supported by `safer-ffi` , thanks to the combination of *two* things:

1. the `#[derive_ReprC(dyn)]` annotation on a given `Trait` definition (this makes it so `dyn Trait : ReprCTrait` )

2. the `VirtualPtr<dyn Trait + …>` FFI-safe `Box`-like pointer usage in some function signature.

```
//! REAL CODE

use ::safer_ffi::prelude::*;

#[derive_ReprC(dyn)] // 👈 1
trait FfiIterator : Send {
    fn next(&mut self) -> u32;
}

#[ffi_export]
fn fibonacci()
  -> VirtualPtr<dyn FfiIterator> // 👈 2
{
    struct Fibo(u32, u32);

    impl FfiIterator for Fibo {
        fn next(&mut self) -> u32 {
            let to_ret = self.0;
            (self.0, self.1) = (self.1, self.0 + self.1);
            to_ret
        }
    }

    Box::new(Fibo(0, 1))
        .into()
}
```

- The resulting `VirtualPtr` then has an opaque data `.ptr` field, as well as a `.vtable` field, containing all the (virtual) methods of the trait, as well as the special `release_vptr` method[1].

  ▸ Generated header (click to see)

  FFI usage:

  ```
  // 1. Create it
  auto obj = fibonacci();
  // 2. Use it
  for(int i = 0; i < 5; ++i) {
      printf("%u\n", obj.vtable.next(obj.ptr));
  }
  // 3. Release it
  obj.vtable.release_vptr(obj.ptr);
  ```

  Trick:

1. ```
   #define CALL(obj, method, ...) \
       obj.vtable method(obj.ptr ##__VA_ARGS__)
   ```

2. So as to:

   ```
   // 1. Create it
   auto obj = fibonacci();
   // 2. Use it
   for(int i = 0; i < 5; ++i) {
       printf("%u\n", CALL(obj, .next));
   }
   // 3. Release it
   CALL(obj, .release_vptr);
   ```

[1] as well as the special `retain_vptr`, in the case of `#[derive_ReprC(dyn, Clone)]`

# `VirtualPtr<dyn Trait>`

`VirtualPtr` is the key **pointer type** enabling all the FFI-safe `dyn`-support machinery in `safer-ffi`.

- In order to better convey its purpose and semantics, other names considered for this type (besides a `VPtr` shorthand) have been:
  - `DynPtr<dyn Trait>`
  - `DynBox<dyn Trait>`
  - `VirtualBox<dyn Trait>` / `VBox` (this one has been *very* strongly considered)

Indeed, this type embodies **owning pointer** semantics, much like `Box` does.

But it does so with a twist, hence the dedicated special name: **the owning mode is, itself, virtual/** `dyn`!

- As will be seen in the remainder of this post, this aspect of `VirtualPtr` is gonna be the key element to allow **full type unification across even *different pointer types*!**

  For instance, consider:
   ```

```
fn together<'r>(
    a: Box<impl 'r + Trait>,
    b: Rc<impl 'r + Trait>,
    c: &'r impl Trait,
) -> [???; 3] // 🤔🤔🤔
{
    [a.into(), b.into(), c.into()]
}
```

With `VirtualPtr`, we can fully type-erase and thus type-unify all these three types into a common one:

```
) -> [VirtualPtr<dyn 'r + Trait>; 3] // 💡💡💡
```

### This allows a unified type able to cover all of `Box<dyn Trait>`, `{A,}Rc<dyn Trait>`, `&[mut] dyn Trait` under one same umbrella

---

*One type to unify them all,*

*One type to coërce them,*

*One type to bring them all*

*and in the erasure bind them.*

---

One VirtualPtr to rule them all

## Constructing a `VirtualPtr` from Rust

That is, whilst **a `Box<impl Trait>` can[1] be "coërced" `.into()` a `VirtualPtr<dyn Trait>`**, `Box` will oftentimes not be the sole pointer/indirection with that capability. Indeed, there will often be other similar "coërcions" from a `&impl Trait`, a `&mut impl Trait`, a `Rc<impl Trait>`, or a `Arc<impl Trait + Send + Sync>`!

[1] provided that `dyn Trait` be a `ReprCTrait`, *i.e.*, that the `Trait` definition have been `#[derive_ReprC(dyn)]`-annotated.

Here is the complete list of possible conversion at the moment:

1. **Given** `<T> where T : 'T + Trait,`

2. With `Trait` "being `ReprC`" / FFI-safe (*i.e.*, `dyn Trait : ReprCTrait`)

| `From<...>` | `.into()` | **Notes for** `Trait` |
|---|---|---|
| `Box<T>` | `VirtualPtr<dyn 'T + Trait>` | • (requires `T : Clone` when `Clone`-annotated) |
| `&T` | `VirtualPtr<dyn '_ + Trait>` | • cannot have `&mut self` methods |
| `&mut T` | `VirtualPtr<dyn '_ + Trait>` | • cannot be `Clone`-annotated |
| `Rc<T>` | `VirtualPtr<dyn 'T + Trait>` | • must be `Clone`-annotated<br>• cannot have `&mut self` methods |
| `Arc<T>` | `VirtualPtr<dyn 'T + Trait + Send + Sync>` | • must be `Clone`-annotated<br>• cannot have `&mut self` methods<br>• requires `T : Send + Sync` |

- Where "`Clone`-annotated" refers to the `#[derive_Repr(dyn, Clone)]` case.

### Remarks

- Whenever `T : 'static`, we can pick `'T = 'static`, so that `dyn 'T + Trait` may be more succintly written as `dyn Trait`.

- If the trait has methods with a `Pin`ned `self` receiver, then the `From<…>`-column needs to be `Pin`-wrapped.

- `+ Send` **and/or** `+ Sync` **can always be added** inside a `VirtualPtr`, in which case `T : Send` and/or `T : Sync` (respectively) will be required.

    - The only exception here is `Rc`, since `Rc<dyn Trait + Send + Sync>` *& co.* are oxymorons which have been deemed not to deserve the necessary codegen (if multiple ownership and `Send + Sync` is required, use `Arc`, otherwise, use `Rc`).

  **Tip**: Since `+ Send + Sync` is so pervasive(ly recommended for one's sanity) when doing FFI, these can be added as super-traits of our `Trait`, so that they be implied in both `T : Trait` and `dyn Trait`, thereby alleviating the syntax without compromising the thread-safety:

```
#[derive_ReprC(dyn, /* Clone */)]
trait Trait : Send + Sync {
```

    - But be aware that, even with such a super trait annotation, `dyn Trait` and `dyn Trait + Send + Sync` will remain being distinct types as far as Rust is concerned! ⚠️

### Its FFI-layout: constructing and using `VirtualPtr` from the FFI

Given some:

```
#[derive_ReprC(dyn, /* Clone */)]
trait Trait {
    fn get(&self, _: bool) -> i32;
    fn set(&mut self, _: i32);
    fn method3(&… self, _: Arg1, _: Arg2, …) -> Ret;
    …
}
```

- (with `Arg1 : ReprC<CLayout = CArg1>`, *etc.*)

A `VirtualPtr<dyn Trait>` will be laid out as the following:

```
type ErasedPtr = ptr::NonNull<ty::Erased>; /* modulo const/mut */

#[repr(C)]
struct VirtualPtr<dyn Trait> {
    ptr: ErasedPtr,
    // Note: it is *inlined* / *no* pointer indirection!
    vtable: {
        // the `drop` / `free`ing function.
        release_vptr: unsafe extern fn(ErasedPtr),

        /* if `Clone`-annotated:
        retain_vptr: unsafe extern fn(ErasedPtr) -> VirtualPtr<dyn Trait>, */

        /* and the FFI-safe virtual methods of the trait: */
        get: unsafe extern fn(ErasedPtr, _: CLayoutOf<bool>) -> i32,
        set: unsafe extern fn(ErasedPtr, _: i32),
        method3: unsafe extern fn(ErasedPtr, _: CArg1, _: CArg2, …) -> CRet,
        …
    },
}
```

## A fully virtual owning mode

Remember the sentence above?

---

> But it does so with a twist, hence the dedicated special name: **the owning mode is,**

**itself, virtual/ `dyn` !**

---

What this means is that ***all of the destructor* is virtual / `dyn` amically-dispatched**, for instance (and ditto for `.clone()` ing, when applicable).

## Non-fully-virtual examples

To better understand this nuance, consider the opposite (types which are not *fully* virtual / `dyn` amically dispatched, such as `Box<dyn …>`): what happens when you drop a `Box<dyn Trait>` *vs.* dropping a `Rc<dyn Trait>` ?

- **when you drop a `Box<dyn Trait>`:**

    1. It *virtually/ `dyn`* -amically queries the `Layout` knowledge of that `dyn Trait` type-erased data;
    2. It *virtually/ `dyn`* -amically drops the `dyn Trait` pointee *in place*;
    3. It then calls `dealloc` ( `free` ) of the backing storage using the aforementioned data `Layout` (as the layout of the whole allocation, since a `Box<T>` allocates exactly as much memory as needed to hold a `T` )

    This last step is thus *statically* dispatched, thanks to the *static*/compile-time knowledge of the hard-coded `Box` type in `Box<dyn Trait>` !

    ▸ Pseudo-code

- **when you drop a `Rc<dyn Trait>`:**

    1. It *virtually/ `dyn`* -amically queries the `Layout` knowledge of that `dyn Trait` type-erased data;
    2. It then embiggens the aforementioned layout so as to get the layout of all of the `Rc` 's actual pointee / actual allocation (that is, the `RcBox` , *i.e.*, the data alongside two reference counters), so as to be able to access those counters,
    3. and then decrements the appropriate counters (mostly the strong count);
    4. if it detects that it was the last owner (strong count from 1 to 0):
        1. It *virtually/ `dyn`* -amically drops the `dyn Trait` pointee *in place*;
        2. It then calls `dealloc` ( `free` ) for that whole `RcBox` 's backing storage (when there are no outstanding `Weak` s).

    The steps `2.` , `3.` and `4.2` are thus *statically* dispatched, thanks to the *static*/compile-time knowledge of the hard-coded `Rc` type in `Rc<dyn Trait>` !

    ▸ Pseudo-code

We can actually even go further, and wonder what Rust does:

- **when a `&mut dyn Trait` or a `&dyn Trait` goes out of scope:**

    1. Nothing.

        (Since it knows that the `&[mut] _` types have no drop glue whatsoever)

    This step (or rather, lack thereof) is another example of *statically* dispatched logic.

It should thus now be clear that:

- whilst type erasure *of the pointee* does happen whenever your deal with a `ConcretePtr<dyn Trait>` such as `Box<dyn Trait>`, `&mut dyn Trait`, *etc.*
- on the other hand, the `ConcretePtr` behind which such erasure happens is not, itself, type-erased! It is still statically-known, and functionality such as `Drop`, `Clone`, or even `Copy` may take advantage of that information (*e.g.*, `&dyn Trait` is `Copy`).

## Another example: `dyn_clone()`

Let's now compare, in the context of type-erased `dyn Trait` pointees, a static operation *vs.* a virtual / `dyn` amically dispatched one.

For starters, let's consider the following `Trait` definition:

```
trait Trait : 'static {
    //                  &dyn Trait
    fn dyn_clone(self: &Self) -> Box<dyn Trait>;
}

impl<T : 'static + Clone> Trait for T {
    fn dyn_clone(self: &T) -> Box<dyn Trait> {
        Box::new(T::clone(self)) /* as Box<dyn Trait> */
    }
}
```

and now, let's think about and compare the behaviors of the two following functions:

```
fn clone_box(b: &Box<dyn Trait>) -> Box<dyn Trait> {
    b.dyn_clone()
}

fn clone_rc(r: &Rc<dyn Trait>) -> Rc<dyn Trait> {
    r.clone() // Rc::clone(r)
}
```

- `clone_box` is `dyn` amically calling and delegating to `dyn Trait`'s `dyn_clone` virtual

method;

- `clone_rc` is statically / within-hard-coded code logic performing a (strong) reference-count increment inside the `RcBox<dyn Trait>` pointee, thereby never interacting with the `dyn Trait` value itself.

(Granted, the former is performing a statically-dispatched `Deref` coercion beforehand, and the latter may be `dyn` amically looking up `dyn Trait`'s `Layout`, but the main point still stands).

## From partially dynamic to *fully* dynamic

From all this, I hope the hybrid static-`dyn`amic nature of Rust's `ConcretePtr<dyn ErasedPointee>` (wide) pointers logic is now more apparent and clearer.

From there, we can then wonder what happens if we made it all *fully* `dynamic`: `VirtualPtr` is born!

### Summary

- *all* of the `drop` glue is to be `dyn`amically dispatched (through some virtual `fn` pointer performing a `drop_ptr` operation):

```
//! Pseudo-code
impl<T> DynDrop for Box<T> {
    fn dyn_drop_ptr(self)
    {
        drop::<Box<T>>(self);
    }
}

impl<T> DynDrop for Arc<T> {
    fn dyn_drop_ptr(self)
    {
        drop::<Arc<T>>(self);
    }
}

impl<T> DynDrop for &mut T {
    fn dyn_drop_ptr(self)
    {}
}

impl<T> DynDrop for &T {
    fn dyn_drop_ptr(self)
    {}
}
```

Notice how this shall therefore imbue with `move`/ownership semantics originally-`Copy` pointers such as `&T`. Indeed, once we go fully virtual, by virtue of being compatible/type-unified with non-`Copy` pointers such as `Box<T>` or `&mut T`, it means we have to conservatively assume any `VirtualPtr<…>` instance may have to run significant drop glue at most once, which thence makes `VirtualPtr`s not be `Copy`, even when they've originated from a `&T` reference.

- `Clone`, if any, is also to be fully `dyn`amically dispatched as well:

```
//! Pseudo-code
impl<T> DynClone for Box<T>
where
    T : Clone,
{
    fn dyn_clone_ptr(self: &Self)
      -> Self
    {
        Box::new(T::clone(&**self))
    }
}

impl<T> DynClone for Arc<T> {
    fn dyn_clone_ptr(self: &Self)
      -> Self
    {
        Arc::clone(self)
    }
}

/*
 * no `Clone` for `&mut`, obviously:
 * thus, no `From<&mut T>` for `VirtualPtr<dyn DynClone>` either.
 */

impl<T> DynClone for &'_ T {
    fn dyn_clone_ptr(self: &Self)
      -> Self
    {
        // `&T : Copy`
        *self
    }
}
```

Regarding the previous point about `&T`-originated `VirtualPtr`s not being `Copy` anymore, we can see we can get the functional API back (*i.e.*, `Clone`), if we pinky promise not to mix such `VirtualPtr`s with non-`Clone`-originating pointers (such as `&mut T`)

- ▶ Bonus: `&mut T`-reborrowing

## Related read/concept: `dyn *`

In a way, the API/functionality of `VirtualPtr` is quite similar to the very recent `dyn *` experimental[2] unstable feature of Rust.

As of this writing[2], there isn't that much proper documentation about it, and one would have to wander through Zulip discussions to know more about it, but for the following post:

---

> ***A Look at dyn* Code Generation* — by Eric Holk**

---

[2] as of 1.68.0

▸ Click here to see my own summary of `dyn *`

# `#[derive_ReprC(dyn, …)]` usage

## Summary

Given a *simple* `Trait` definition with **signatures involving `ReprC` types *exclusively***, and using `Traits` as syntax for `'usability + Trait $(+ Send)? $(+ Sync)?`, then:

1. annotating it with `#[derive_ReprC(dyn)]`

   (or `#[derive_ReprC(dyn, Clone)]`) when "`Clone`-annotating"),

   ```
   #[derive_ReprC(dyn, /* Clone */)]
   trait Trait /* : Send + Sync */ {
   ```

2. makes `dyn Traits : ReprCTrait`, which, in turn,

3. makes `VirtualPtr<dyn Traits>` become a legal/nameable type, so that:

   - `VirtualPtr<dyn Traits> : Traits`,
   - `VirtualPtr<dyn Traits> : ReprC` and thus, FFI-compatible,
   - `VirtualPtr<dyn Traits> : From<Box<impl Traits>>` and so on for the other most pervasive Rust "smart" pointer types,

- In the case of `#[derive_ReprC(dyn, Clone)]`, we'll also have:
  - `VirtualPtr<dyn Traits> : Clone`,
  - `From<{A,}Rc<impl Traits>>`,

  But at the cost of losing the `From` impls for `&mut` and `Box<impl !Clone>`.
- `From<&impl Traits>` (and `From<{A,}Rc<impl Traits>>`) will only be available when there are no `&mut self` methods in the trait definition.

### A *simple* `Trait` definition?

A `Trait` definition is deemed *simple* if:

- it only has *methods* in it ( `fn method(self: …, …)` );
- is `dyn`-safe (no generics, no `where Self : Sized` );
- with only `&Self` and `&mut Self` receiver types (the owned case is not supported yet).
  - `Pin<>`-wrapping them is, however, accepted (thereby making the `From` impls require it).

# Example: FFI-safe `Future`s and executors

By the way, all this is actually already directly featured by `safer-ffi` itself if you enable the `futures` or `tokio` features.

## `FfiFuture`

1. `Future` Trait:

```
trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>, &'_ mut Context<'_>)
      -> Poll<Self::Output>
    ;
}
```

2. *Simplify* it: `Future<Output = ()>`

```
trait SimpleFuture {
    fn dyn_poll(self: Pin<&mut Self>, &'_ mut Context<'_>)
      -> Poll<()>
    ;
}
```

- Renamed to `dyn_poll` for more readable semantics down the line.

3. Make it FFI-compatible: define an FFI-safe `Poll<()>` equivalent

```
#[derive_ReprC]
#[repr(u8)]
enum FfiPoll {
    Completed,
    Pending,
}

#[derive_ReprC(dyn)]
trait FfiFuture {
    fn dyn_poll(self: Pin<&mut Self>, &'_ mut Context<'_>)
      -> FfiPoll
    ;
}
```

- We have reached a point where `#[derive_ReprC(dyn)]` can be used!
- This means we have now gotten the `impl -> dyn` "coërcions":
  - ```
    impl<'f, F : 'f + FfiFuture> From<
        Pin<Box<F>>
    > for // ↓
        VirtualPtr<dyn 'f + FfiFuture>
    ```
  - ```
    impl<'f, F : 'f + FfiFuture> From<
        Pin<&'f mut F>
    > for // ↓
        VirtualPtr<dyn 'f + FfiFuture>
    ```

4. Convenience conversions to/from Rust Futures.
   - From:

```
impl<F : Future<Output = ()>> FfiFuture for F {
    fn dyn_poll(self: Pin<&mut Self>, ctx: &'_ mut Context<'_>)
      -> FfiPoll
    {
        match self.poll(ctx) {
            | Poll::Pending => FfiPoll::Pending,
            | Poll::Ready(()) => FfiPoll::Completed,
        }
    }
}
```

- Into:

```
impl<'fut> VirtualPtr<dyn 'fut + FfiFuture> {
    fn into_future(self)
      -> impl 'fut + Future<Output = ()>
    {
        let mut vptr = self;
        ::core::future::poll_fn(move /* vptr */ |cx| {
            match Pin::new(&mut vptr).dyn_poll(cx) {
                | FfiPoll::Pending => Poll::Pending,
                | FfiPoll::Completed => Poll::Ready(()),
            }
        })
    }
}
```

## Bonus: offering `Wake` ups to the FFI:

Remember: `Context` is an opaque FFI type, which makes it unusable there. FFI users won't be able to provide their own custom `dyn FfiFuture` objects (other than trivial `Future`s or `Future`s busy-looping the polls).

We can palliate this by exposing virtual methods / accessors specific to the opaque `Context` type:

```
#[macro_export]
macro_rules! ffi_export_future_helpers {() => (
    const _: () = {
        use $crate::⍰::std::{sync::Arc, task::Context, prelude::v1::*};

        #[ffi_export]
        fn rust_future_task_context_wake (
            task_context: &'static Context<'static>,
        )
        {
            task_context.waker().wake_by_ref()
        }

        #[ffi_export]
        fn rust_future_task_context_get_waker (
            task_context: &'static Context<'static>,
        ) -> Arc<dyn 'static + Send + Sync + Fn()>
        {
            let waker = task_context.waker().clone();
            Arc::new(move || waker.wake_by_ref()).into()
        }
    };
)}
```

## FFI-safe `Executor` / `Runtime` Handle

Note: `::tokio` does not expose an `Executor` API, but rather, a `Runtime`, which packages/bundles both the `Executor` part[1] as well as the `Reactor` part[2]. The latter is the reason some of `::tokio`'s **leaf** `Future`s (such as File I/O and `sleep`s) cannot be polled, by default, by other executors... See https://docs.rs/async-compat for more info.

[1] the `block_on()` runtime in charge of calling `.poll()` to make a future progress, with concurrent `.poll()`s to spawed `Future`s.

[2] background thread(s)/threadpool to which certain `Waker::wake()` calls are scheduled, needed by certain leaf futures.

1. A `trait` abstraction thereof:

```
trait Executor : Send + Sync {
    fn block_on<T>(
        self: &'_ Self,
        future: impl '_ + Future<Output = T>,
    ) -> T
    ;
    fn spawn(
        self: &'_ Self,
        future: impl 'static + Send + Future<Output = ()>
    ) -> Pin<Box<dyn Send + Future<Output = ()>>>
    ;
}
```

2. Making it `dyn`-safe:

```
trait Executor : Send + Sync {
    fn dyn_block_on(
        self: &'_ Self,
        future: Pin<Box<dyn '_ + Future<Output = ()>>>,
    )
    ;
    fn dyn_spawn(
        self: &'_ Self,
        future: Pin<Box<dyn Send + Future<Output = ()>>>
    ) -> Pin<Box<dyn Send + Future<Output = ()>>>
    ;
}
```

3. Making it FFI-safe:

```
#[derive_ReprC(dyn, Clone)]
trait FfiFutureExecutor : Send + Sync {
    fn dyn_block_on(
        self: &'_ Self,
        future: VirtualPtr<dyn '_ + FfiFuture>,
    )
    ;
    fn dyn_spawn(
        self: &'_ Self,
        future: VirtualPtr<dyn Send + FfiFuture>
    ) -> VirtualPtr<dyn Send + FfiFuture>
    ;
}
```

4. From a `::tokio::Handle`:

```
impl FfiFutureExecutor for ::tokio::runtime::Handle {
    fn dyn_block_on (
        self: &'_ Self,
        ffi_fut: VirtualPtr<dyn '_ + FfiFuture>,
    )
    {
        self.block_on(ffi_fut.into_future())
    }

    fn dyn_spawn (
        self: &'_ Self,
        future: VirtualPtr<dyn Send + FfiFuture>,
    ) -> VirtualPtr<dyn Send + FfiFuture>
    {
        let handle_result = self.spawn(future.into_future());
        let handle_unwrapped = async {
            handle
                .await
                .unwrap_or_else(|caught_panic| {

::std::panic::resume_unwind(caught_panic.into_panic())
                })
        };
        Box::pin(handle_unwrapped)
            .into()
    }
}
```

5. Usable as an `Executor`:

```
/// Notice how we got the generics back!
impl VirtualPtr<dyn FfiFutureExecutor> {
    fn block_on<R> (
        self: &'_ Self,
        fut: impl Future<Output = R>
    ) -> R
    {
        // We use `Option<R>` as a simple non-`'static` channel
        // to get the generic payload back.
        let mut ret = None;
        self.dyn_block_on(
            // From< Pin<&mut F> >
            ::core::pin::pin!(async {
                ret = Some(fut.await);
            })
            .into()
        );
        ret.expect("`.dyn_block_on()` did not complete")
    }

    fn spawn<R : 'static + Send> (
        self: &'_ Self,
        fut: impl 'static + Send + Future<Output = R>,
    ) -> impl 'static + Future<Output = R>
    {
        // Channel to be able to get the generic payload back.
        let (tx, rx) = ::futures::channel::oneshot::channel();
        let ffi_handle = self.dyn_spawn(
            // From< Pin<Box<F>> >
            Box::pin(async move {
                tx.send(fut.await).ok();
            })
            .into()
        );
        let handle = async move {
            ffi_handle.into_future().await;
            rx  .await
                .expect("\
                    executor dropped the `spawn`ed task \
                    before its completion\
                ")
        };
```

```
            handle
        }
    }
```

# Real-world example at Ditto-logo




# Example: our own `hashmap` in C

⚠️ Coming soon

# Appendix: A quick reminder of C compilation in Unix

Exporting / generating a C library requires *two* things:

- **the header file(s)** ( `.h` ), which contain the C signatures and thus the type (and ABI!) information of the exported functions and types.
    - Such file(s) must be `#include` d at the beginning of the C code, and are thus required to compile any C source file that *directly* calls into our Rust functions.
    - It may be necessary to tell the compiler (the C preprocessor, to be exact) the path to the folder containg the file(s), by using the `-I` flag: `-I path/to/headers/dir`
- **the object file(s)** ( `.o` ), or archive ( `.a` ) of such files (also called a *static library*), or a dynamic library ( `.so` on Linux, `.dylib` on OS X), which contain the machine code with the actual logic of such functions.
    - When linking, such file(s) must be referred to:
        - either by full path in the case of `.o` and `.a` files,
        - or, in the case of libraries ( `.a` and `.so` / `.dylib` ), when those are named as `libsome_name.extension` , by using the `-l` flag, and feeding it the parameter `some_name` ( `-l some_name` ).
            - It may be necessary to tell the compiler (the linker, to be exact) the

path to the folder containg the file(s), by using the `-L` flag: `-L path/to/libraries/dir`

- yes, there are *two* ways to refer to a static library, due to its dual nature of being both a library and a "simple" archive of raw `.o` files.

In all cases, "remember" to refer to the library object files *after* the files for your downstream binary:

```
# Incorrect
cc -L my_lib/dir -l mylib_name main.o -o main
# Correct
cc main.o -o main -L my_lib/dir -l mylib_name
```

- This is because the linker may disregard symbols that are not (yet) needed, so the callers need to come before the callees.
- In the case of a Rust-originated library, the `dl` and `pthread` libraries are very likely to be required. On Linux, they are not included by default, so, when linking, you may need to append `-lpthread -ldl` to the command for it to work.

## Static *vs.* Dynamic library

If you don't know which to use, **it is highly recommended to use a static library**. Indeed:

- Dynamic (also named *shared*) libraries are mainly a file-size optimization when having multiple downstream binaries that all depend upon the same (*shared*) library, which is quite unlikely to be the case for a Rust library.
- Dynamic libraries result in the produced program being split among multiple files (the main binary and the dynamic library), which is not only slightly less convinient than a bundled single file, but it also incurs in requiring a correct setup system-wise or binary-wise so that the dynamic library can be found at *load time*, *i.e.*, each and every time the binary is run.

  This means the the dynamic library needs to be located:

  - either in special directories such as `/usr/lib`, which may require `root` access and/or a special (`make`) `install`ation step.
    - I'd even say that this is, by the way, the main *raison d'être* for tools such as Docker: having a reliable dynamic-library setup is so painful that one

ends up scripting each and every step of the installation process to guarantee that all the tools are correcly laid out within the filesystem, and that there are no extraneous misinteractions.

    - or in a relative path ( `-Wl,-rpath,...` flag), either relative to the working directory, or relative to the location of the main binary. In both cases this may expose the user to code injection (one can easily shadow the dynamic library with their own in such cases), which, especially when the main binary has special privileges, is a security hazard.

- When *all* the libraries used by a binary are static, one gets to have a **stand-alone program**, also called "portable" (only across machines of the same architecture, though), which, contrary to "setup hell", leads to very simple "installation"s (simply copy-paste the binary, and you can run it!)

- That being said, the layer of indirection that dynamic libraries introduce can be beneficial or interesting in very special cases, which leads to some situations where releasing the library as a dynamic one is mandatory. In such cases there is no real choice, and you should be using Rust's `cdylib` 's `crate-type` to generate the shared library.

# Appendix: how does `safer_ffi` work

Most of the limitations of traditional FFI are related to the design of `cbindgen` and its being **implemented as a syntactic tool**: without access to the semantics of the code, only its representation, `cbindgen` will never truly be able to overcome these issues.

Instead, a tool for true FFI integration, including header generation, needs to **have a way to interact with the high-level code and type semantics created by the compiler**, instead of just the original source code.

There are two ways to achieve this (outside official compiler support, of course):

- either through a compiler plugin, such as `clippy` . This requires a very advanced knowledge of unstable compiler implementation details and internals, thus leading to a high maintenance burden: every new release of Rust could break such a compiler plugin (*c.f.* `clippy` -incompatible `nightly` Rust releases).

- by encoding invariants and reflection within the type system, through a complex but stable use of helper traits. **This is the choice made by** `safer_ffi` , whereby *two* traits suffice to express the necssary semantics for FFI compatibility and integration:

    - the user-facing `ReprC` trait, implemented for types having a defined C layout:

        - either directly provided by the `safer_ffi` crate (*c.f.* its dedicated

chapter),

- or implemented for custom types having the `#[derive_ReprC]` attribute.
- this is the trait that `[ffi_export]` *directly* relies on.

- the more avdanced raw `CType` trait, that you can simply dismiss as an internal trait.
  - Still, for those interested, know that it defines the necessary logic for C reflection, that you can, by the way, `unsafe` -ly implement for your custom definitions for an **advanced but doable custom extension of the framework**!