Foreign function interface

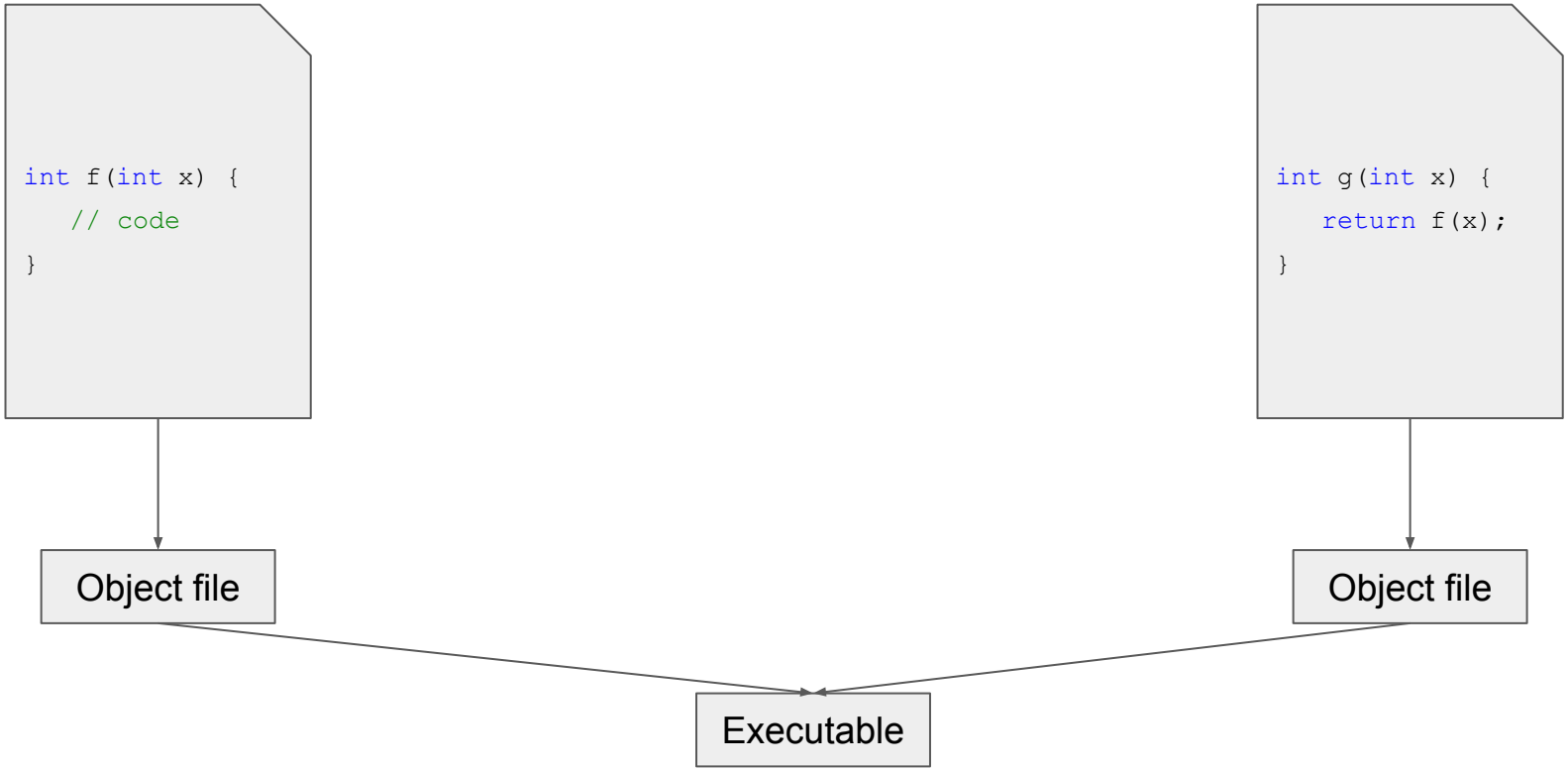
Rust -> C -> Rust -> Python

(https://github.com/lrobidou/foreign-function-interface-in-rust)

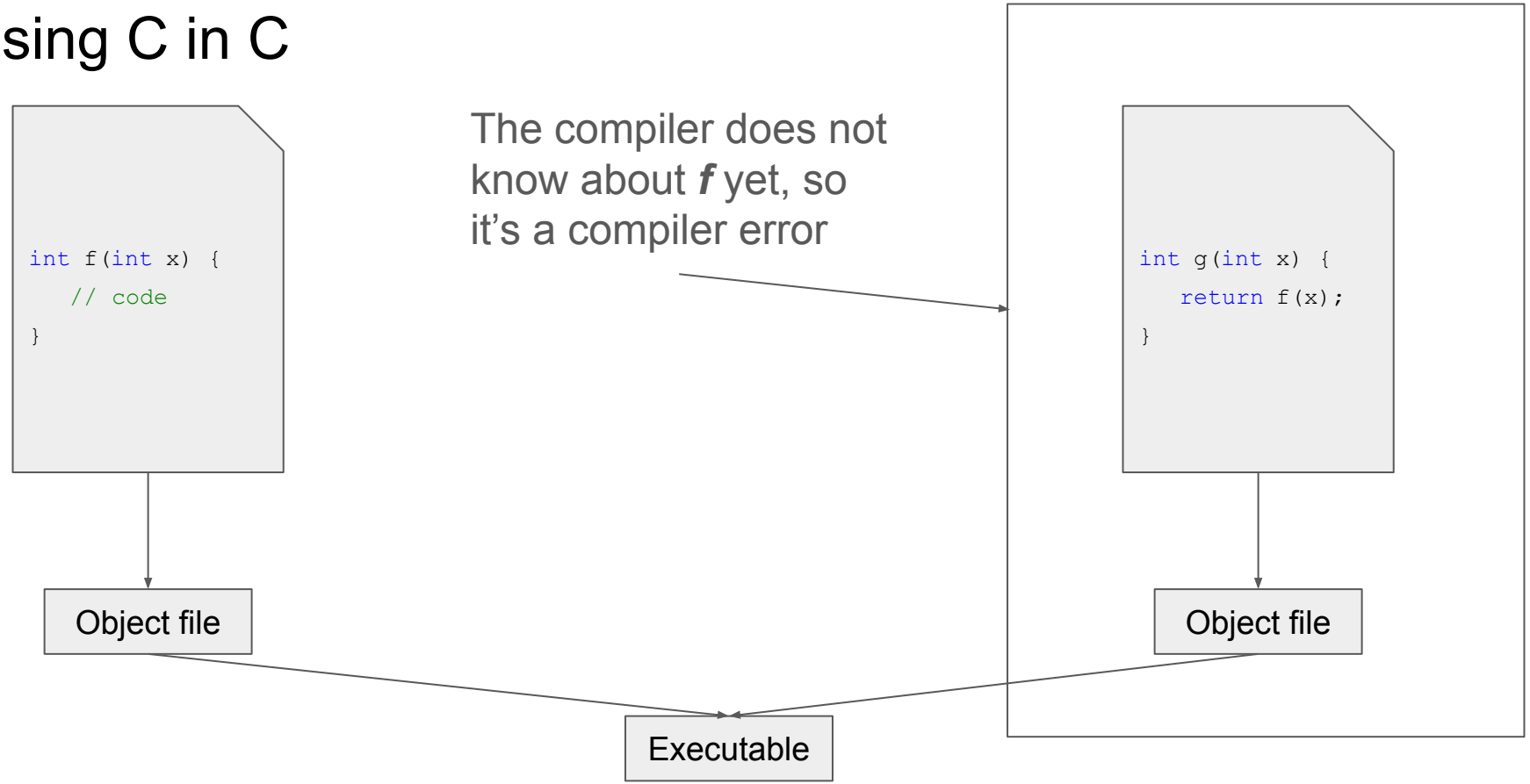
Part 1: using C in Rust

Part 1: using C in Rust

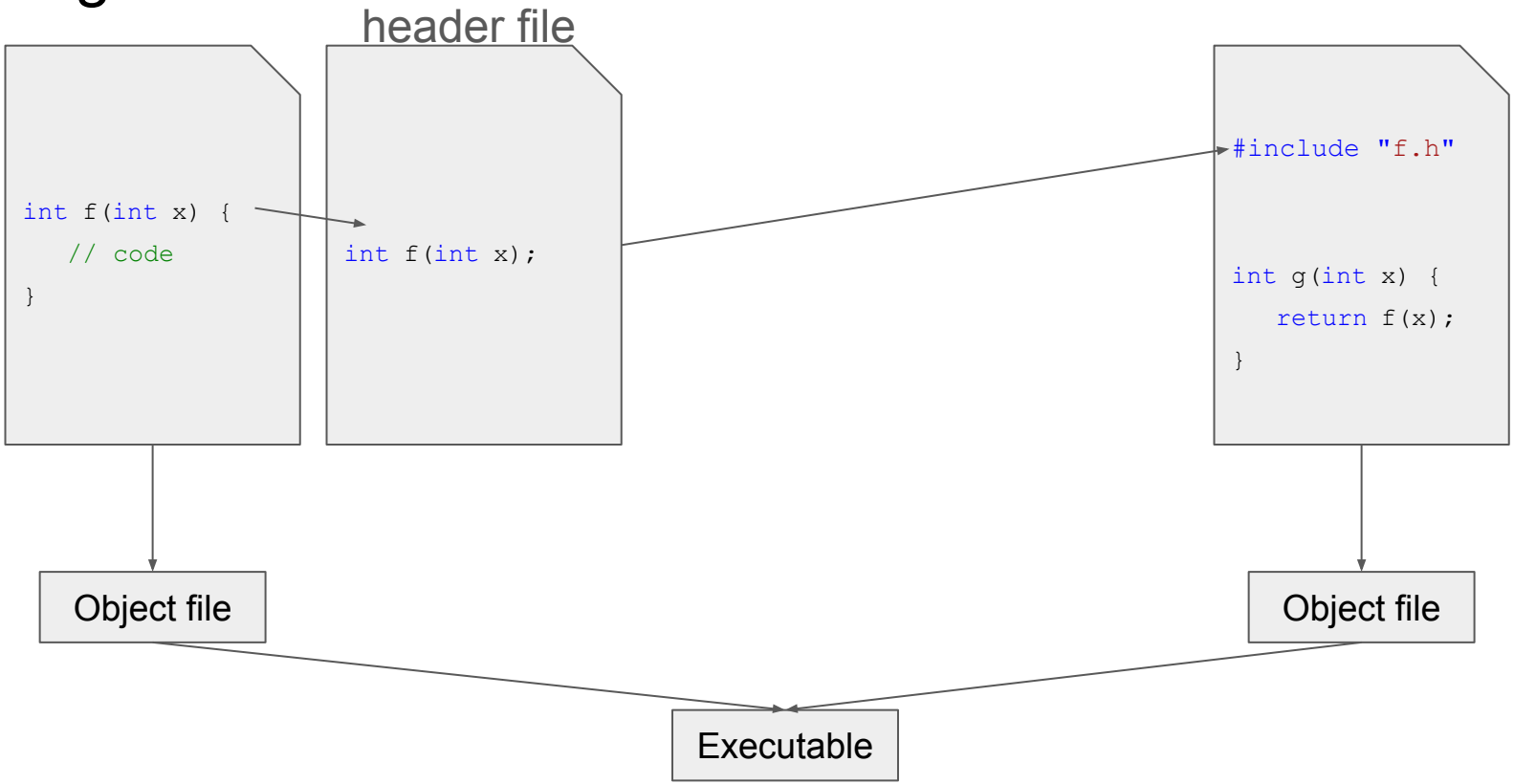
Using C in C


int f(int x) {
// code
}
int g(int x) {
return f(x);
}
Object file
Object file
Executable

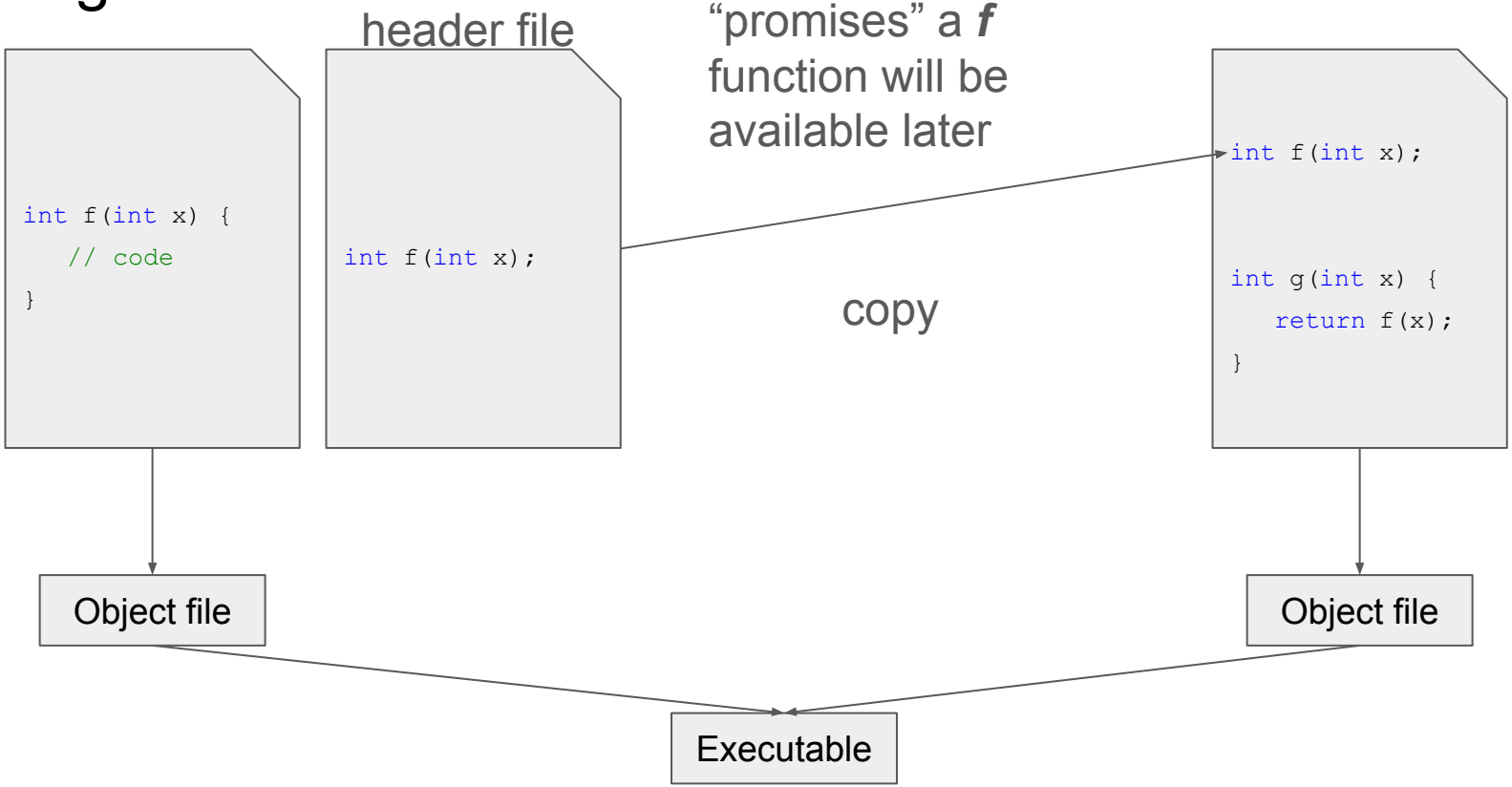
Using C in C


int f(int x) {
// code
}
The compiler does not know about f yet, so it's a compiler error
int g(int x) {
return f(x);
}
Object file
Object file
Executable

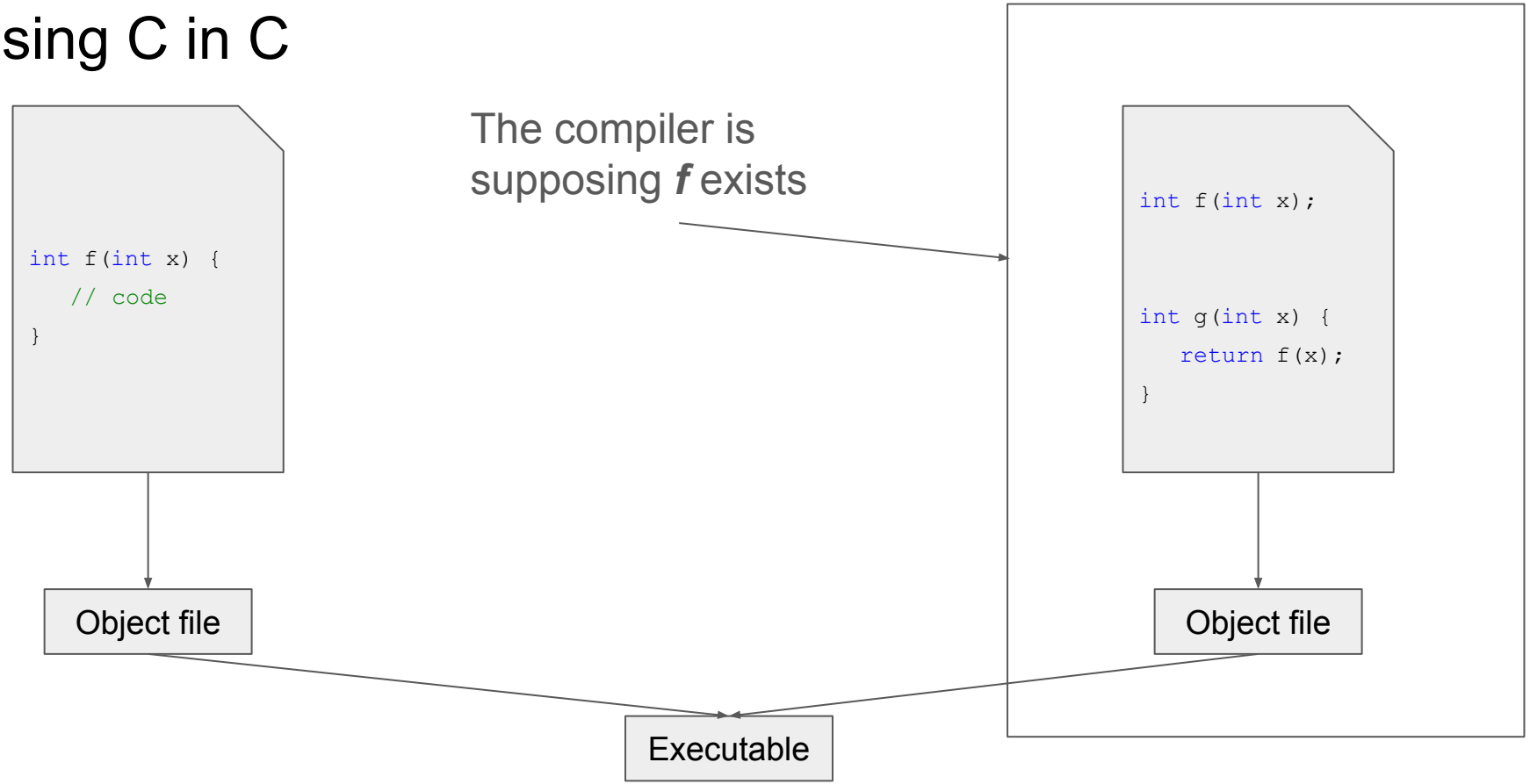
Using C in C


header file
int f(int x) {
// code
}
int f(int x);
#include "f.h"
int g(int x) {
return f(x);
}
Object file
Object file
Executable

Using C in C


header file
“promises” a *f* function will be available later
int f(int x) {
// code
}
int f(int x);
int f(int x);
int g(int x) {
return f(x);
}
copy
Object file
Object file
Executable

Using C in C


int f(int x) {
// code
}
The compiler is supposing *f* exists
int f(int x);
int g(int x) {
return f(x);
}
Object file
Object file
Executable

Using C in C

Overview:

- implement the function ***f*** in C
- place the signature of ***f*** in other C files (via header files)
- use ***f*** in other C files
- link all the C files to “fetch” the implementation of ***f***

Part 1: using C in Rust

Using C in Rust

Overview:

- implement the function ***f*** in C
- tell your rust code a function ***f*** is implemented somewhere
- use the function ***f*** in Rust
- link the C and Rust code to “fetch” the implementation of ***f***

Lets implement a stupid bloom filter in C

Do not use this implementation for real

- I used chatGPT

- the hash function is incorrect

- it uses int instead of something normalized

Example of C code (header file)

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef STUPID_BLOOM_H
#define STUPID_BLOOM_H

typedef struct {
 uint8_t* bits;
 size_t size;  // in bits
} BloomFilter;
```

```c
// Initialize a Bloom filter
BloomFilter* bloom_create(size_t num_bits);

// Free a Bloom filter
void bloom_free(BloomFilter* bf);

// Insert an integer into the Bloom filter
void bloom_insert(BloomFilter* bf, int value);

// Check if an integer is possibly in the Bloom
filter
bool bloom_query(BloomFilter* bf, int value);

#endif
```

Example of C code

```c
#include "bloom.h"

static inline size_t hash(int x, size_t size) {
 return ((uint32_t)x * 2654435761u) % size;
}

// Free a Bloom filter
void bloom_free(BloomFilter* bf) {
 if (bf) {
   free(bf->bits);
   free(bf);
 }
}

// Insert an integer into the Bloom filter
void bloom_insert(BloomFilter* bf, int value) {
 size_t h = hash(value, bf->size);
 bf->bits[h / 8] |= (1 << (h % 8));
}
```

```c
// Initialize a Bloom filter
BloomFilter* bloom_create(size_t num_bits) {
 BloomFilter* bf = malloc(sizeof(BloomFilter));
 if (!bf) {
   return NULL;
 }

 bf->size = num_bits;
 size_t num_bytes = (num_bits +7) / 8;
 bf->bits = calloc(num_bytes, sizeof(uint8_t));
 if (!bf->bits) {
   free(bf);
   return NULL;
 }
 return bf;
}

// Check if an integer is possibly in the Bloom
filter
bool bloom_query(BloomFilter* bf, int value) {
 size_t h = hash(value, bf->size);
 return bf->bits[h / 8] & (1 << (h % 8));
}
```

Makefile (use `make` to build the library)

```
# Makefile for building a shared library from bloom.c

CC = gcc
CFLAGS = -Wall -O3 -fPIC
TARGET = thirdparty/libbloom.so
SRCS = thirdparty/bloom.c
OBJS = $(SRCS:.c=.o)
HEADERS = bloom.h

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) -shared -o $@ $^

%.o: %.c $(HEADERS)
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(OBJS) $(TARGET)

.PHONY: all clean
```

Example (Rust side)

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct BloomFilter {
    pub bits: *mut u8,
    pub size: usize,
}

unsafe extern "C" {
    pub fn bloom_create(num_bits: usize) -> *mut BloomFilter;
    pub fn bloom_free(bf: *mut BloomFilter);
    pub fn bloom_insert(bf: *mut BloomFilter, value: ::std::os::raw::c_int);
    pub fn bloom_query(bf: *mut BloomFilter, value: ::std::os::raw::c_int) -> bool;
}
```

Example (Rust side)

```
mod bindings;

fn main() {
    println!("Hello, world!");
}

#[cfg(test)]
mod tests {
    use super::*;
    use bindings::{BloomFilter,
bloom_create, bloom_free, bloom_insert,
bloom_query};
```

```
    #[test]
    fn test_insert() {
        unsafe {
            let bloom: *mut BloomFilter = bloom_create(50);
            bloom_insert(bloom, 4);
            assert!(!bloom_query(bloom, 5));
            assert!(bloom_query(bloom, 4));
            bloom_free(bloom);
        }
    }
}
```

build.rs

```
fn main() {
    // Tell cargo to look for shared libraries in the specified directory
    println!("cargo:rustc-link-search=./thirdparty");

    // link with the library
    println!("cargo:rustc-link-lib=bloom");
}
```

run the tests

```
make

export LD_LIBRARY_PATH=/home/lrobidou/Documents/cours/donnés/Rust/interop/thirdparty cargo test
```

Issues

- you have to write the bindings yourself
  - an error here leads to UB
- you have to write a makefile
- you have to build the C code every time you change it
- the use looks horrible and “leaks” the “C feeling”
  - raw pointers, no impl block, no destructor

Let’s find solutions.

Automatic call and link

```
[package]
name = "interop"
version = "0.1.0"
edition = "2024"

[dependencies]


[build-dependencies]
cc = "1.2"
```

```
fn main() {
  println!("cargo:rustc-link-lib=bloom");
  println!("cargo::rerun-if-changed=thirdparty/bloom.c");
  println!("cargo::rerun-if-changed=thirdparty/bloom.h");


  cc::Build::new().file("thirdparty/bloom.c").compile("bloom");
}
```

Automatic bindings

```
[package]
name = "interop"
version = "0.1.0"
edition = "2024"

[dependencies]


[build-dependencies]
bindgen = "0.72"
cc = "1.2"
```

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


fn main() {
   println!("cargo:rustc-link-lib=bloom" );
   println!("cargo::rerun-if-changed=thirdparty/bloom.c"  );
   println!("cargo::rerun-if-changed=thirdparty/bloom.h"  );

   cc::Build::new().file( "thirdparty/bloom.c" ).compile("bloom");


    let bindings = bindgen::Builder::default()
        .header("thirdparty/bloom.h")
        .parse_callbacks(Box::new(bindgen::CargoCallbacks::new()))
        .generate()
        .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!");
}
```

Automatic bindings

```
[package]
name = "interop"
version = "0.1.0"
edition = "2024"

[dependencies]


[build-dependencies]
bindgen = "0.72"
cc = "1.2"
```

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

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

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


fn main() {
   println!("cargo:rustc-link-lib=bloom" );
   println!("cargo::rerun-if-changed=thirdparty/bloom.c"  );
   println!("cargo::rerun-if-changed=thirdparty/bloom.h"  );

   cc::Build::new().file( "thirdparty/bloom.c" ).compile("bloom");


    let bindings = bindgen::Builder::default()
        .header("thirdparty/bloom.h")
        .parse_callbacks(Box::new(bindgen::CargoCallbacks::new()))
        .generate()
        .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!");
}
```

Wrap C code in Rust struct

```rust
mod bindings;

pub struct BloomFilter {
    inner: *mut bindings::BloomFilter,
}

impl BloomFilter {
    pub fn new(size: usize) -> Self {
        Self {
            inner: unsafe { bindings::bloom_create(size) },
        }
    }

    pub fn insert(&mut self, data: i32) {
        unsafe {
            bindings::bloom_insert( self.inner, data);
        }
    }

    pub fn query(&mut self, data: i32) -> bool {
        unsafe { bindings::bloom_query( self.inner, data) }
    }
}
```

```rust
impl Drop for BloomFilter {
    fn drop(&mut self) {
        unsafe { bindings::bloom_free( self.inner) }
    }
}

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

    #[test]
    fn test_insert() {
        let mut bloom: BloomFilter = BloomFilter::new( 50);
        bloom.insert( 4);
        assert!(!bloom.query( 5));
        assert!(bloom.query( 4));
    }
}
```

Part 2: using Rust in C

Moving code from C to Rust

```rust
// no mangle: do not modify the name of the function during
compilation
// (it's turned on by default to prevent collision)
// that's unsafe, because if another function have this
name, the one that gets called is undefined
#[unsafe(no_mangle)]
extern "C" fn hash(x: i32, size: usize) -> usize {
    let mut s = DefaultHasher::new();
    x.hash(&mut s);
    let h = s.finish() as usize;
    h % size
}
```

```c
// remove code, keep declaration
size_t hash(int x, size_t size);
```

Part 3: using Rust in Python

Create a python virtual environment

```
python -m venv .env
source .env/bin/activate
```

Cargo.toml

```
[package]
name = "interop"
version = "0.1.0"
edition = "2024"

[lib]
name = "interop"
crate-type = ["cdylib"]

[dependencies]
pyo3 = { version = "0.25.0", features = ["extension-module"] }

[build-dependencies]
bindgen = "0.72"
cc = "1.2"
```

Make your wrapper safe to share between threads

```
use std::{
    ptr::NonNull,
    sync::{Arc, Mutex},
};



mod bindings;


pub struct BloomFilter {
    inner: Arc<Mutex<NonNull<bindings::BloomFilter>>>,
}

unsafe impl Send for BloomFilter {}
unsafe impl Sync for BloomFilter {}


impl BloomFilter {
```

Make your wrapper useable in Python

```
use std::{
   ptr::NonNull,
   sync::{Arc, Mutex},
};

use pyo3::{PyErr, PyResult, pyclass, pymethods};

mod bindings;

#[pyclass]
pub struct BloomFilter {
   inner: Arc<Mutex<NonNull<bindings::BloomFilter>>>,
}

unsafe impl Send for BloomFilter {}
unsafe impl Sync for BloomFilter {}

#[pymethods]
impl BloomFilter {
```

Make your wrapper importable (lib.rs)

```rust
mod bloom_filter;

use pyo3::{
   Bound, PyResult, pymodule,
   types::{PyModule, PyModuleMethods},
};

/// A Python module implemented in Rust. The name of this function must match
/// the `lib.name` setting in the `Cargo.toml`, else Python will not be able to
/// import the module.
#[pymodule]
fn interop(m: &Bound<'_, PyModule>) -> PyResult<()> {
   m.add_class::<bloom_filter::BloomFilter>()?;
   Ok(())
}
```

Issue: your unit tests now depend on Python

You'd like to

- compile your tests in one way…
- … but your main code in another way

Introducing… features

Your first feature

```
[package]
name = "interop"
version = "0.1.0"
edition = "2024"

[lib]
name = "interop"
crate-type = ["cdylib"]

[dependencies]
pyo3 = { version = "0.25.0" }

[features]
extension-module = ["pyo3/extension-module"]
default = ["extension-module"]

[build-dependencies]
bindgen = "0.72"
cc = "1.2"
```

Let’s go

```
cargo test --no-default-features
[...]
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

```
python
Python 3.11.2 (main, Apr 28 2025, 14:11:48) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import interop
>>> bloom = interop.BloomFilter(100)
>>> bloom.insert(4)
>>> bloom.query(4)
True
>>> bloom.query(5)
False
```