Department I - C Plus Plus

Modern and Lucid C++ Advanced
for Professional Programmers

Week 13 – Hourglass Interfaces

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 16.05.2019
FS2019

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Hourglass Interfaces

based on Stefanus DuToit "Hourglass Interfaces" talk at CppCon 2016

https://www.youtube.com/watch?v=PVYdHDm0q6Y

- **DLL APIs work best (and cross-platform compatible) with C only**

  - We ignore the Windows burden of providing DLL-export and DLL-import syntax

- **C++ can provide C-compatible function interfaces using `extern "C"` in front of a declaration**

- **C-APIs are error-prone and can be tedious to use**

- **C++ exceptions do not pass nicely across a C-API**

- **Foreign language bindings (e.g. for Python etc) often expect C-APIs**

- **API - Application Programming Interface**

  - If stable, you do not need to change your code, if something changes

- **ABI - Application Binary Interface**

  - If stable, you can use and share DLLs/shared libraries without recompilation

- **Not universally applicable, but very common**

- **Shape of an hourglass**

C++ Client

C++ Client API
Header Only
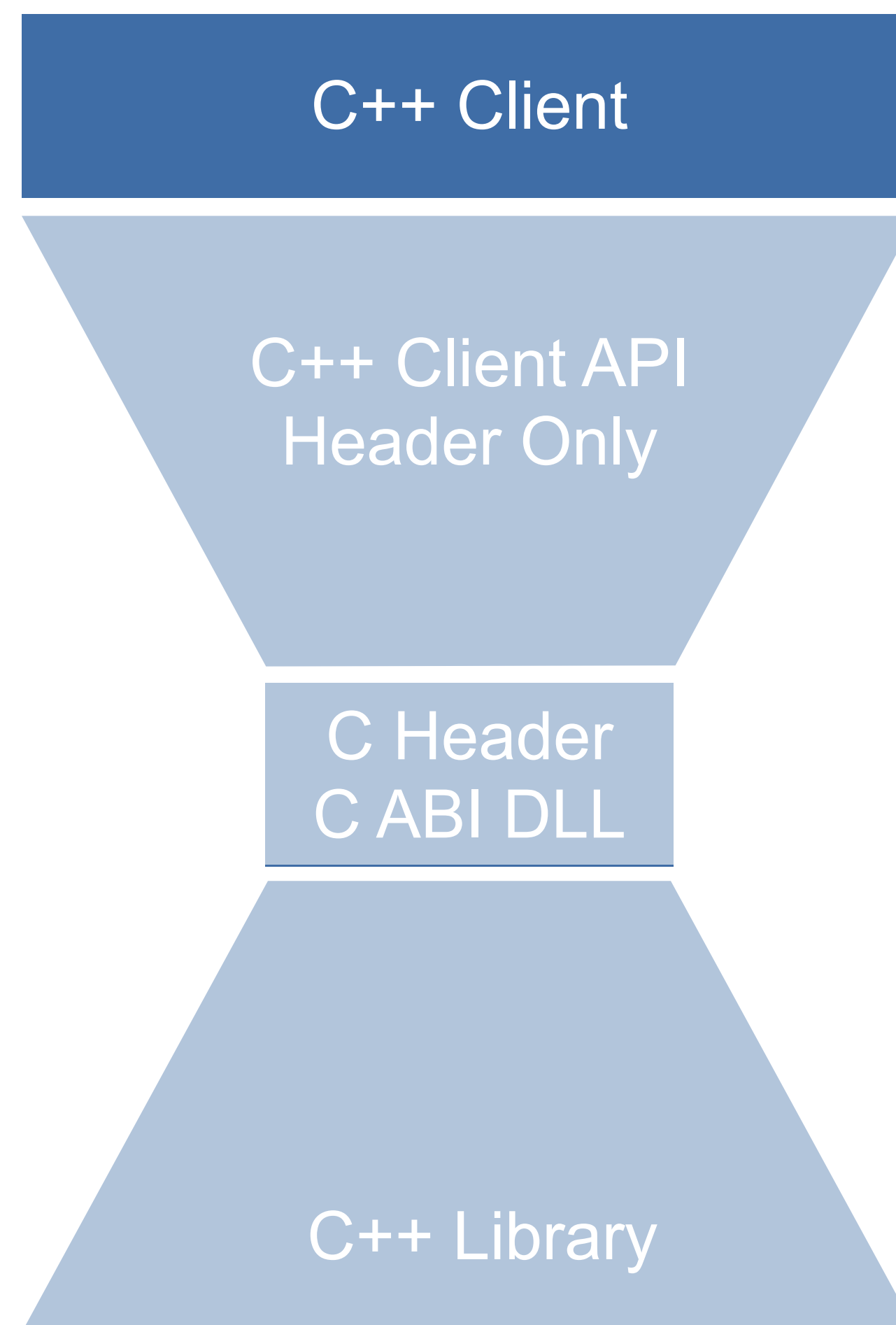
C Header
C ABI DLL

C++ Library
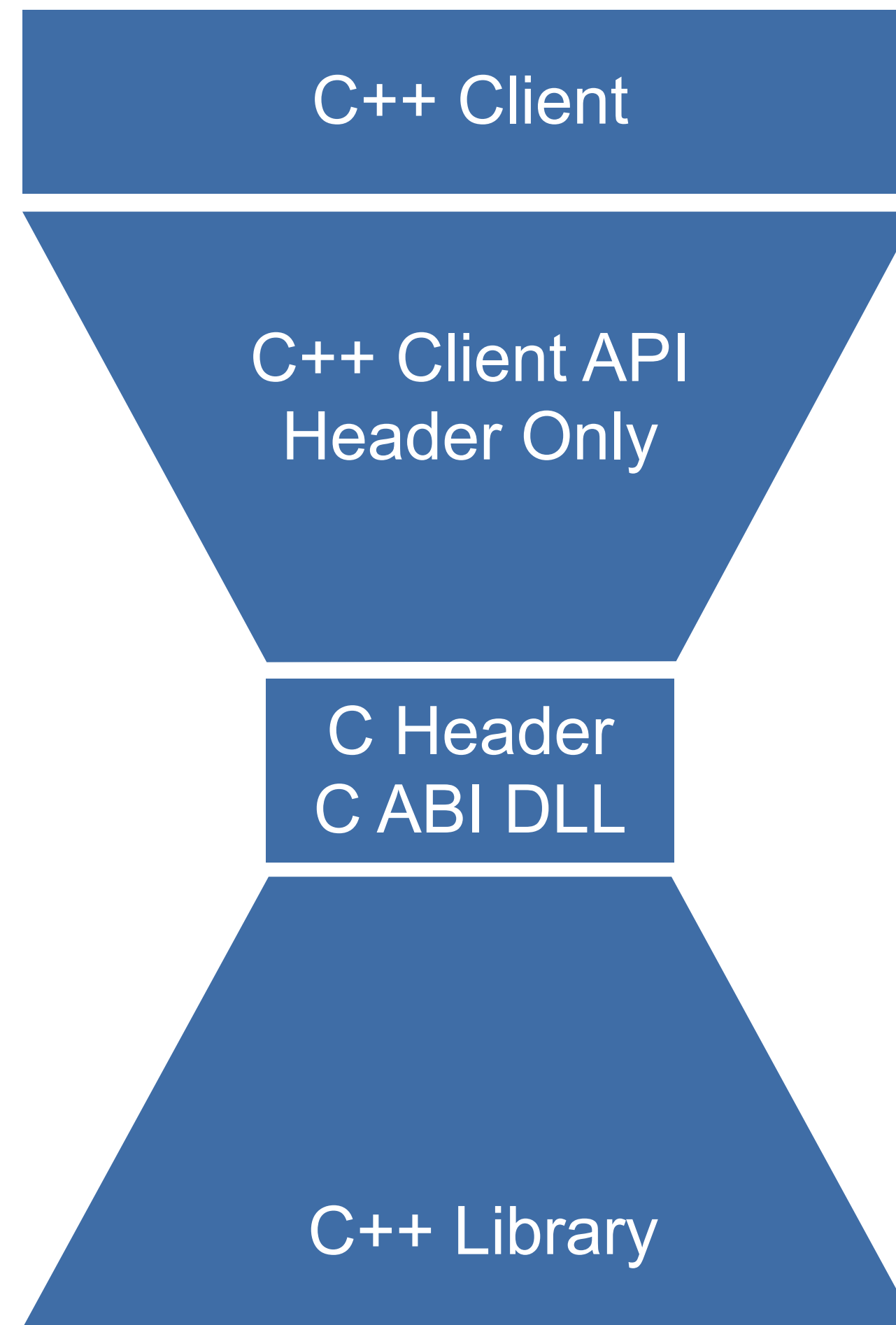
- **Let's add some functionality to our Wizard**

  - `doMagic()` – still casts a spell ("wootsh") or uses a potion ("zapp")

  - `learnSpell()` – learns a new spell (by name)

  - `maxAndStorePotion()` – creates a potion and puts it to the inventory

  - `getName()` – function to make Java programmers happy, otherwise there wouldn't be a "getX" function

```cpp
struct Wizard {
  Wizard(std::string name = "Rincewind")
    : name{name}, wand{} {
  }
  char const * doMagic(std::string const & wish);
  void learnSpell(std::string const & newspell);
  void mixAndStorePotion(std::string const & potion);
  char const * getName() const {
    return name.c_str();
  }
};
```

C++ Client

C++ Client API
Header Only

C Header
C ABI DLL

C++ Library
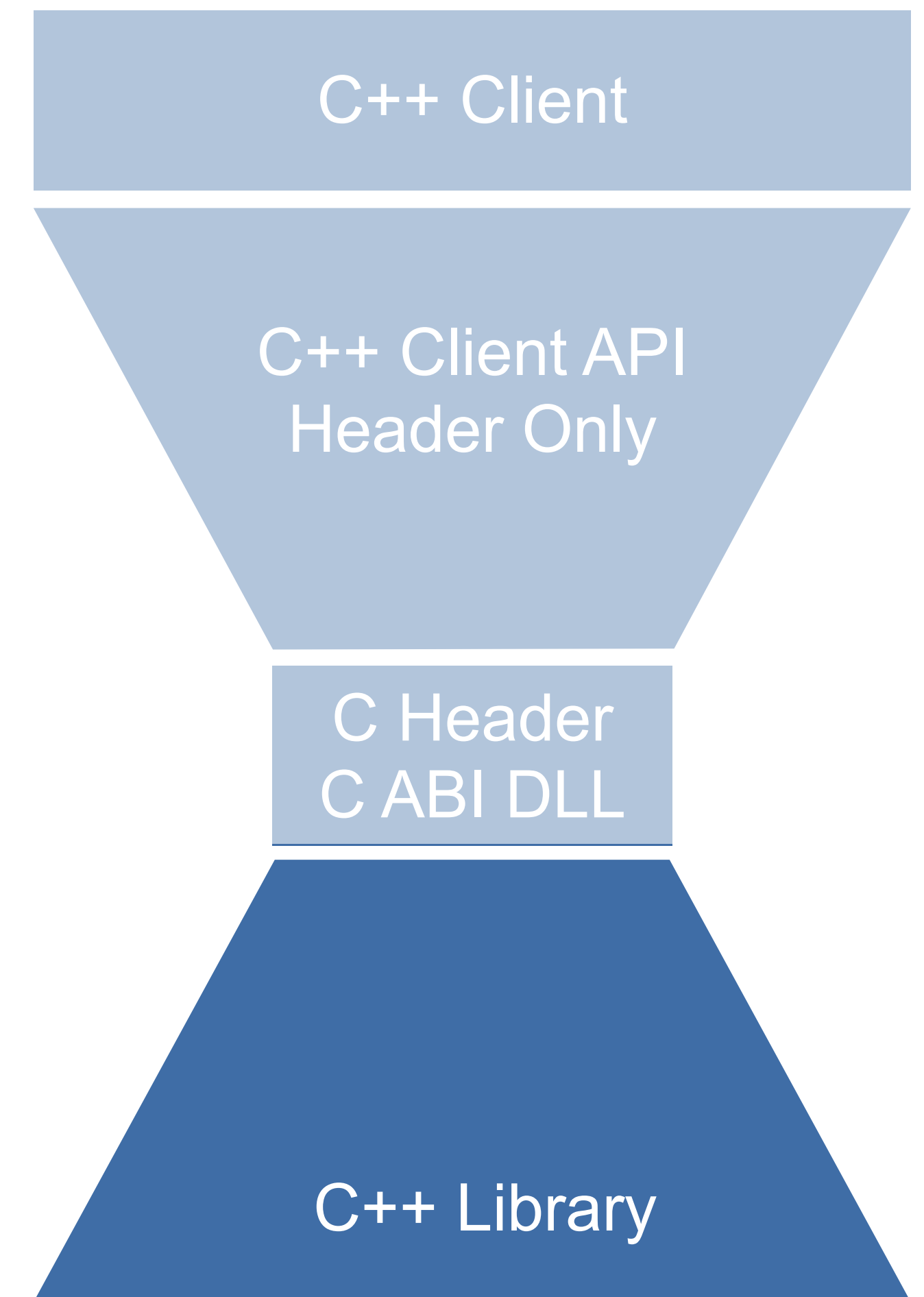
- **Testing a wizard provides the same view a client has**

C++ Client

C++ Client API
Header Only

C Header
C ABI DLL

C++ Library

```cpp
using wizard_client::Wizard;

void canCreateDefaultWizard() {
  Wizard const magician{};
  ASSERT_EQUAL("Rincewind",magician.getName());
}

void canCreateWizardWithName() {
  Wizard const magician{ "Petrosilius Zwackelmann" };
  ASSERT_EQUAL("Petrosilius Zwackelmann", magician.getName());
}

void wizardLearnsSpellAndCanRecall() {
  Wizard magician{};
  magician.learnSpell("Expelliarmus");
  ASSERT_EQUAL("wootsh", magician.doMagic("Expelliarmus"));
}

void wizardMixesPotionAndCanApply() {
  Wizard magician{};
  magician.mixAndStorePotion("Polyjuice Potion");
  ASSERT_EQUAL("zapp", magician.doMagic("Polyjuice Potion"));
}
void uknownMagicFails() {
  Wizard magician{};
  ASSERT_THROWS(magician.doMagic("Expecto Patronum!"), std::runtime_error);
}
```

- **Abstract data types can be represented by pointers**

  - Ultimate abstract pointer `void *`

- **Member functions map to functions taking the abstract data type pointer as first argument**

- **Requires Factory and Disposal functions to manage object lifetime**

- **Strings can only be represented by `char *`**

  - Need to know who will be responsible for memory

  - Make sure not to return pointers to temporary objects!

- **Exceptions do not work across a C API**

C++ Client

C++ Client API
Header Only

C Header
C ABI DLL

C++ Library
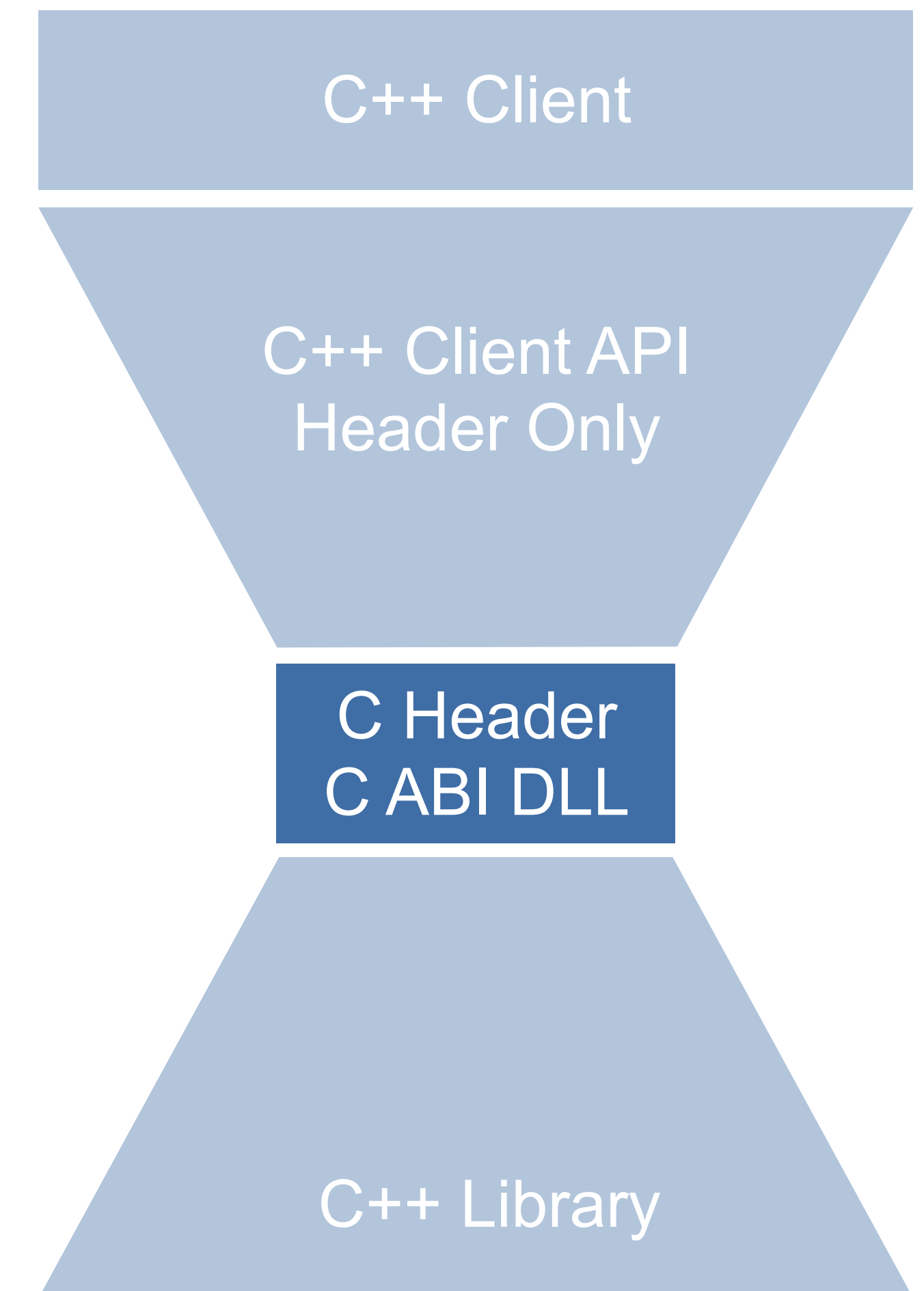
- **A Wizard can only be accessed through a pointer (const and non-const)**

  ▪ Construction and destruction through functions

- **An error pointer stores messages of exceptions**

  ▪ Functions that may fail need an error pointer parameter for reporting exceptions

  ▪ Errors need to be cleaned up when not used anymore

- **Member functions take a Wizard (pointer) as first parameter**

Wizard.h

```c
typedef struct Wizard * wizard;
typedef struct Wizard const * cwizard;
wizard createWizard(char const * name,
                    error_t * out_error);
void disposeWizard(wizard toDispose);


typedef struct Error * error_t;
char const * error_message(error_t error);
void error_dispose(error_t error);


char const *doMagic(wizard w,
                    char const * wish,
                    error_t *out_error);
void learnSpell(wizard w,
                char const * spell);
void mixAndStorePotion(wizard w,
                       char const * potion);
char const *wizardName(cwizard w);
```

What Parts of C++ Can Be Used in an **extern "C"** Interface?

9

- **Functions, but not templates or variadic**

  - No overloading in C!

- **C primitive types (`char, int, double, void`)**

- **Pointers, including function pointers**

- **Forward-declared structs**

  - Pointers to those are opaque types!

  - Are used for abstract data types

- **Enums (unscoped - without class or base type!)**

- **If using from C must embrace it with extern "C" when compiling it with C++**

  - Otherwise names do not match, because of mangling

Wizard.h

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

typedef struct Wizard * wizard;
typedef struct Wizard const * cwizard;
wizard createWizard(char const * name,
                    error_t * out_error);
void disposeWizard(wizard toDispose);

// ...
// Comments are ok too, as the preprocessor
// eliminates them anyway

#ifdef __cplusplus
}
#endif
```

- **Wizard class must be implemented**

- **To allow full C++ including templates, we need to use a "trampoline" class**

  ▪ It wraps the actual Wizard implementation

WizardHidden.h

```cpp
namespace unseen {
struct Wizard {
  // ...
  Wizard(std::string name = "Rincewind")
    : name{name}, wand{} {
  }
  char const * doMagic(std::string const & wish);
  void learnSpell(std::string const & newspell);
  void mixAndStorePotion(std::string const & potion);
  char const * getName() const {
    return name.c_str();
  }
};
}
```

Wizard.cpp

```cpp
extern "C" {
struct Wizard { // C linkage trampoline
  Wizard(char const * name)
    : wiz{name} {
  }
  unseen::Wizard wiz;
};
}
```

Note: The Hairpoll example of Stefanus Du Toit has non-standard code in the trampoline

- **Remember the 5 ways to deal with errors!**

- **You can't use references in C API, must use pointers to pointers**

- **In case of an error, allocate error value on the heap**

  ▪ You must provide a disposal function to clean up

- **You can use C++ types internally (`std::string`)**

- **It is safe to return the `char const *`**

  ▪ because caller owns the object providing the memory

`Wizard.h`

```cpp
typedef struct Error * error_t;
char const * error_message(error_t error);
void error_dispose(error_t error);

wizard createWizard(char const * name,
                    error_t * out_error);
```

`Wizard.cpp`

```cpp
extern "C" {
struct Error {
  std::string message;
};

const char * error_message(error_t error) {
  return error->message.c_str();
}

void error_dispose(error_t error) {
  delete error;
}
}
```

- **Call the function body and catch exceptions**

- **Map them to an Error object**

- **Set the pointer pointed to by `out_error`**

  - Use pointer to pointer as reference to pointer

  - Passed `out_error` must not be `nullptr`!

Wizard.cpp

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

wizard createWizard(const char * name,
                    error_t * out_error) {
  wizard result = nullptr;
  translateExceptions(out_error,[&] {
    result = new Wizard{name};
  });
  return result;
}
```

- **Client-side C++ usage requires mapping error codes back to exceptions**

  - Unfortunately exception type doesn't map through

  - But can use a generic standard exception

    - `std::runtime_error`, keep the message

  - Dedicated RAII class for disposal

- **Temporary object with throwing destructor**

  - Strange but possible

  - Automatic type conversion passes the address of its guts (opaque)

  - Tricky, take care you don't leak when creating the object!

WizardClient.h

```cpp
struct ErrorRAII {
  ErrorRAII(error_t error) : opaque {error} {}
  ~ErrorRAII() {
    if (opaque) {
      error_dispose(opaque);
    }
  }
  error_t opaque;
};

struct ThrowOnError {
  ThrowOnError() = default;
  ~ThrowOnError() noexcept(false) {
    if (error.opaque) {
      throw std::runtime_error{error_message(error.opaque)};
    }
  }
  operator error_t*() {
    return &error.opaque;
  }
private:
  ErrorRAII error{nullptr};
};
```
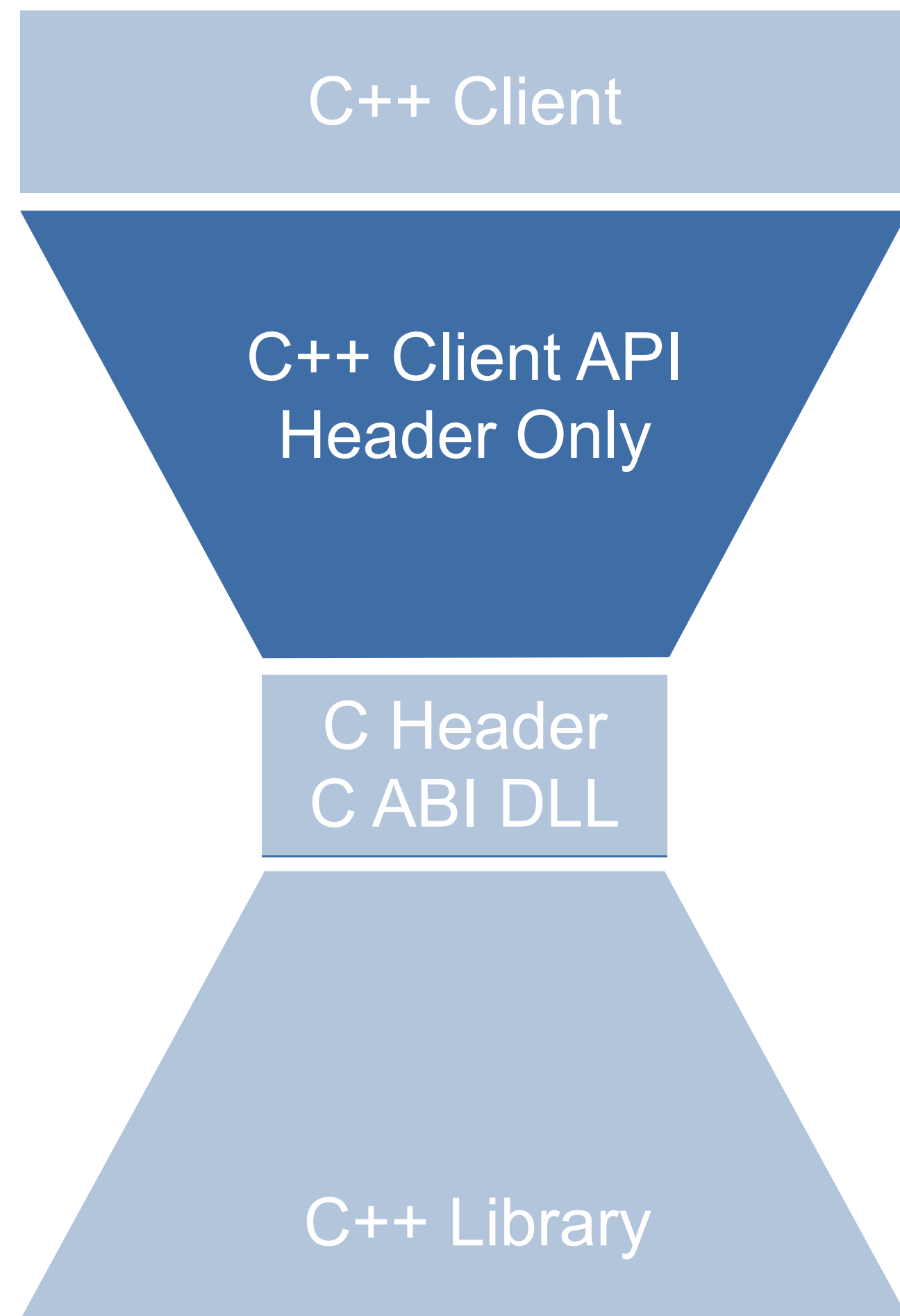
WizardClient.h

```cpp
struct ThrowOnError {
  ThrowOnError() = default;
  ~ThrowOnError() noexcept(false) {
    if (error.opaque) {
      throw std::runtime_error{error_message(error.opaque)};
    }
  }
  operator error_t*() {
    return &error.opaque;
  }
private:
  ErrorRAII error{nullptr};
};

struct Wizard {
  Wizard(std::string const & who = "Rincewind")
      : wiz {createWizard(who.c_str(), ThrowOnError{})} {
  }
  // C linkage trampoline
};
```

- **Here the complete view of the client side Wizard class**

- **Calls "C" functions from global namespace**

  - Namespace prefix needed for synonyms to member functions

- **Header-only**

  - Inline functions delegating

- **Need to take care of passed and returned Pointers, esp. char ***

  - Do not pass/return dangling pointers!

WizardClient.h

```cpp
struct Wizard {
  Wizard(std::string const & who = "Rincewind")
    : wiz {createWizard(who.c_str(), ThrowOnError{})} {
  }
  ~Wizard() {
    disposeWizard(wiz);
  }
  std::string doMagic(std::string const &wish) {
    return ::doMagic(wiz, wish.c_str(), ThrowOnError{});
  }
  void learnSpell(std::string const &spell) {
    ::learnSpell(wiz, spell.c_str());
  }
  void mixAndStorePotion(std::string const & potion) {
    ::mixAndStorePotion(wiz, potion.c_str());
  }
  char const * getName() const {
    return wizardName(wiz);
  }
private:
  Wizard(Wizard const &) = delete;
  Wizard & operator=(Wizard const &) = delete;
  wizard wiz;
};
```

- **With the Gnu compiler (and clang I presume)**

  - `-fvisibility=hidden`

    - Can be added to suppress exporting symbols

    - Must mark exported ABI functions with default visibility

- **Visibility refers to dynamic library/object file export of symbols**

  - Windows: `__declspec(dllexport)`

  - See also hairpoll demo project
    https://youtu.be/PVYdHDm0q6Y

  - For more on gcc visibility (expert-level knowledge): see https://gcc.gnu.org/wiki/Visibility

`WizardClient.h`

```c
#define WIZARD_EXPORT_DLL
                    __attribute__ ((visibility ("default")))

WIZARD_EXPORT_DLL
char const * error_message(error_t error);
WIZARD_EXPORT_DLL
void error_dispose(error_t error);

WIZARD_EXPORT_DLL
wizard createWizard(char const * name,
                        error_t *out_error);
WIZARD_EXPORT_DLL
void disposeWizard(wizard toDispose);

WIZARD_EXPORT_DLL
char const * doMagic(wizard w,
                        char const * wish,
                        error_t *out_error);
WIZARD_EXPORT_DLL
void learnSpell(wizard w, char const *spell);
WIZARD_EXPORT_DLL
void mixAndStorePotion(wizard w, char const *potion);
WIZARD_EXPORT_DLL
char const * wizardName(cwizard w);
```

● **Library API and ABI design can be tricky for third party users**

  ▪ Only really a problem if not in-house or all open source

  ▪ Even with open source libraries, re-compiles can be a burden

    ▪ There are just too many compiler options

    ▪ Plus DLL versioning

● **API stability can be important**

  ▪ PIMPL idiom helps with avoiding client re-compiles (but should be considered a legacy)

  ▪ Not easily applicable with heavily templated code -> that often is header-only

● **ABI stability is even more important when delivering DLLs/shared libraries**

  ▪ Only relevant when not header-only

  ▪ "C" linkage safe, but crippling - Hourglass-Interfaces allow shielding C++ clients from the crippled ABI

    ▪ Still easy to make mistakes (which we always tried to avoid)