Introduction

`bindgen` **automatically generates Rust FFI bindings to C and C++ libraries.**

For example, given the C header `cool.h`:

```c
typedef struct CoolStruct {
    int x;
    int y;
} CoolStruct;

void cool_function(int i, char c, CoolStruct* cs);
```

`bindgen` produces Rust FFI code allowing you to call into the `cool` library's functions and use its types:

```rust

/* automatically generated by rust-bindgen */

#[repr(C)]
pub struct CoolStruct {
    pub x: ::std::os::raw::c_int,
    pub y: ::std::os::raw::c_int,
}

extern "C" {
    pub fn cool_function(i: ::std::os::raw::c_int,
                         c: ::std::os::raw::c_char,
                         cs: *mut CoolStruct);
}
```

Requirements

This page lists the requirements for running `bindgen` and how to get them.

Clang

`bindgen` leverages `libclang` to preprocess, parse, and type check C and C++ header files.

It is recommended to use Clang 3.9 or greater, however `bindgen` can run with older Clangs with some features disabled.

- **If you are generating bindings to C,** 3.7 and 3.8 will probably work OK for you.
- **If you are generating bindings to C++,** you almost definitely want 3.9 or greater.

Installing Clang 3.9

Windows

Download and install the official pre-built binary from LLVM download page.

You will also need to set `LIBCLANG_PATH` as an environment variable pointing to the `bin` directory of your LLVM install. For example, if you installed LLVM to `D:\programs\LLVM`, then you'd set the value to be `D:\programs\LLVM\bin`.

Alternatively, for Mingw64, you can install clang via

```
pacman -S  mingw64/mingw-w64-x86_64-clang
```

macOS

If you use Homebrew:

```
$ brew install llvm
```

If you use MacPorts:

```
$ port install clang
```

Debian-based Linuxes

```
# apt install llvm-dev libclang-dev clang
```

Ubuntu 16.10 provides the necessary packages directly. If you are using older version of Ubuntu or other Debian-based distros, you may need to add the LLVM repos to get version 3.9. See http://apt.llvm.org/.

Arch

```
# pacman -S clang
```

From source

If your package manager doesn't yet offer Clang 3.9, you'll need to build from source. For that, follow the instructions here.

Those instructions list optional steps. For `bindgen`:

- Checkout and build clang
- Checkout and build the extra-clang-tools

- You do not need to checkout or build compiler-rt
- You do not need to checkout or build libcxx

Library Usage with `build.rs`

💡 This is the recommended way to use `bindgen`. 💡

Often times C and C++ headers will have platform- and architecture-specific `#ifdef`s that affect the shape of the Rust FFI bindings we need to create to interface Rust code with the outside world. By using `bindgen` as a library inside your `build.rs`, you can generate bindings for the current target on-the-fly. Otherwise, you would need to generate and maintain `x86_64-unknown-linux-gnu-bindings.rs`, `x86_64-apple-darwin-bindings.rs`, etc... separate bindings files for each of your supported targets, which can be a huge pain. The downside is that everyone building your crate also needs `libclang` available to run `bindgen`.

Library API Documentation

📚 There is complete API reference documentation on docs.rs 📚

Tutorial

The next section contains a detailed, step-by-step tutorial for using `bindgen` as a library inside `build.rs`.

Tutorial

The following tutorial is adapted from this blog post.

What follows is a whirlwind introductory tutorial to using `bindgen` from inside `build.rs`. We'll generate bindings to `bzip2` (which is available on most systems) on-the-fly.

TL;DR? The full tutorial code is available here.

Add `bindgen` as a Build Dependency

First we need to declare a build-time dependency on `bindgen` by adding it to the `[build-dependencies]` section of our crate's `Cargo.toml` file.

Please always use the latest version of `bindgen`, it has the most fixes and best compatibility. At the time of writing the latest bindgen is `0.53.1`, but you can always check the bindgen page of crates.io to verify the latest version if you're unsure.

```toml
[build-dependencies]
bindgen = "0.53.1"
```

Create a `wrapper.h` Header

The `wrapper.h` file will include all the various headers containing declarations of structs and functions we would like bindings for. In the particular case of `bzip2`, this is pretty easy since the entire public API is contained in a single header. For a project like SpiderMonkey, where the public API is split across multiple header files and grouped by functionality, we'd want to include all those headers we want to bind to in this single `wrapper.h` entry point for `bindgen`.

Here is our `wrapper.h`:

```c
#include <bzlib.h>
```

This is also where we would add any replacement types, if we were using some.

Create a `build.rs` File

We create a `build.rs` file in our crate's root. Cargo will pick up on the existence of this file, then compile and execute it before the rest of the crate is built. This can be used to generate code at compile time. And of course in our case, we will be generating Rust FFI bindings to `bzip2` at compile time. The resulting bindings will be written to `$OUT_DIR/bindings.rs` where `$OUT_DIR` is chosen by `cargo` and is something like `./target/debug/build/bindgen-tutorial-bzip2-sys-afc7747d7eafd720/out/`.

```
extern crate bindgen;

use std::env;
use std::path::PathBuf;

fn main() {
    // Tell cargo to tell rustc to link the system bzip2
    // shared library.
    println!("cargo:rustc-link-lib=bz2");

    // Tell cargo to invalidate the built crate whenever the wrapper changes
    println!("cargo:rerun-if-changed=wrapper.h");

    // The bindgen::Builder is the main entry point
    // to bindgen, and lets you build up options for
    // the resulting bindings.
    let bindings = bindgen::Builder::default()
        // The input header we would like to generate
        // bindings for.
        .header("wrapper.h")
        // Tell cargo to invalidate the built crate whenever any of the
        // included header files changed.
        .parse_callbacks(Box::new(bindgen::CargoCallbacks))
        // Finish the builder and generate the bindings.
        .generate()
        // Unwrap the Result and panic on failure.
        .expect("Unable to generate bindings");

    // Write the bindings to the $OUT_DIR/bindings.rs file.
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}
```

Now, when we run `cargo build`, our bindings to `bzip2` are generated on the fly!

There's more info about `build.rs` files in the crates.io documentation.

Include the Generated Bindings in `src/lib.rs`

We can use the `include!` macro to dump our generated bindings right into our crate's main entry point, `src/lib.rs`:

```
#![allow(non_upper_case_globals)]
#![allow(non_camel_case_types)]
#![allow(non_snake_case)]

include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

Because `bzip2`'s symbols do not follow Rust's style conventions, we suppress a bunch of warnings with a few `#![allow(...)]` pragmas.

We can run `cargo build` again to check that the bindings themselves compile:

```
$ cargo build
   Compiling bindgen-tutorial-bzip2-sys v0.1.0
    Finished debug [unoptimized + debuginfo] target(s) in 62.8 secs
```

And we can run `cargo test` to verify that the layout, size, and alignment of our generated Rust FFI structs match what `bindgen` thinks they should be:

```
$ cargo test
   Compiling bindgen-tutorial-bzip2-sys v0.1.0
    Finished debug [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/deps/bzip2_sys-10413fc2af207810

running 14 tests
test bindgen_test_layout___darwin_pthread_handler_rec ... ok
test bindgen_test_layout___sFILE ... ok
test bindgen_test_layout___sbuf ... ok
test bindgen_test_layout__bindgen_ty_1 ... ok
test bindgen_test_layout__bindgen_ty_2 ... ok
test bindgen_test_layout__opaque_pthread_attr_t ... ok
test bindgen_test_layout__opaque_pthread_cond_t ... ok
test bindgen_test_layout__opaque_pthread_mutex_t ... ok
test bindgen_test_layout__opaque_pthread_condattr_t ... ok
test bindgen_test_layout__opaque_pthread_mutexattr_t ... ok
test bindgen_test_layout__opaque_pthread_once_t ... ok
test bindgen_test_layout__opaque_pthread_rwlock_t ... ok
test bindgen_test_layout__opaque_pthread_rwlockattr_t ... ok
test bindgen_test_layout__opaque_pthread_t ... ok

test result: ok. 14 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests bindgen-tutorial-bzip2-sys

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Write a Sanity Test

Finally, to tie everything together, let's write a sanity test that round trips some text through compression and decompression, and then asserts that it came back out the same as it went in. This is a little wordy using the raw FFI bindings, but hopefully we wouldn't usually ask people to do this, we'd provide a nice Rust-y API on top of the raw FFI bindings for them. However, since this is for testing the bindings directly, our sanity test will use the bindings directly.

The test data I'm round tripping are some Futurama quotes I got off the internet and put in the `futurama-quotes.txt` file, which is read into a `&'static str` at compile time via the `include_str!("../futurama-quotes.txt")` macro invocation.

Without further ado, here is the test, which should be appended to the bottom of our `src/lib.rs` file:

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::mem;

    #[test]
    fn round_trip_compression_decompression() {
        unsafe {
            let input = include_str!("../futurama-quotes.txt").as_bytes();
            let mut compressed_output: Vec<u8> = vec![0; input.len()];
            let mut decompressed_output: Vec<u8> = vec![0; input.len()];

            // Construct a compression stream.
            let mut stream: bz_stream = mem::zeroed();
            let result = BZ2_bzCompressInit(&mut stream as *mut _,
                                            1,   // 1 x 100000 block size
                                            4,   // verbosity (4 = most verbose)
                                            0);  // default work factor
            match result {
                r if r == (BZ_CONFIG_ERROR as _) => panic!("BZ_CONFIG_ERROR"),
                r if r == (BZ_PARAM_ERROR as _) => panic!("BZ_PARAM_ERROR"),
                r if r == (BZ_MEM_ERROR as _) => panic!("BZ_MEM_ERROR"),
                r if r == (BZ_OK as _) => {},
                r => panic!("Unknown return value = {}", r),
            }

            // Compress `input` into `compressed_output`.
            stream.next_in = input.as_ptr() as *mut _;
            stream.avail_in = input.len() as _;
            stream.next_out = compressed_output.as_mut_ptr() as *mut _;
            stream.avail_out = compressed_output.len() as _;
            let result = BZ2_bzCompress(&mut stream as *mut _, BZ_FINISH as _);
            match result {
                r if r == (BZ_RUN_OK as _) => panic!("BZ_RUN_OK"),
                r if r == (BZ_FLUSH_OK as _) => panic!("BZ_FLUSH_OK"),
                r if r == (BZ_FINISH_OK as _) => panic!("BZ_FINISH_OK"),
                r if r == (BZ_SEQUENCE_ERROR as _) => panic!
("BZ_SEQUENCE_ERROR"),
                r if r == (BZ_STREAM_END as _) => {},
                r => panic!("Unknown return value = {}", r),
            }

            // Finish the compression stream.
            let result = BZ2_bzCompressEnd(&mut stream as *mut _);
            match result {
                r if r == (BZ_PARAM_ERROR as _) => panic!(BZ_PARAM_ERROR),
                r if r == (BZ_OK as _) => {},
                r => panic!("Unknown return value = {}", r),
            }

            // Construct a decompression stream.
            let mut stream: bz_stream = mem::zeroed();
            let result = BZ2_bzDecompressInit(&mut stream as *mut _,
                                              4,   // verbosity (4 = most
verbose)
                                              0);  // default small factor
            match result {
                r if r == (BZ_CONFIG_ERROR as _) => panic!("BZ_CONFIG_ERROR"),
                r if r == (BZ_PARAM_ERROR as _) => panic!("BZ_PARAM_ERROR"),
                r if r == (BZ_MEM_ERROR as _) => panic!("BZ_MEM_ERROR"),
                r if r == (BZ_OK as _) => {},
                r => panic!("Unknown return value = {}", r),
            }

            // Decompress `compressed_output` into `decompressed_output`.
            stream.next_in = compressed_output.as_ptr() as *mut _;
            stream.avail_in = compressed_output.len() as _;
            stream.next_out = decompressed_output.as_mut_ptr() as *mut _;
            stream.avail_out = decompressed_output.len() as _;
            let result = BZ2_bzDecompress(&mut stream as *mut _);
            match result {
                r if r == (BZ_PARAM_ERROR as _) => panic!("BZ_PARAM_ERROR"),
                r if r == (BZ_DATA_ERROR as _) => panic!("BZ_DATA_ERROR"),
                r if r == (BZ_DATA_ERROR_MAGIC as _) => panic!("BZ_DATA_ERROR"),
                r if r == (BZ_MEM_ERROR as _) => panic!("BZ_MEM_ERROR"),
                r if r == (BZ_OK as _) => panic!("BZ_OK"),
                r if r == (BZ_STREAM_END as _) => {},
                r => panic!("Unknown return value = {}", r),
            }

            // Close the decompression stream.
            let result = BZ2_bzDecompressEnd(&mut stream as *mut _);
            match result {
                r if r == (BZ_PARAM_ERROR as _) => panic!("BZ_PARAM_ERROR"),
                r if r == (BZ_OK as _) => {},
                r => panic!("Unknown return value = {}", r),
            }

            assert_eq!(input, &decompressed_output[..]);
        }
    }
}
```

Now let's run `cargo test` again and verify that everything is linking and binding properly!

```
$ cargo test
   Compiling bindgen-tutorial-bzip2-sys v0.1.0
    Finished debug [unoptimized + debuginfo] target(s) in 0.54 secs
     Running target/debug/deps/bindgen_tutorial_bzip2_sys-1c5626bbc4401c3a

running 15 tests
test bindgen_test_layout___darwin_pthread_handler_rec ... ok
test bindgen_test_layout___sFILE ... ok
test bindgen_test_layout___sbuf ... ok
test bindgen_test_layout__bindgen_ty_1 ... ok
test bindgen_test_layout__bindgen_ty_2 ... ok
test bindgen_test_layout__opaque_pthread_attr_t ... ok
test bindgen_test_layout__opaque_pthread_cond_t ... ok
test bindgen_test_layout__opaque_pthread_condattr_t ... ok
test bindgen_test_layout__opaque_pthread_mutex_t ... ok
test bindgen_test_layout__opaque_pthread_mutexattr_t ... ok
test bindgen_test_layout__opaque_pthread_once_t ... ok
test bindgen_test_layout__opaque_pthread_rwlock_t ... ok
test bindgen_test_layout__opaque_pthread_rwlockattr_t ... ok
test bindgen_test_layout__opaque_pthread_t ... ok
    block 1: crc = 0x47bfca17, combined CRC = 0x47bfca17, size = 2857
        bucket sorting ...
        depth      1 has   2849 unresolved strings
        depth      2 has   2702 unresolved strings
        depth      4 has   1508 unresolved strings
        depth      8 has    538 unresolved strings
        depth     16 has    148 unresolved strings
        depth     32 has      0 unresolved strings
        reconstructing block ...
      2857 in block, 2221 after MTF & 1-2 coding, 61+2 syms in use
      initial group 5, [0 .. 1], has 570 syms (25.7%)
      initial group 4, [2 .. 2], has 256 syms (11.5%)
      initial group 3, [3 .. 6], has 554 syms (24.9%)
      initial group 2, [7 .. 12], has 372 syms (16.7%)
      initial group 1, [13 .. 62], has 469 syms (21.1%)
      pass 1: size is 2743, grp uses are 13 6 15 0 11
      pass 2: size is 1216, grp uses are 13 7 15 0 10
      pass 3: size is 1214, grp uses are 13 8 14 0 10
      pass 4: size is 1213, grp uses are 13 9 13 0 10
      bytes: mapping 19, selectors 17, code lengths 79, codes 1213
    final combined CRC = 0x47bfca17

    [1: huff+mtf rt+rld {0x47bfca17, 0x47bfca17}]
    combined CRCs: stored = 0x47bfca17, computed = 0x47bfca17
test tests::round_trip_compression_decompression ... ok

test result: ok. 15 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests bindgen-tutorial-bzip2-sys

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Publish Your Crate!

That's it! Now we can publish our crate on crates.io and we can write a nice, Rust-y API wrapping the raw FFI bindings in a safe interface. However, there is already a `bzip2-sys` crate providing raw FFI bindings, and there is already a `bzip2` crate providing a nice, safe, Rust-y API on top of the bindings, so we have nothing left to do here!

Check out the full code on Github!

Command Line Usage

Install the `bindgen` executable with `cargo`:

```
$ cargo install bindgen
```

The `bindgen` executable is installed to `~/.cargo/bin`. You have to add that directory to your `$PATH` to use `bindgen`.

`bindgen` takes the path to an input C or C++ header file, and optionally an output file path for the generated bindings. If the output file path is not supplied, the bindings are printed to `stdout`.

If we wanted to generated Rust FFI bindings from a C header named `input.h` and put them in the `bindings.rs` file, we would invoke `bindgen` like this:

```
$ bindgen input.h -o bindings.rs
```

For more details, pass the `--help` flag:

```
$ bindgen --help
```

Customizing the Generated Bindings

The translation of classes, structs, enums, and typedefs can be adjusted in a few ways:

1. By using the `bindgen::Builder`'s configuration methods, when using `bindgen` as a library.

2. By passing extra flags and options to the `bindgen` executable.

3. By adding an annotation comment to the C/C++ source code. Annotations are specially formatted HTML tags inside doxygen style comments:

   - For single line comments:

     ```
     /// <div rustbindgen></div>
     ```

   - For multi-line comments:

```
/**
 * <div rustbindgen></div>
 */
```

We'll leave the nitty-gritty details to the docs.rs API reference and `bindgen --help`, but provide higher level concept documentation here.

Whitelisting

Whitelisting allows us to be precise about which type, function, and global variable definitions `bindgen` generates bindings for. By default, if we don't specify any whitelisting rules, everything is considered whitelisted. This may not be desirable because of either

- the generated bindings contain a lot of extra definitions we don't plan on using, or
- the header file contains C++ features for which Rust does not have a corresponding form (such as partial template specialization), and we would like to avoid these definitions

If we specify whitelisting rules, then `bindgen` will only generate bindings to types, functions, and global variables that match the whitelisting rules, or are transitively used by a definition that matches them.

Library

- `bindgen::Builder::whitelist_type`
- `bindgen::Builder::whitelist_function`
- `bindgen::Builder::whitelist_var`

Command Line

- `--whitelist-type <type>`
- `--whitelist-function <function>`
- `--whitelist-var <var>`

Annotations

None.

Blacklisting

If you need to provide your own custom translation of some type (for example, because you need to wrap one of its fields in an `UnsafeCell`), you can explicitly blacklist generation of its definition. Uses of the blacklisted type will still appear in other types' definitions. (If you don't want the type to appear in the bindings at all, make it opaque instead of blacklisting it.)

Blacklisted types are pessimistically assumed not to be able to `derive` any traits, which can transitively affect other types' ability to `derive` traits or not.

Library

- `bindgen::Builder::blacklist_type`

Command Line

- `--blacklist-type <type>`

Annotations

```
/// <div rustbindgen hide></div>
class Foo {
    // ...
};
```

Treating a Type as an Opaque Blob of Bytes

Sometimes a type definition is simply not translatable to Rust, for example it uses C++'s SFINAE for which Rust has no equivalent. In these cases, it is best to treat all occurrences of the type as an opaque blob of bytes with a size and alignment. `bindgen` will attempt to detect such cases and do this automatically, but other times it needs some explicit help from you.

Library

- `bindgen::Builder::opaque_type`

Command Line

- `--opaque-type <type>`

Annotation

```
/// <div rustbindgen opaque></div>
class Foo {
    // ...
};
```

Replacing One Type with Another

The `replaces` annotation can be used to use a type as a replacement for other (presumably more complex) type. This is used in Stylo to generate bindings for structures that for multiple reasons are too complex for bindgen to understand.

For example, in a C++ header:

```
/**
 * <div rustbindgen replaces="nsTArray"></div>
 */
template<typename T>
class nsTArray_Simple {
  T* mBuffer;
public:
  // The existence of a destructor here prevents bindgen from deriving the Clone
  // trait via a simple memory copy.
  ~nsTArray_Simple() {};
};
```

That way, after code generation, the bindings for the `nsTArray` type are the ones that would be generated for `nsTArray_Simple`.

Replacing is only available as an annotation. To replace a C or C++ definition with a Rust definition, use blacklisting.

Preventing the Derivation of `Copy` and `Clone`

`bindgen` will attempt to derive the `Copy` and `Clone` traits on a best-effort basis. Sometimes, it might not understand that although adding `#[derive(Copy, Clone)]` to a translated type definition will compile, it still shouldn't do that for reasons it can't know. In these cases, the `nocopy` annotation can be used to prevent bindgen to autoderive the `Copy` and `Clone` traits for a type.

Library

- `bindgen::Builder::no_copy`

Command Line

- `--no-copy <regex>`

Annotations

```
/**
 * Although bindgen can't know, this struct is not safe to move because pthread
 * mutexes can't move in memory!
 *
 * <div rustbindgen nocopy></div>
 */
struct MyMutexWrapper {
    pthread_mutex_t raw;
    // ...
};
```

Generating Bindings to C++

`bindgen` can handle some C++ features, but not all of them. To set expectations: `bindgen` will give you the type definitions and FFI declarations you need to build an API to the C++ library, but using those types in Rust will be nowhere near as nice as using them in C++. You will have to manually call constructors, destructors, overloaded operators, etc yourself.

When passing in header files, the file will automatically be treated as C++ if it ends in `.hpp`. If it doesn't, adding `-x c++` clang args can be used to force C++ mode. You probably also want to use `-std=c++14` or similar clang args as well.

You pretty much **must** use whitelisting when working with C++ to avoid pulling in all of the `std::*` types, many of which `bindgen` cannot handle. Additionally, you may want to mark other types as opaque that `bindgen` stumbles on. It is recommended to mark all of `std::*` opaque, and to whitelist only precisely the functions and types you intend to use.

You should read up on the FAQs as well.

Supported Features

- Inheritance (for the most part; there are some outstanding bugs)
- Methods
- Bindings to constructors and destructors (but they aren't implicitly or automatically invoked)
- Function and method overloading

- Templates *without* specialization. You should be able to access individual fields of the class or struct.

Unsupported Features

When `bindgen` finds a type that is too difficult or impossible to translate into Rust, it will automatically treat it as an opaque blob of bytes. The philosophy is that

1. we should always get layout, size, and alignment correct, and

2. just because one type uses specialization, that shouldn't cause `bindgen` to give up on everything else.

Without further ado, here are C++ features that `bindgen` does not support or cannot translate into Rust:

- Inline functions and methods: see "Why isn't `bindgen` generating bindings to inline functions?"

- Template functions, methods of template classes and structs. We don't know which monomorphizations exist, and can't create new ones because we aren't a C++ compiler.

- Anything related to template specialization:

    - Partial template specialization
    - Traits templates
    - Substitution Failure Is Not An Error (SFINAE)

- Cross language inheritance, for example inheriting from a Rust struct in C++.

- Automatically calling copy and/or move constructors or destructors. Supporting this isn't possible with Rust's move semantics.

- Exceptions: if a function called through a `bindgen`-generated interface raises an exception that is not caught by the function itself, this will generate undefined behaviour. See the tracking issue for exceptions for more details.

Generating Bindings to Objective-C

`bindgen` does not (yet) have full objective-c support but it can generate bindings for a lot of the apple frameworks without too much blacklisting.

In order to generate bindings, you will need `-x objective-c` as the clang args. If you'd like to use block you will need `-fblocks` as a clang arg as well.

Depending on your setup, you may need `--generate-block` to generate the block function aliases and `--block-extern-crate` to insert a `extern crate block` at the beginning of the generated bindings. The same logic applies to the `--objc-extern-crate` parameter.

The objective-c classes will be represented as a `struct Foo(id)` and a trait `IFoo` where `Foo` is the objective-c class and `id` is an alias for `*mut objc::runtime::Object` (the pointer to the objective-c instance). The trait `IFoo` is needed to allow for the generated inheritance.

Each class (struct) has an `alloc` and a `dealloc` to match that of some of the alloc methods found in `NSObject`.

In order to initialize a class `Foo`, you will have to do something like `let foo = Foo(Foo::alloc().initWithStuff())`.

Supported Features

- Inheritance matched to rust traits with prefixes of `I` which stands for interface.
- Protocols which match to rust traits with prefixes of `P` which stands for Protocol.
- Classes will generate `struct Foo(id)` where `Foo` is the class name and `id` is a pointer to the objective-c Object.
- Blocks

Useful Notes

- If you're targeting `aarch64-apple-ios`, you'll need to have the clang arg `--target=arm64-apple-ios` as mentioned here.
- The generated bindings will almost certainly have some conflicts so you will have to blacklist a few things. There are a few cases of the parameters being poorly named in the objective-c headers. But if you're using anything with Core Foundation, you'll find that `time.h` as has a variable called timezone that conflicts with some of the things in `NSCalendar.h`.
- Some small subset of the function headers in the apple frameworks go against apple's guidelines for parameter names and duplicate the names in the header which won't compile as mentioned here.
- instancetype return methods does not return `Self` for you given class, it returns a `mut * objc::runtime::Objc` which is aliased as `id`. This is because objective-c's inheritance doesn't perfectly match that of rusts.
- Depending on what you're trying `bindgen` against, you may end up including all of Core Foundation and any other frameworks. This will result in a very long compile time.

Not (yet) Supported

- Nullablibility attributes which return `Option`s.
- Probably many other things. Feel free to open an issue.

Example crate(s)

- uikit-sys

Using the Union Types Generated by Bindgen

NOTE: Rust 1.19 stabilized the `union` type (see Rust issue #32836).

You can pass the `--rust-target` option to tell `bindgen` to target a specific version of Rust. By default, `bindgen` will target the latest stable Rust. The `--rust-target` option accepts a specific stable version (such as "1.0" or "1.19") or "nightly".

NOTE: The `--unstable-rust` option is deprecated; use `--rust-target nightly` instead.

In general, most interactions with unions (either reading or writing) are unsafe, meaning you must surround union accesses in an `unsafe {}` block.

For this discussion, we will use the following C type definitions:

```c
typedef struct {
    int32_t a;
    int32_t b;
} alpha_t;

typedef struct {
    uint32_t c;
    uint16_t d;
    uint16_t e;
    uint8_t  f;
} beta_t;

typedef union {
    alpha_t alfa;
    beta_t  bravo;
} greek_t;
```

Relevant Bindgen Options

Library

- `bindgen::Builder::rust_target()`
- `bindgen::Builder::derive_default()`

Command Line

- `--rust-target`
- `--with-derive-default`

Which union type will Bindgen generate?

Bindgen can emit one of two Rust types that correspond to C unions:

- Rust's `union` builtin (only available in Rust >= 1.19, including nightly)
- Bindgen's `BindgenUnion` (available for all Rust targets)

Bindgen uses the following logic to determine which Rust union type to emit:

- If the Rust target is >= 1.19 (including nightly) AND each field of the union can derive `Copy`, then generate a `union` builtin.
- Otherwise, generate a `BindgenUnion`.

Using the `union` builtin

When using the `union` builtin type, there are two choices for initialization:

1. Zero
2. With a specific variant

```rust
mod bindings_builtin_union;

fn union_builtin() {
    // Initalize the union to zero
    let x = bindings_builtin_union::greek_t::default();

    // If `--with-derive-default` option is not used, the following may be used
    //   to initalize the union to zero:
    let x = unsafe { std::mem::zeroed::<bindings_builtin_union::greek_t>() };

    // Or, it is possible to initialize exactly one variant of the enum:
    let x = bindings_builtin_union::greek_t {
        alfa: bindings_builtin_union::alpha_t {
            a: 1,
            b: -1,
        },
    };

    unsafe {
        println!("{:?}", z.alfa);  // alpha_t { a: 1, b: -1 }
        println!("{:?}", z.bravo); // beta_t { c: 1, d: 65535, e: 65535, f: 127
}
    }
}
```

Using the `BindgenUnion` type

If the target Rust version does not support the new `union` type or there is a field that cannot derive `Copy`, then bindgen will provide union-like access to a `struct`.

Interacting with these unions is slightly different than the new `union` types. You must access union variants through a reference.

```
mod bindings;

fn bindgenunion() {
    // `default()` or `zeroed()` may still be used with Bindgen's Union types
    let mut x = bindings::greek_t::default();

    // This will not work:
    // let x = bindings::greek_t {
    //     alfa: bindings::alpha_t {
    //         a: 1,
    //         b: -1,
    //     },
    // };

    // Instead, access the field through `.as_ref()` and `.as_mut()` helpers:
    unsafe {
        *x.alfa.as_mut() = bindings::alpha_t {
            a: 1,
            b: -1,
        };

        println!("{:?}", x.alfa.as_ref());  // alpha_t { a: 1, b: -1 }
        println!("{:?}", x.bravo.as_ref()); // beta_t { c: 1, d: 65535, e:
65535, f: 0 }
    }
```

If you attempt to access a `BindgenUnion` field directly, you will see errors like this:

```
error[E0308]: mismatched types
  --> src/main.rs:44:15
   |
44 |           alfa: bindings::alpha_t {
   |  ______________________^
45 | |             a: 1,
46 | |             b: -1,
47 | |         },
   | |_________^ expected struct `bindings::__BindgenUnionField`, found struct
`bindings::alpha_t`
   |
   = note: expected type `bindings::__BindgenUnionField<bindings::alpha_t>`
              found type `bindings::alpha_t`
```

Using the Bitfield Types Generated by Bindgen

Bitfield Strategy Overview

As Rust does not support bitfields, Bindgen generates a struct for each with the following characteristics

- Immutable getter functions for each bitfield named `<bitfield>`
- Setter functions for each contiguous block of bitfields named `set_<bitfield>`
- Far each contiguous block of bitfields, Bindgen emits an opaque physical field that contains one or more logical bitfields
- A static constructor `new_bitfield_{1, 2, ...}` with a parameter for each bitfield contained within the opaque physical field.

Bitfield examples

For this discussion, we will use the following C type definitions and functions.

```
typedef struct {
    unsigned int a: 1;
    unsigned int b: 1;
    unsigned int c: 2;

} StructWithBitfields;

// Create a default bitfield
StructWithBitfields create_bitfield();

// Print a bitfield
void print_bitfield(StructWithBitfields bfield);
```

Bindgen creates a set of field getters and setters for interacting with the bitset. For example,

```
    let mut bfield = unsafe { create_bitfield() };

    bfield.set_a(1);
    println!("a set to {}", bfield.a());
    bfield.set_b(1);
    println!("b set to {}", bfield.b());
    bfield.set_c(3);
    println!("c set to {}", bfield.c());

    unsafe { print_bitfield(bfield) };
```

will print out

```
a set to 1
b set to 1
c set to 3
StructWithBitfields: a:1, b:1, c:3
```

Overflowing a bitfield will result in the same behavior as in C/C++: the bitfield will be set to 0.

```rust
    let mut bfield = unsafe { create_bitfield() };
    bfield.set_a(1);
    bfield.set_b(1);
    bfield.set_c(12);
    println!("c set to {} due to overflow", bfield.c());

    unsafe { print_bitfield(bfield) };
```

will print out

```
c set to 0 due to overflow
StructWithBitfields: a:1, b:1, c:0
```

To create a new bitfield in Rust, use the bitfield allocation unit constructor.

Note: This requires the Builder's derive_default to be set to true, otherwise the necessary Default functions won't be generated.

```rust
    let bfield = StructWithBitfields{
        _bitfield_1: StructWithBitfields::new_bitfield_1(0,0,0),
        ..Default::default()
    };

    unsafe { print_bitfield(bfield) };
```

This will print out

```
StructWithBitfields: a:1, b:0, c:2
```

Frequently Asked Questions

- Why isn't `bindgen` generating methods for this whitelisted class?
- Why isn't `bindgen` generating bindings to inline functions?
- Does `bindgen` support the C++ Standard Template Library (STL)?

Why isn't `bindgen` generating methods for this whitelisted class?

Are the methods `inline` methods, or defined inline in the class? For example:

```
class Dooder {
  public:
    // Function defined inline in the class.
    int example_one() { return 1; }

    // `inline` function whose definition is supplied later in the header, or in
    // another header.
    inline bool example_two();
};

inline bool Dooder::example_two() {
    return true;
}
```

If so, see "Why isn't `bindgen` generating bindings to inline functions?"

If not, consider filing an issue!

Why isn't `bindgen` generating bindings to inline functions?

These functions don't typically end up in object files or shared libraries with symbols that we can reliably link to, since they are instead inlined into each of their call sites. Therefore, we don't generate bindings to them, since that creates linking errors.

However, if you are compiling the C/C++ yourself (rather than using a system shared library, for example), then you can pass `-fkeep-inline-functions` or `-fno-inline-functions` to `gcc` or `clang`, and invoke `bindgen` with either the `bindgen::Builder::generate_inline_functions` method or the `--generate-inline-functions` flag.

Note that these functions and methods are usually marked inline for a reason: they tend to be hot. The above workaround makes them an out-of-line call, which might not provide acceptable performance.

Does `bindgen` support the C++ Standard Template Library (STL)?

Sort of. A little. Depends what you mean by "support".

Most functions, methods, constructors, and destructors are inline in the STL. That ties our hands when it comes to linking: "Why isn't `bindgen` generating bindings to inline functions?"

As far as generating opaque blobs of bytes with the correct size and alignment, `bindgen` can do pretty well. This is typically enough to let you use types that transitively contain STL things. We generally recommend marking `std::.*` as opaque, and then whitelisting only the specific things you need from the library you're binding to that is pulling in STL headers.

How to deal with bindgen generated padding fields?

Depending the architecture, toolchain versions and source struct, it is possible that bindgen will generate padding fields named `__bindgen_padding_N`. As these fields might be present when compiling for one architecture but not for an other, you should not initialize these fields manually when initializing the struct. Instead, use the `Default` trait. You can either enable this when constructing the `Builder` using the `derive_default` method, or you can implement this per struct using:

```
impl Default for SRC_DATA {
    fn default() -> Self {
        unsafe { std::mem::zeroed() }
    }
}
```

This makes it possible to initialize `SRC_DATA` by:

```
SRC_DATA {
    field_a: "foo",
    field_b: "bar",
    ..Default::default()
}
```

In the case bindgen generates a padding field, then this field will be automatically initialized by `..Default::default()`.