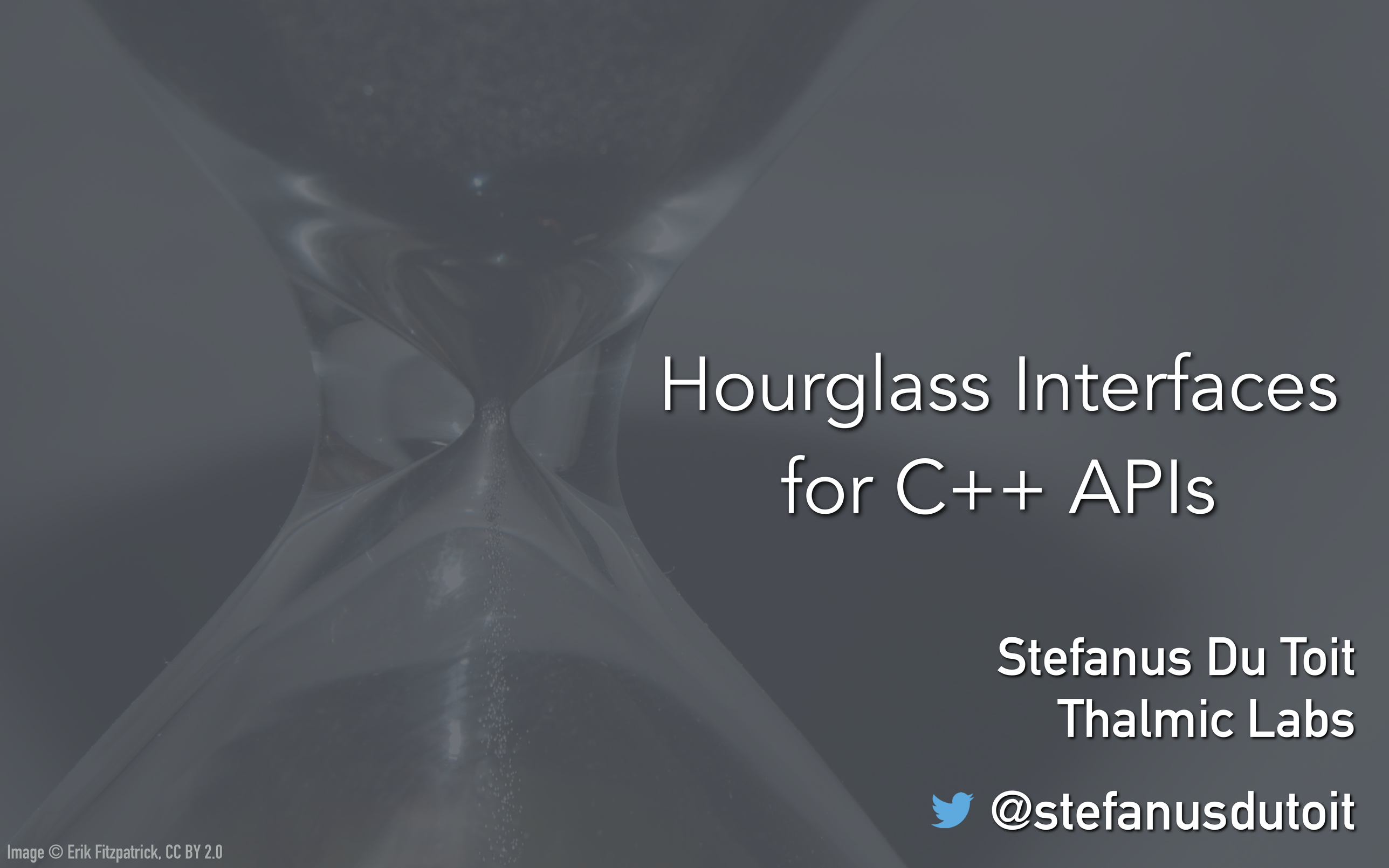


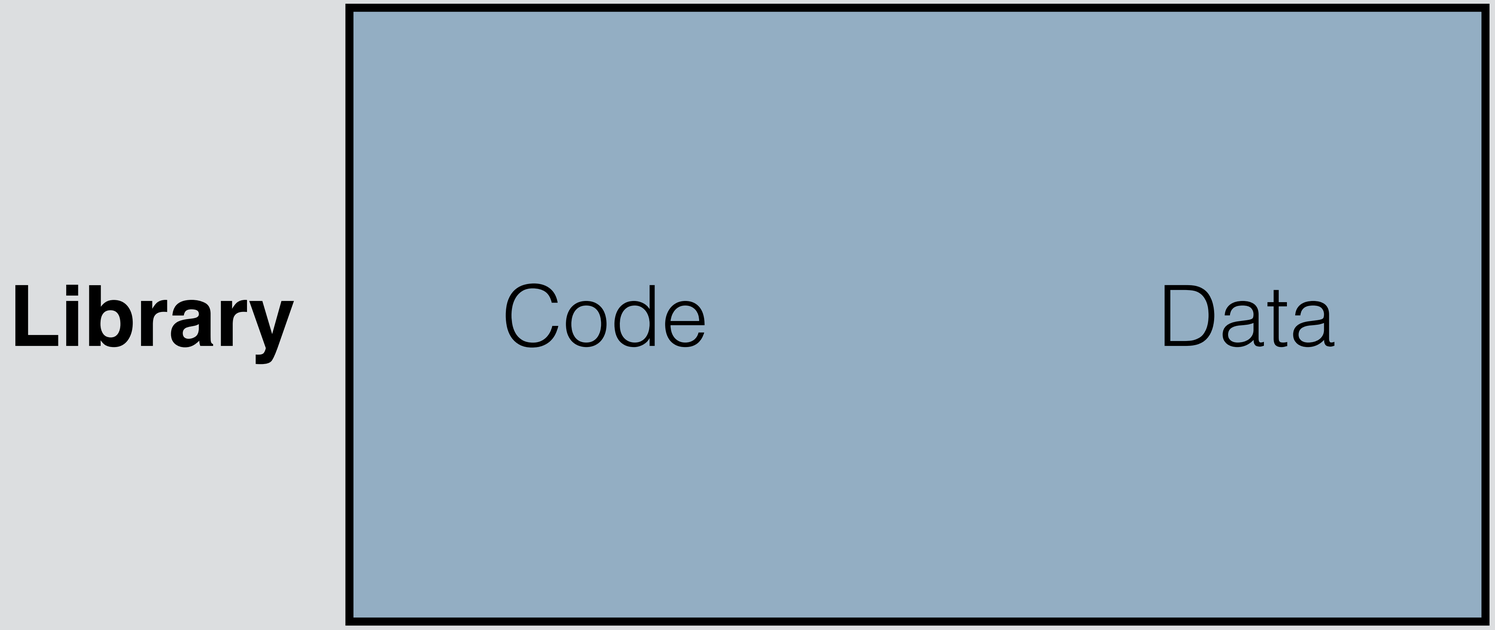
Hourglass Interfaces for C++ APIs

Stefanus Du Toit
Thalmic Labs

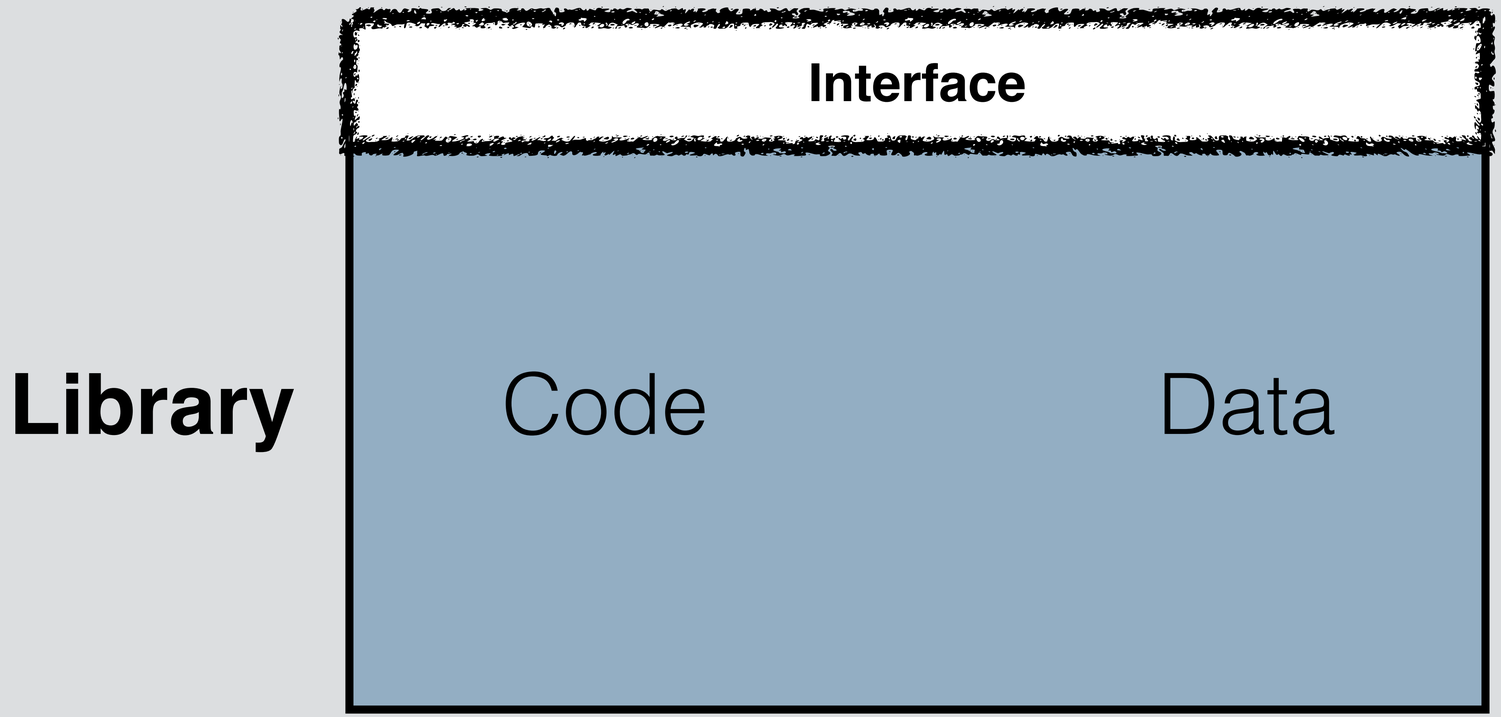
@stefanusdutoit

Motivation and Introduction

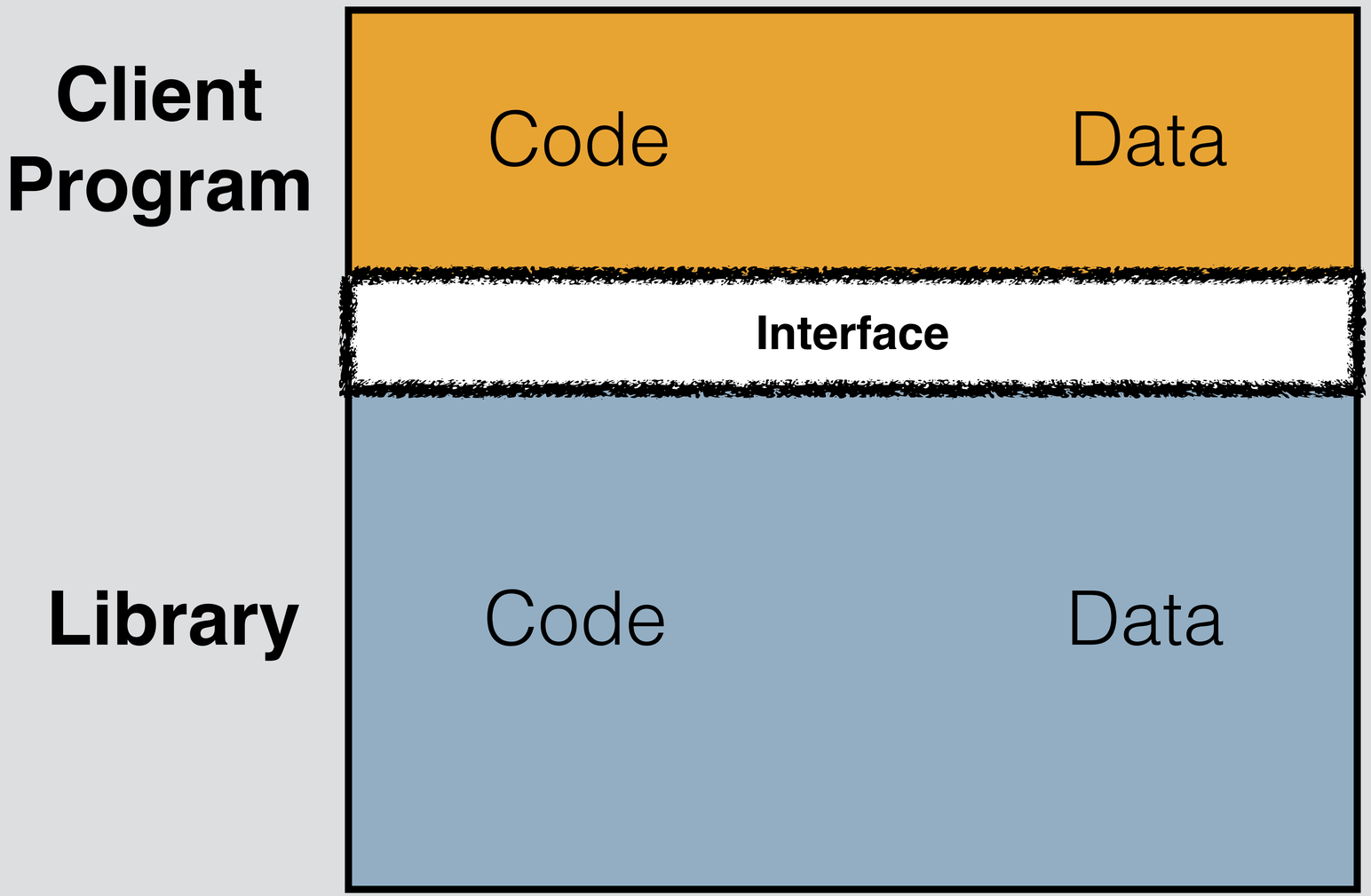
Shared Libraries



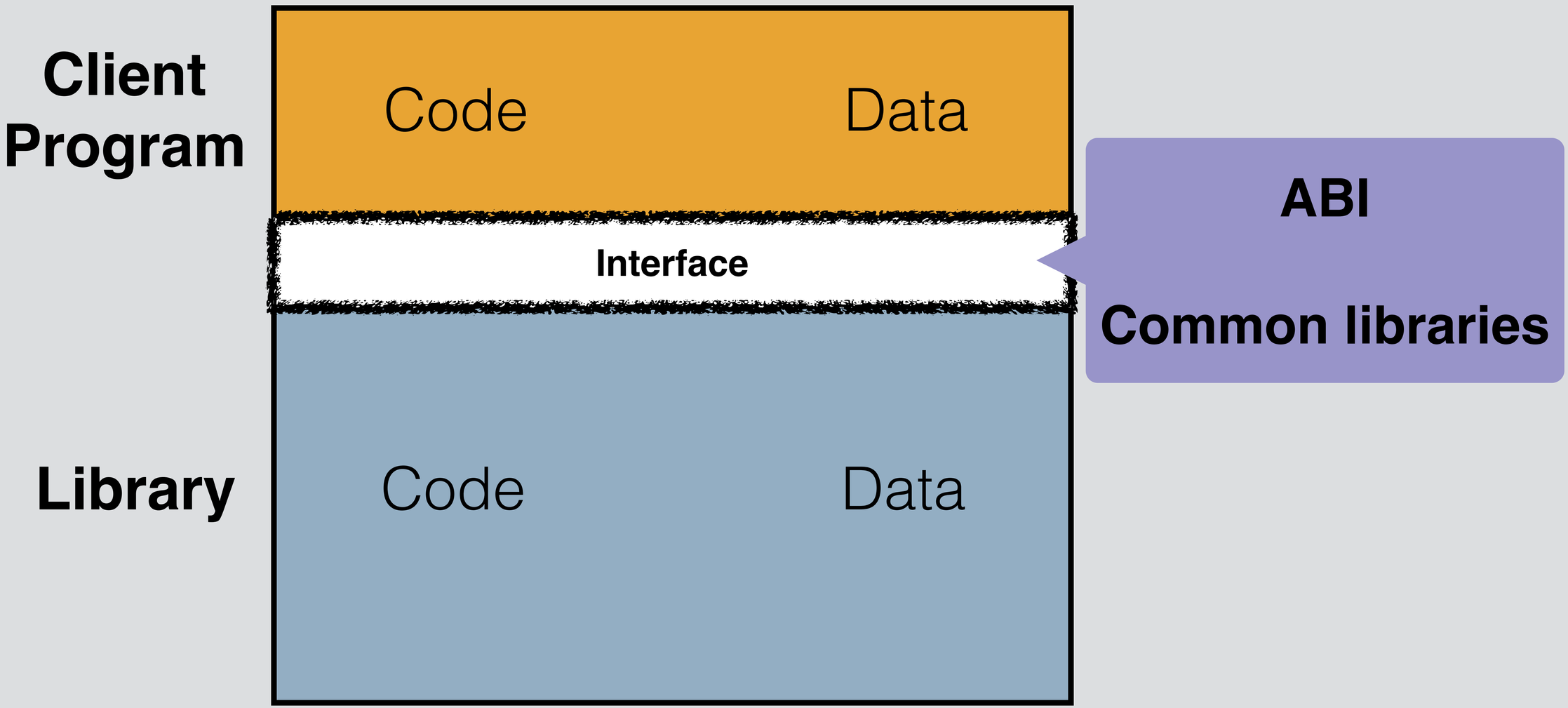
Shared Libraries



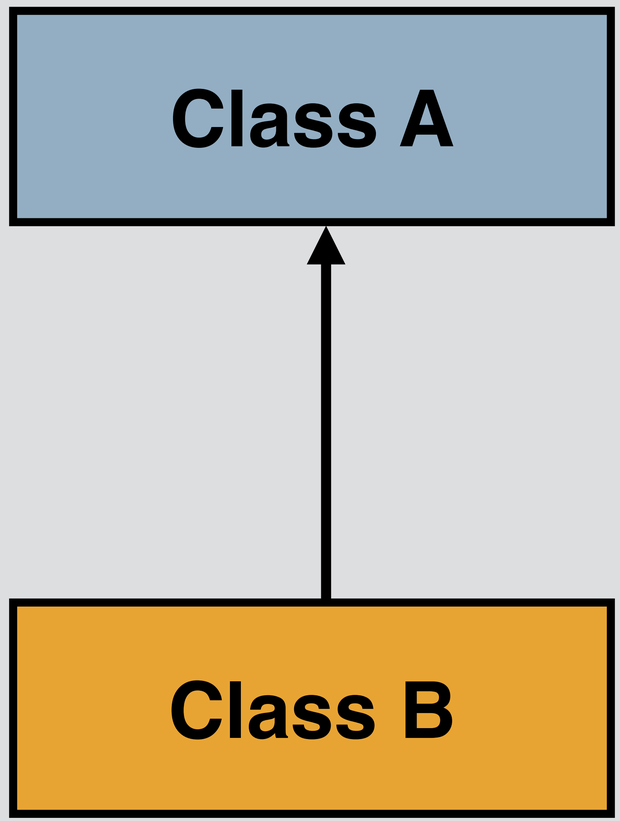
Shared Libraries



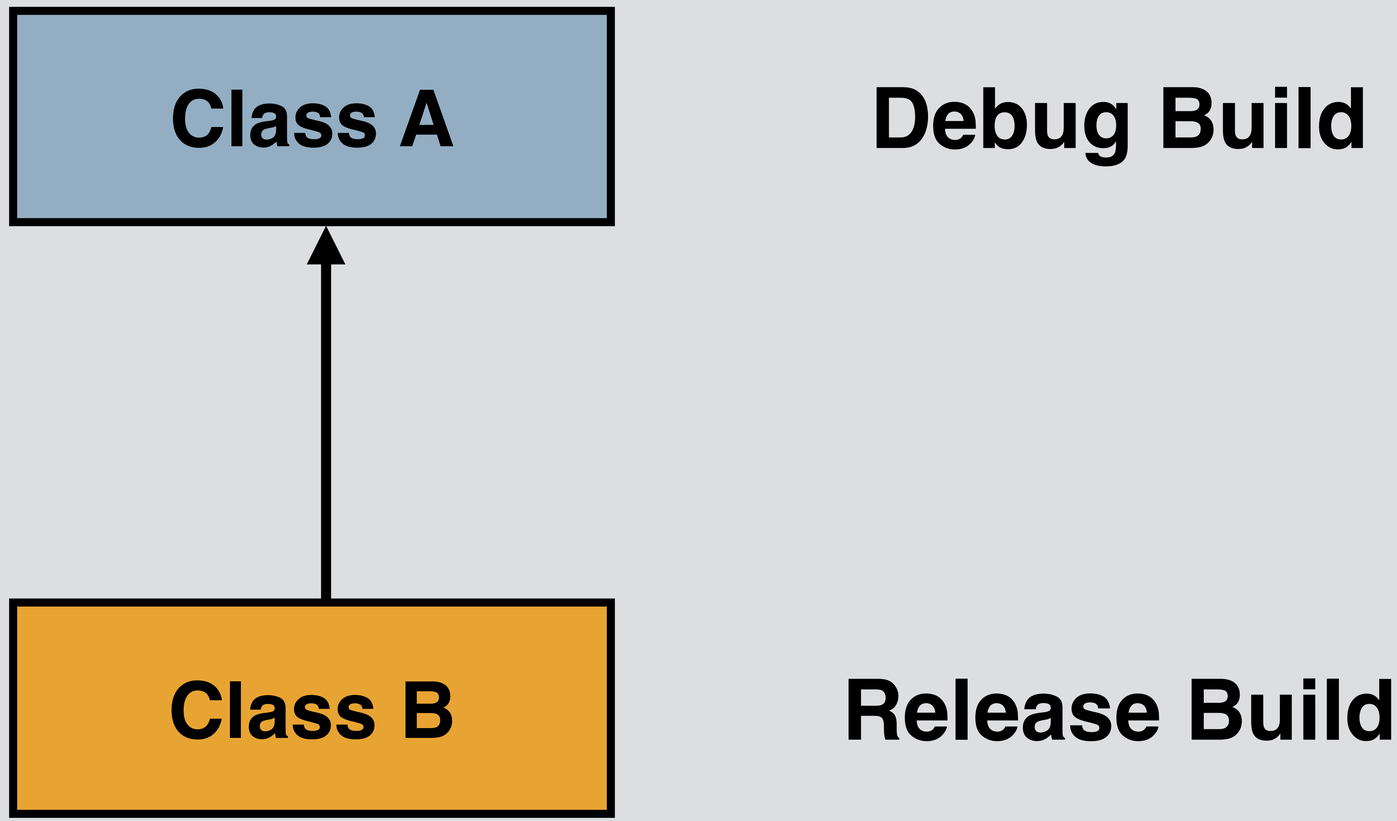
Shared Libraries



Different ABIs? Subtle problems…



Different ABIs? Subtle problems…



My Goal for Libraries

Get out of the client's way.

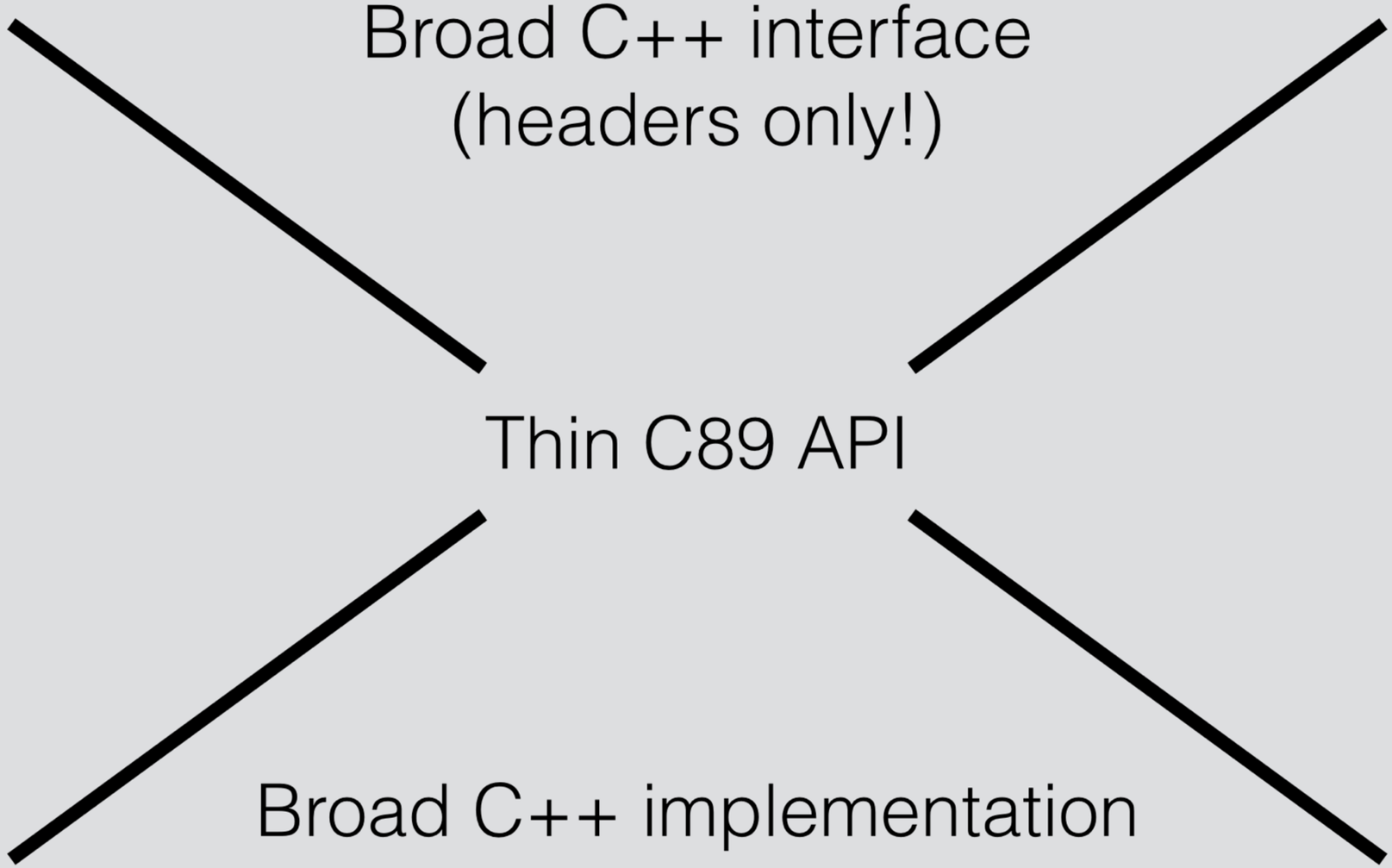
My Goal for Libraries

Don't make my clients:

- build my code if they don't want to
- change how they build their code
- rebuild their code because I shipped a fix in mine
- learn a new language if they don't want to

The Hourglass Pattern

The Hourglass Pattern



Why does this help?

- Avoids ABI-related binary compatibility issues
- Enforces no use of non-trivial C++ types in the binary interface (e.g. std::string)
- Keeps internal layout of data types a secret
- Makes binding to other languages much easier

- Still get all the convenience of C++ at both ends

Example: libhairpoll

- A library for creating hair-related polls, taking votes, and tallying the results

hairpoll.h

```
typedef struct hairpoll* hairpoll_t;

hairpoll_t hairpoll_construct(const char* person);
void hairpoll_destruct(hairpoll_t poll);

int32_t hairpoll_add_option(hairpoll_t hairpoll,
                            const char* name, const char* image_url);

void hairpoll_vote(hairpoll_t hairpoll, int32_t option);

typedef void (*hairpoll_result_handler_t)(void* client_data,
          const char* name, int32_t votes, const char* html);

void hairpoll_tally(const hairpoll_t hairpoll,
      hairpoll_result_handler_t handler, void* client_data);
```

HairPoll class - client-facing

```
class HairPoll {
public:
    HairPoll(std::string person);
    ~HairPoll();

    int addOption(std::string name, std::string imageUrl);
    void vote(int option);

    struct Result {
        std::string name;
        int votes;
        std::string html;
    };
    std::vector<Result> results() const;

private:
    hairpoll_t _opaque;
};
```

Best Practices

Parts of C to use in the thin API

- C89 + const and // comments
- Functions, but no variadic functions
- C primitive types (char, void, etc.)
- Pointers
- forward-declared structs
- enumerations
- Function pointers are OK, too

Opaque Types

```
typedef struct hairpoll* hairpoll_t;

void hairpoll_vote(
  hairpoll_t hairpoll,
  int32_t option
);
```

Opaque Types

```
extern "C"
struct hairpoll {
    Poll actual;
};
```

Opaque Types

```
extern "C"
struct hairpoll {
    template<typename... Args>
    hairpoll(Args&&... args)
    : actual(std::forward<Args>(args)...)
    {
    }

    Poll actual;
};
```

Opaque Types

```
class Poll {
public:
    Poll(std::string person);

    struct option {
        option(std::string name, std::string url);

        std::string name;
        std::string url;
        int votes;
    };

    std::string person;
    std::vector<option> options;
};
```

Opaque Types

```
hairpoll_t
hairpoll_construct(const char* person)
{
    return new hairpoll(person);
}

void hairpoll_destruct(hairpoll_t poll)
{
    delete poll;
}
```

Integral types

Prefer stdint.h types (int32_t, int64_t, etc.) over plain C integer types (short, int, long):

```
void hairpoll_vote(
  hairpoll_t hairpoll,
  int32_t option
);
```

Integral types

Using char is OK, though:

```
int32_t
hairpoll_add_option(
  hairpoll_t hairpoll,
  const char* name,
  const char* image_url
);
```

Enumerations

```
typedef enum motu_alignment {
    heroic_warrior = 0,
    evil_warrior = 1,
};

int32_t motu_count(
    int32_t alignment
);
```

Error Handling

```
void hairpoll_vote(
  hairpoll_t hairpoll,
  int32_t option,
  error_t* out_error
);
```

Error Handling

```
typedef struct error* error_t;

const char* error_message(
  error_t error
);

void error_destruct(error_t error);

void hairpoll_vote(hairpoll_t hairpoll, int32_t option, error_t* out_error);
```

Errors → Exceptions

```
struct Error {
    Error() : opaque(nullptr) {}
    ~Error() { if (opaque) { error_destruct(opaque); } }
    error_t opaque;
};

class ThrowOnError {
public:
    ~ThrowOnError() noexcept(false) {
        if (_error.opaque) {
            throw std::runtime_error(error_message(_error.opaque));
        }
    }

    operator error_t*() { return &_error.opaque; }

private:
    Error _error;
};
```

Errors → Exceptions

```
void hairpoll_vote(hairpoll_t hairpoll, int32_t option, error_t* out_error);
```

```
void vote(int option) {
    return hairpoll_vote(
        _opaque,
        option,
        ThrowOnError{}
    );
}
```

Exceptions → Errors

```
template<typename Fn>
bool translateExceptions(error_t* out_error, Fn&& fn)
{
    try {
        fn();
    } catch (const std::exception& e) {
        *out_error = new error{e.what()};
        return false;
    } catch (...) {
        *out_error = new error{"Unknown internal error"};
        return false;
    }
    return true;
}
```

Exceptions → Errors

```
void hairpoll_vote(const hairpoll_t poll,
                   int32_t option, error_t* out_error)
{
    translateExceptions(out_error, [&]{
      if (option < 0 ||
          option >= poll->actual.options.size()) {
        throw std::runtime_error("Bad optn index");
      }

      poll->actual.options[option].votes++;
    });
}
```

Callbacks

```
typedef void (*hairpoll_result_handler_t)(
  void* client_data,
  const char* name,
  int32_t votes,
  const char* html
);

void hairpoll_tally(const hairpoll_t hairpoll,
  hairpoll_result_handler_t handler,
  void* client_data, error_t* out_error);
```

Using lambdas as callbacks

```
std::vector<Result> results() const {
    std::vector<Result> ret;

    auto addResult = [&ret](const char* name, int32_t votes,
                            const char* html){
        ret.push_back(Result{name, votes, html});
    };

    auto callback = [](void* client_data, const char* name, int32_t votes,
                       const char* html){
        auto fn = static_cast<decltype(&addResult)>(client_data);
        (*fn)(name, votes, html);
    };

    hairpoll_tally(_opaque, callback, &addResult, ThrowOnError{});

    return ret;
}
```

Symbol visibility

```
#if defined(_WIN32) || defined(__CYGWIN__)
  #ifdef hairpoll_EXPORTS
    #ifdef __GNUC__
      #define HAIRPOLL_EXPORT __attribute__ ((dllexport))
    #else
      #define HAIRPOLL_EXPORT __declspec(dllexport)
    #endif
  #else
    #ifdef __GNUC__
      #define HAIRPOLL_EXPORT __attribute__ ((dllimport))
    #else
      #define HAIRPOLL_EXPORT __declspec(dllimport)
    #endif
  #endif
#else
  #if __GNUC__ >= 4
    #define HAIRPOLL_EXPORT __attribute__ ((visibility ("default")))
  #else
    #define HAIRPOLL_EXPORT
  #endif
#endif
```

Symbol visibility

```
#include "visibility.h"

…

HAIRPOLL_EXPORT
hairpoll_t hairpoll_construct(const char* person);

HAIRPOLL_EXPORT
void hairpoll_destruct(hairpoll_t poll);

…
```

Symbol visibility

On Clang and GCC, add to your compile:

```
-fvisibility=hidden
```

(applies to library only)

Remember to extern “C”!

```
#ifdef __cplusplus
extern "C" {
#endif

// C API goes here

#ifdef __cplusplus
} // extern "C"
#endif
```

Lifetime and Ownership

- Keep it simple: construct (if needed) and destruct
- Always use RAII on the client side!
- Can use refcounting (e.g. shared_ptr) both internally to the library and in the client
- For callbacks, you can use the stack

Interfacing with those “other” languages

Foreign Function Interfaces

- “something that allows calling code written in one language from another language”
- C = de-facto standard
- Languages with support for calling C functions in shared libraries include:
  - Common Lisp, Java, JavaScript, Matlab, .NET (C#, F#, etc.), Python, Ruby, OCaml, basically everything…

Hair Polls in Python

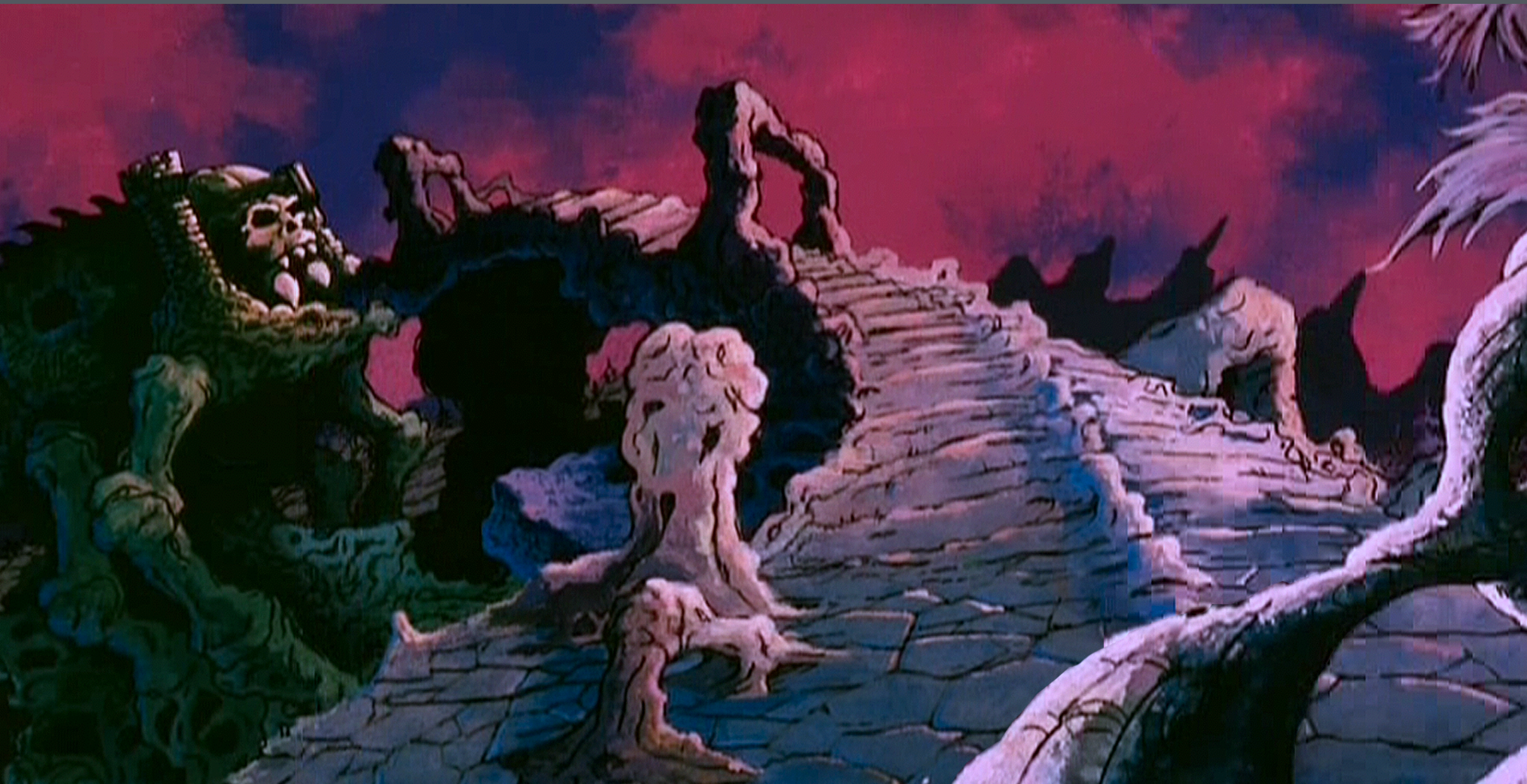
```
hairpoll = lib.hairpoll_construct("Stefanus Du Toit", None)

skeletor = lib.hairpoll_add_option(hairpoll, "Skeletor",
                                   "<long URL omitted>", None)
beast = lib.hairpoll_add_option(hairpoll, "Beast Man",
                                "<long URL omitted>", None)

lib.hairpoll_vote(hairpoll, skeletor, None)
lib.hairpoll_vote(hairpoll, beast, None)
lib.hairpoll_vote(hairpoll, beast, None)

def print_result(client_data, name, votes, html):
    print name, votes

lib.hairpoll_tally(hairpoll, hairpoll_result_handler(print_result),
                   None, None)

lib.hairpoll_destruct(hairpoll)
```

Hair Polls In Python

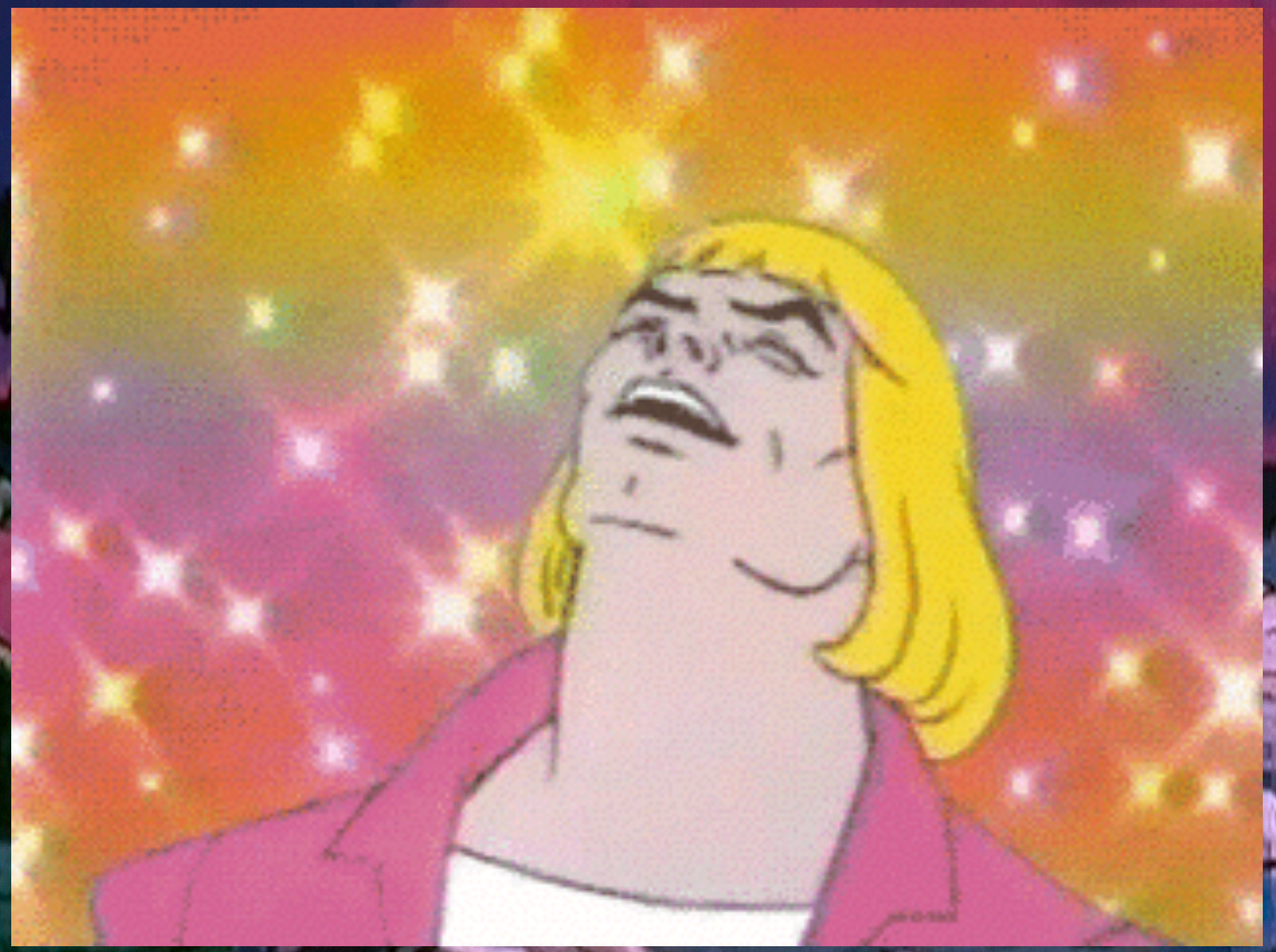
```
from ctypes import *

lib = CDLL("libhairpoll.dylib")
lib.hairpoll_construct.restype = c_void_p
lib.hairpoll_construct.argtypes = [c_char_p, c_void_p]

lib.hairpoll_destruct.restype = None
lib.hairpoll_destruct.argtypes = [c_void_p]

lib.hairpoll_add_option.restype = c_int
lib.hairpoll_add_option.argtypes = [c_void_p, c_char_p, c_char_p, c_void_p]

lib.hairpoll_vote.restype = None
lib.hairpoll_vote.argtypes = [c_void_p, c_int, c_void_p]

hairpoll_result_handler = CFUNCTYPE(None, c_void_p, c_char_p, c_int, c_char_p)
lib.hairpoll_tally.restype = None
lib.hairpoll_tally.argtypes = [c_void_p, hairpoll_result_handler, c_void_p,
                               c_void_p]
```

Summary

- Hourglass = C++ on top of C89 on top of C++
- Avoids ABI issues, sneaky dependencies, etc.
- Hides object layout and other implementation details
- Makes FFI bindings easy



Q&A



http://blog.sduto.it/

@stefanusdutoit