Rust Foreign Function Interface Tutorial

1 Introduction

This tutorial will use the snappy compression/decompression library as an introduction to writing bindings for foreign code. Rust is currently unable to call directly into a C++ library, but snappy includes a C interface (documented in `snappy-c.h`).

The following is a minimal example of calling a foreign function which will compile if snappy is installed:

```
use std::libc::size_t;

#[link_args = "-lsnappy"]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println(fmt!("max compressed length of a 100 byte buffer: %?", x));
}
```

The `extern` block is a list of function signatures in a foreign library, in this case with the platform's C ABI. The `#[link_args]` attribute is used to instruct the linker to link against the snappy library so the symbols are resolved.

Foreign functions are assumed to be unsafe so calls to them need to be wrapped with `unsafe {}` as a promise to the compiler that everything contained within truly is safe. C libraries often expose interfaces that aren't thread-safe, and almost any function that takes a pointer argument isn't valid for all possible inputs since the pointer could be dangling, and raw pointers fall outside of Rust's safe memory model.

When declaring the argument types to a foreign function, the Rust compiler will not check if the declaration is correct, so specifying it correctly is part of keeping the binding correct at runtime.

The `extern` block can be extended to cover the entire snappy API:

```
use std::libc::{c_int, size_t};

#[link_args = "-lsnappy"]
extern {
    fn snappy_compress(input: *u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *u8,
                                         compressed_length: size_t) -> c_int;
}
```

2 Creating a safe interface

The raw C API needs to be wrapped to provide memory safety and make use of higher-level concepts like vectors. A library can choose to expose only the safe, high-level interface and hide the unsafe internal details.

Wrapping the functions which expect buffers involves using the `vec::raw` module to manipulate Rust vectors as pointers to memory. Rust's vectors are guaranteed to be a contiguous block of memory. The length is number of elements currently contained, and the capacity is the total size in elements of the allocated memory. The length is less than or equal to the capacity.

```
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(vec::raw::to_ptr(src), src.len() as size_t) == 0
    }
}
```

The `validate_compressed_buffer` wrapper above makes use of an `unsafe` block, but it makes the guarantee that calling it is safe for all inputs by leaving off `unsafe` from the function signature.

The `snappy_compress` and `snappy_uncompress` functions are more complex, since a buffer has to be allocated to hold the output too.

The `snappy_max_compressed_length` function can be used to allocate a vector with the maximum required capacity to hold the compressed output. The vector can then be passed to the

`snappy_compress` function as an output parameter. An output parameter is also passed to retrieve the true length after compression for setting the length.

```
pub fn compress(src: &[u8]) -> ~[u8] {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = vec::raw::to_ptr(src);

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = vec::with_capacity(dstlen as uint);
        let pdst = vec::raw::to_mut_ptr(dst);

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        vec::raw::set_len(&mut dst, dstlen as uint);
        dst
    }
}
```

Decompression is similar, because snappy stores the uncompressed size as part of the compression format and `snappy_uncompressed_length` will retrieve the exact buffer size required.

```
pub fn uncompress(src: &[u8]) -> Option<~[u8]> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = vec::raw::to_ptr(src);

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = vec::with_capacity(dstlen as uint);
        let pdst = vec::raw::to_mut_ptr(dst);

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            vec::raw::set_len(&mut dst, dstlen as uint);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

For reference, the examples used here are also available as an library on GitHub.

3 Destructors

Rust 0.8-pre

fbeeeebf

Foreign libraries often hand off ownership of resources to the calling code, which should be wrapped

in a destructor to provide safety and guarantee their release.

A type with the same functionality as owned boxes can be implemented by wrapping `malloc` and `free`:

```
use std::cast;
use std::libc::{c_void, size_t, malloc, free};
use std::ptr;
use std::unstable::intrinsics;

// a wrapper around the handle returned by the foreign code
pub struct Unique<T> {
    priv ptr: *mut T
}

impl<T: Send> Unique<T> {
    pub fn new(value: T) -> Unique<T> {
        unsafe {
            let ptr = malloc(std::sys::size_of::<T>() as size_t) as *mut T;
            assert!(!ptr::is_null(ptr));
            // `*ptr` is uninitialized, and `*ptr = value` would attempt to destroy it
            intrinsics::move_val_init(&mut *ptr, value);
            Unique{ptr: ptr}
        }
    }

    // the 'r lifetime results in the same semantics as `&*x` with ~T
    pub fn borrow<'r>(&'r self) -> &'r T {
        unsafe { cast::copy_lifetime(self, &*self.ptr) }
    }

    // the 'r lifetime results in the same semantics as `&mut *x` with ~T
    pub fn borrow_mut<'r>(&'r mut self) -> &'r mut T {
        unsafe { cast::copy_mut_lifetime(self, &mut *self.ptr) }
    }
}

#[unsafe_destructor]
impl<T: Send> Drop for Unique<T> {
    fn drop(&self) {
        unsafe {
            let x = intrinsics::init(); // dummy value to swap in
            // moving the object out is needed to call the destructor
            ptr::replace_ptr(self.ptr, x);
            free(self.ptr as *c_void)
        }
    }
```

Rust 0.8-pre

fbeeeebf

```
}

// A comparison between the built-in ~ and this reimplementation
fn main() {
    {
        let mut x = ~5;
        *x = 10;
    } // `x` is freed here

    {
        let mut y = Unique::new(5);
        *y.borrow_mut() = 10;
    } // `y` is freed here
}
```

4 Linking

In addition to the `#[link_args]` attribute for explicitly passing arguments to the linker, an `extern mod` block will pass `-lmodname` to the linker by default unless it has a `#[nolink]` attribute applied.

5 Unsafe blocks

Some operations, like dereferencing unsafe pointers or calling functions that have been marked unsafe are only allowed inside unsafe blocks. Unsafe blocks isolate unsafety and are a promise to the compiler that the unsafety does not leak out of the block.

Unsafe functions, on the other hand, advertise it to the world. An unsafe function is written like this:

```
unsafe fn kaboom(ptr: *int) -> int { *ptr }
```

This function can only be called from an `unsafe` block or another `unsafe` function.

6 Foreign calling conventions

Most foreign code exposes a C ABI, and Rust uses the platform's C calling convention by default when calling foreign functions. Some foreign functions, most notably the Windows API, use other calling conventions. Rust provides the `abi` attribute as a way to hint to the compiler which calling convention to use:

```
#[cfg(target_os = "win32")]
#[abi = "stdcall"]
#[link_name = "kernel32"]
```

Rust 0.8-pre

fbeeeebf

```
extern {
    fn SetEnvironmentVariableA(n: *u8, v: *u8) -> int;
}
```

The `abi` attribute applies to a foreign module (it cannot be applied to a single function within a module), and must be either `"cdecl"` or `"stdcall"`. The compiler may eventually support other calling conventions.

7 Interoperability with foreign code

Rust guarantees that the layout of a `struct` is compatible with the platform's representation in C. A `#[packed]` attribute is available, which will lay out the struct members without padding. However, there are currently no guarantees about the layout of an `enum`.

Rust's owned and managed boxes use non-nullable pointers as handles which point to the contained object. However, they should not be manually created because they are managed by internal allocators. Borrowed pointers can safely be assumed to be non-nullable pointers directly to the type. However, breaking the borrow checking or mutability rules is not guaranteed to be safe, so prefer using raw pointers (*) if that's needed because the compiler can't make as many assumptions about them.

Vectors and strings share the same basic memory layout, and utilities are available in the `vec` and `str` modules for working with C APIs. Strings are terminated with `\0` for interoperability with C, but it should not be assumed because a slice will not always be nul-terminated. Instead, the `str::as_c_str` function should be used.

The standard library includes type aliases and function definitions for the C standard library in the `libc` module, and Rust links against `libc` and `libm` by default.

Rust 0.8-pre

fbeeeebf