

Department I - C Plus Plus

# Modern and Lucid C++ Advanced for Professional Programmers

Week 12 – Hourglass Interfaces

Thomas Corbat / Felix Morgner  
Rapperswil / St. Gallen, 23.05.2024  
FS2024



- **Recap Week 11**
- **Hourglass Interfaces**
- **Java Native Access (JNA)**

- **Participants should ...**

- know how to provide portable APIs for their libraries
- be able to explain how hourglass interfaces work and why they are needed
- be able to call C APIs from Java using JNA

Recap Week 11



- **noexcept aka no-throw**

- Will never-ever throw an exception (and the operation is successful!)

- **Strong exception safety**

- Operation succeeds and doesn't throw, or nothing happens but an exception is thrown (transaction)

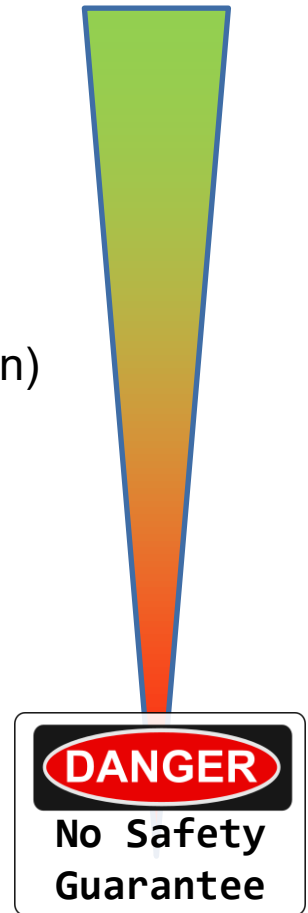
- **Basic exception safety**

- Does not leak resources or garble internal data structures in case of an exception but might be incomplete

- **No guarantee**

- You do not want to go there, undefined behavior and garbled data lurking

- **A function can only be as exception-safe as the weakest sub-function it calls!**



- **noexcept belongs to the function signature**
  - Cannot overload on noexcept
- **noexcept is shorthand for noexcept(true)**
  - noexcept(false) is the default, when no exception specification is given for a function
- **noexcept(expression) can be used to determine the “noexceptiness” of an expression, without actually computing it**
  - noexcept(expression) is true if and only if expression consists only of operations that are noexcept(true)
  - You specify a conditional noexcept as
    - noexcept(noexcept(<expression>))

```
auto function() noexcept -> void {  
    //...  
}  
  
template<typename T>  
auto function(T t) noexcept(<expression>) {  
    //...  
}
```

```
auto main() -> int {  
    std::cout << "is function() noexcept? " <<  
        noexcept(function()) << '\n';  
}
```

- **A function that can handle all argument values of the given parameter types successfully has a “Wide Contract”**
  - It cannot fail
  - It should be specified as `noexcept(true)`
  - `this` is also a parameter
  - Globals and external resources also (heap)
- **A function that has preconditions on its parameters has a narrow contract**
  - I.e., `int` parameter must not be negative
  - I.e., pointer parameter must not be `nullptr`
  - Even if not checked and no exception thrown, those functions should not be `noexcept`
  - This allows later checking and throwing if U.B.

- **Name known (declared) but not the content (structure)**
  - Introduced by a forward declaration
- **Can be used for pointers and references**
  - but not dereference values without definition (access members)
- **C only uses pointers**
  - `void *` is the universally opaque pointer in C
- **`void *` can be cast to any other pointer type**
- **Validity and avoidance of undefined behavior is left to the programmer**
- **Sometimes `std::byte *` is used for memory of a given size (see `BoundedBuffer`)**

```
struct S; //Forward Declaration
auto foo(S & s) -> void {
    foo(s);
    //S s{}; //Invalid
}
struct S{}; //Definition
auto main() -> int {
    S s{};
    foo(s);
}
```

```
template<typename T>
auto makeOpaque(T * ptr) -> void * {
    return ptr;
}
template<typename T>
auto ptrCast(void * p) -> T * {
    return static_cast<T*>(p);
}
auto main() -> int {
    int i{42};
    void * const pi {makeOpaque(&i)};
    cout << *ptrCast<int>(pi) << endl;
}
```

**DANGER**

Unsafe



- Minimal header (Wizard.h)
- All details hidden in implementation (see next slide)
- Delegation to Impl (see `Wizard::doMagic`)

## Wizard.hpp

```
class Wizard {  
    std::shared_ptr<class WizardImpl> pImpl;  
public:  
    Wizard(std::string name = "Rincewind");  
    auto doMagic(std::string wish) -> std::string;  
};
```

## WizardImpl.cpp (Wizard Members)

```
//Implementation of WizardImpl ...  
  
//Implementation of Wizard  
Wizard::Wizard(std::string name):  
    pImpl{std::make_shared<WizardImpl>(name)} {  
}  
  
auto Wizard::doMagic(std::string wish) -> std::string {  
    return pImpl->doMagic(wish);  
}
```

- Define the destructor of Wizard after the definition of WizardImpl

## Wizard.hpp

```
class Wizard {  
    std::unique_ptr<class WizardImpl> pImpl;  
public:  
    Wizard(std::string name);  
    ~Wizard();  
    auto doMagic(std::string wish) -> std::string;  
};
```

## WizardImpl.cpp

```
class WizardImpl {  
    //...  
};  
  
//...  
  
Wizard::~~Wizard() = default;
```

- **How should objects be copied?**

No Copying – Only Moving	<code>std::unique_ptr&lt;class Impl&gt;</code> <ul style="list-style-type: none"><li>• Declare destructor &amp; =default</li><li>• Declare move operations &amp; =default</li></ul>
Shallow Copying (Sharing the implementation)	<code>std::shared_ptr&lt;class Impl&gt;</code>
Deep Copying (Default for C++)	<code>std::unique_ptr&lt;class Impl&gt;</code> <ul style="list-style-type: none"><li>• with DIY copy constructor (use copy constructor of Impl)</li></ul>

- **Can `plmpl == nullptr`?**

- IMHO: never!

- **Can you inherit from PIMPL class?**

- Better don't

# Hourglass Interfaces



- **ABIs define how programs interact on a binary level**

- Names of structures and functions
- Calling conventions
- Instruction sets

- **C++ does not define any specific ABI**

- Because they are tightly coupled to the platform

- **ABIs change between OSes, compiler versions, library versions, etc.**

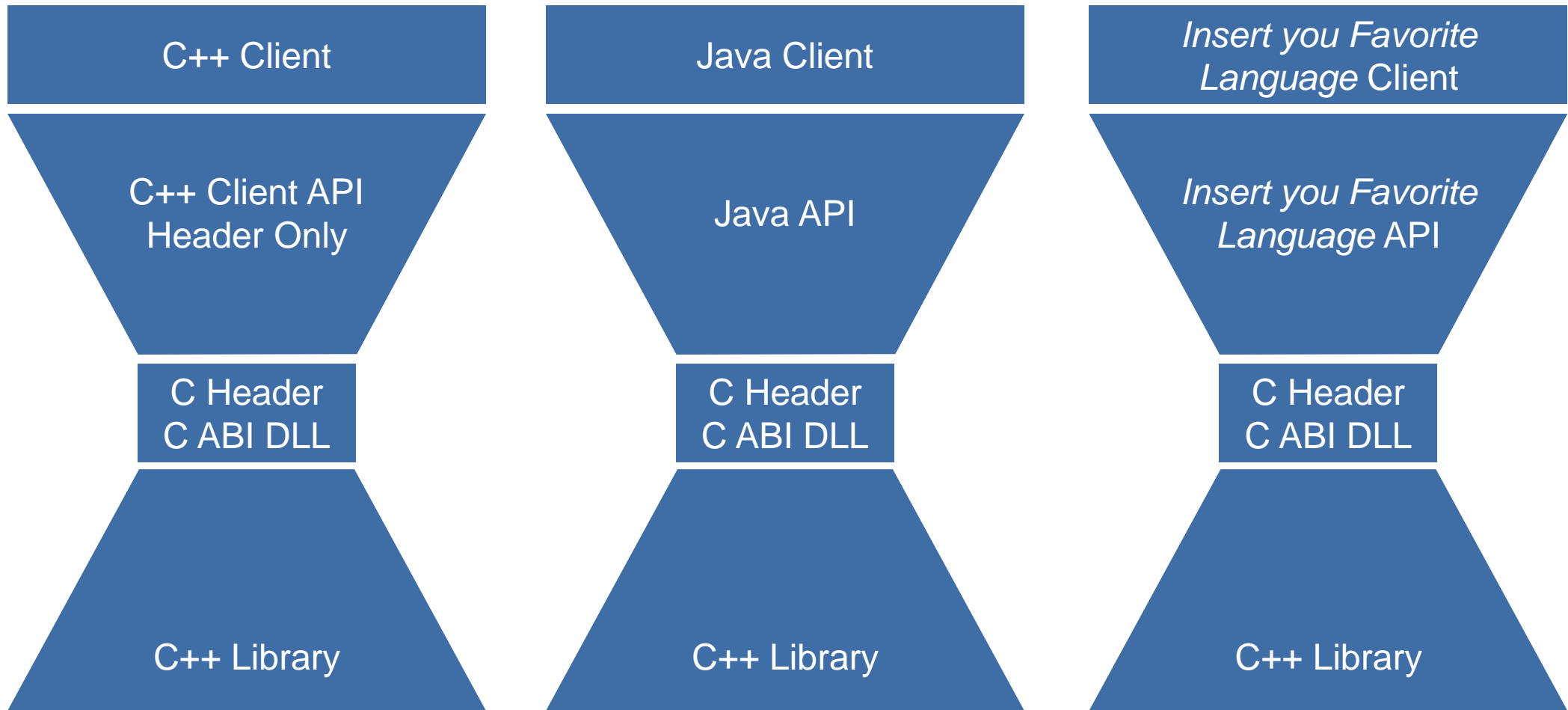
- **Case in point:**

- GCC changed its ABI from:

- Version 2.95 to 3.0
- 3.0 to 3.1
- 3.1 to 3.2
- 3.3 to 3.4
- 5.0 to 5.1
- ...

- Different standard library implementations are usually incompatible

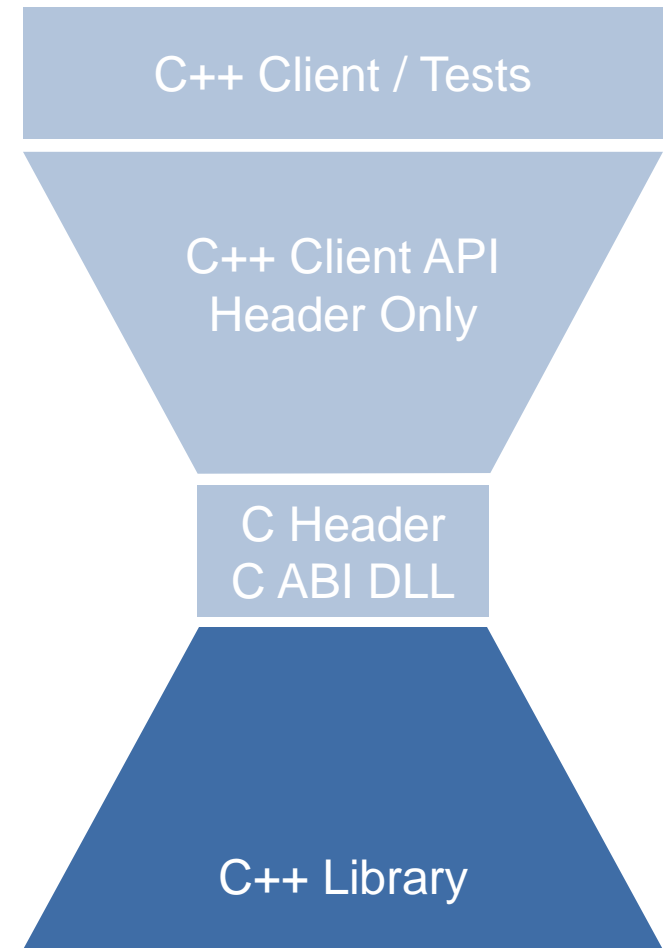
- **Use C (89) as an “intermediate” layer**
  - Think of it as a C frontend for our C++ code
- **While C also does not define, it has an extremely stable ABI**
  - No namespaces
  - No name mangling
- **However, C also has no ...**
  - member functions
  - exceptions
  - templates



## ● Back to our Wizard again

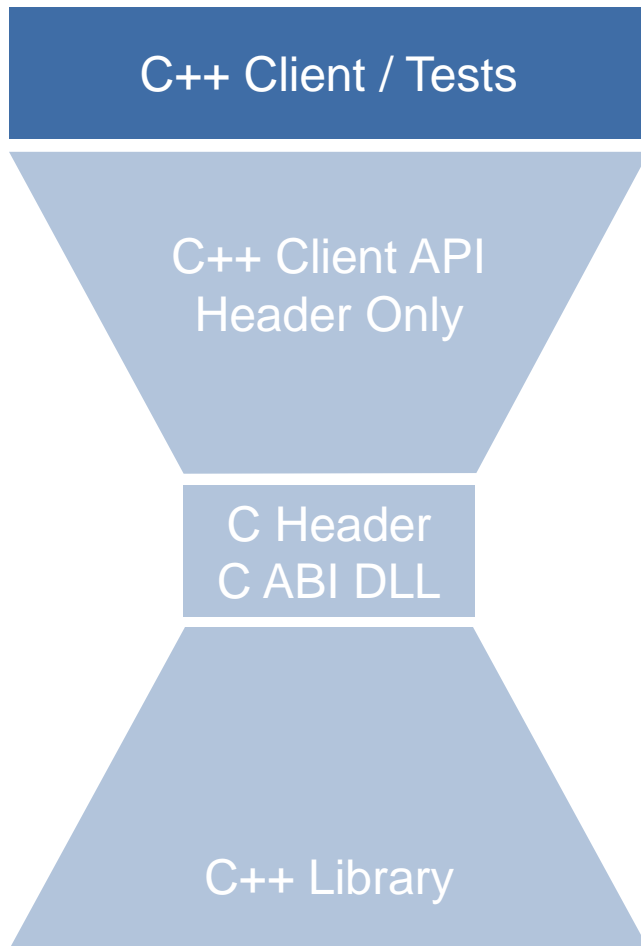
- `doMagic()` – still casts a spell ("wootsh") or uses a potion ("zapp")
- `learnSpell()` – learns a new spell (by name)
- `mixAndStorePotion()` – creates a potion and puts it to the inventory
- `getName()` – function to make Java programmers happy, otherwise there wouldn't be a "getX" function

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





- **Testing a wizard provides the same view a client has**



```
using wizard_client::Wizard;

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

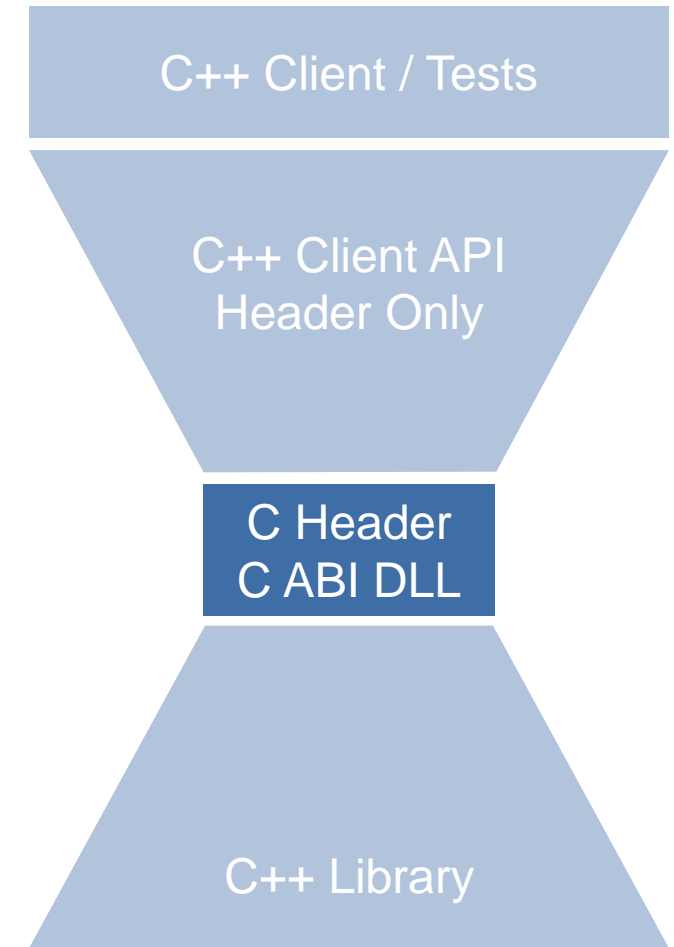
TEST(canCreateWizardWithName) {
    Wizard const magician{ "Petrosilius Zwackelmann" };
    ASSERT_EQUAL("Petrosilius Zwackelmann", magician.getName());
}

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

TEST(wizardMixesPotionAndCanApply) {
    Wizard magician{};
    magician.mixAndStorePotion("Polyjuice Potion");
    ASSERT_EQUAL("zapp", magician.doMagic("Polyjuice Potion"));
}

TEST(unknownMagicFails) {
    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**



- **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

```
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);
```

- **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

```
#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 "trampolin" class
  - It wraps the actual Wizard implementation

## Wizard.cpp

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

## WizardHidden.hpp

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

- 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

```
typedef struct Error * error_t;  
char const * error_message(error_t error);  
void error_dispose(error_t error);  
  
wizard create_wizard(char const * name,  
                    error_t * out_error);
```

## Wizard.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

```
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 create_wizard(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**

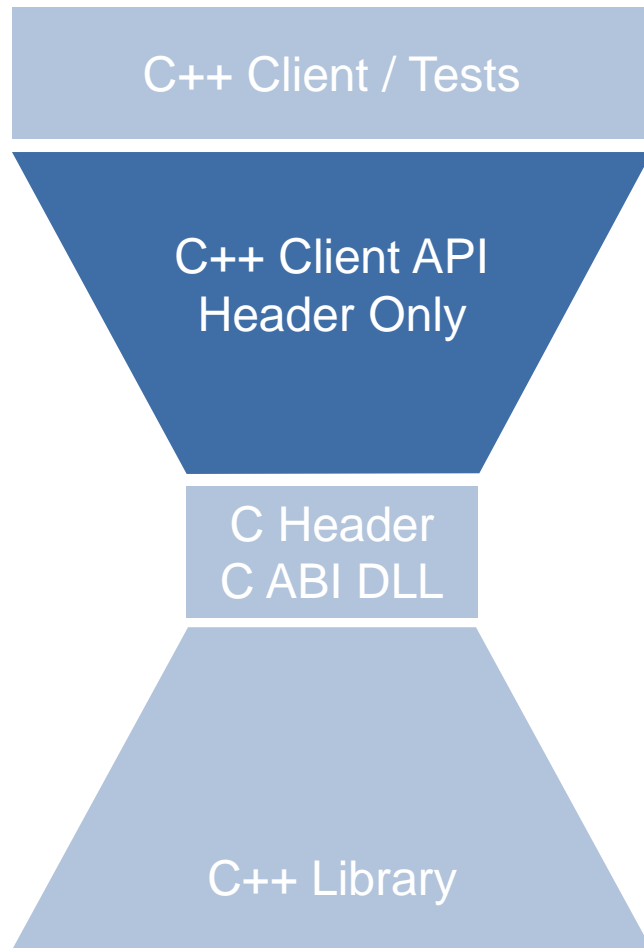
- Strange but possible
- Automatic type conversion passes the address of its guts (opaque)
- Tricky, take care you don't leak when creating the object!

### WizardClient.hpp

```
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.hpp

```
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 {create_wizard(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.hpp

```
struct Wizard {
    Wizard(std::string const & who = "Rincewind")
        : wiz {createWizard(who.c_str(), ThrowOnError{})} {}
    ~Wizard() {
        dispose_wizard(wiz);
    }
    auto doMagic(std::string const &wish) -> std::string {
        return ::do_magic(wiz, wish.c_str(), ThrowOnError{});
    }
    auto learnSpell(std::string const &spell) -> void {
        ::learn_spell(wiz, spell.c_str());
    }
    auto mixAndStorePotion(std::string const & potion) -> void {
        ::mix_and_store_potion(wiz, potion.c_str());
    }
    auto getName() const -> char const * {
        return wizard_name(wiz);
    }
private:
    Wizard(Wizard const &) = delete;
    Wizard & operator=(Wizard const &) = delete;
    wizard wiz;
};
```

- **With the GCC and Clang**

- -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

```
#define WIZARD_EXPORT_DLL [[gnu::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 create_wizard(char const * name,
                    error_t *out_error);
WIZARD_EXPORT_DLL
void dispose_wizard(wizard toDispose);

WIZARD_EXPORT_DLL
char const * do_magic(wizard w,
                     char const * wish,
                     error_t *out_error);

WIZARD_EXPORT_DLL
void learn_spell(wizard w, char const *spell);
WIZARD_EXPORT_DLL
void mix_and_store_potion(wizard w, char const *potion);
WIZARD_EXPORT_DLL
char const * wizard_name(cwizard w);
```

# Java Native Access (JNA)



- **JNA provides a simple interface to C libraries**

- Community Library (not part of the JDK/JRE)
- Based on standard JNI
- Generates “interfaces” at runtime
- <https://github.com/java-native-access/jna>

- **Single JAR file**

- **Cross-platform**

- Windows
- Linux
- macOS

Native Type	Java Type
char	byte
short	short
wchar_t	char
int	int
bool (int)	boolean
long	NativeLong
long long (64-bit)	long
float	float
double	double
char *	String
some_type *	Pointer
struct xyz	Structure

```
public interface CplaLib extends Library {  
    CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);  
}
```

- **Calling the loaded library handle `INSTANCE` is only by convention**
- **The loader searches for a suitable library (`lib<name>.so`, `<libname>.dylib`, `<libname>.dll`)**
  - First in the path specified by `jna.library.path`
  - Otherwise in the system default library search path
  - Fallback into the classpath

```
extern "C" {  
    void printInt(int number);  
}
```



```
public interface CplaLib extends Library {  
    CplaLib INSTANCE = (CplaLib) Native.Load("cpla", CplaLib.class);  
  
    void printInt(int number);  
}
```

- **Function names and parameter types must match**
  - However: The types are not validated! Even at runtime! (They are not part of the signature in C!)
- **Parameter names don't matter**



```
extern "C" {  
  
    struct Point {  
        int x;  
        int y;  
    };  
  
    void printPoint(Point point);  
}
```

```
public interface CplaLib extends Library {
    CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);

    public static class Point extends Structure implements Structure.ByValue {
        public int x, y;

        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        protected List<String> getFieldOrder() {
            return List.of("x", "y");
        }
    }

    void printPoint(Point point);
}
```

```
CplaLib.Point p = new CplaLib.Point(12, 90);  
CplaLib.INSTANCE.printPoint(p);
```

- **Plain (non-opaque) struct types must inherit from Structure**
  - You must override `getFieldOrder()`
  - Can use the tag-interface `Structure.ByValue`
- **You can access pointers to such types using `getPointer()`**
  - However: Remember the GC!

```
extern "C" {  
  
    typedef struct Unicorn * unicorn;  
  
    unicorn createUnicorn(char * name);  
  
    void disposeUnicorn(unicorn instance);  
  
    void printUnicorn(unicorn unicorn);  
  
}
```

```
public interface CplaLib extends Library {
    CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);

    public static class Unicorn extends Pointer {
        Unicorn(String name) {
            super(Pointer.nativeValue(INSTANCE.createUnicorn(name)));
        }

        void dispose() {
            INSTANCE.disposeUnicorn(this);
        }
    }

    Pointer createUnicorn(String name);

    void disposeUnicorn(Unicorn instance);

    void printUnicorn(Unicorn unicorn);
}
```

```
CplaLib.Unicorn u = new CplaLib.Unicorn("freddy");  
CplaLib.INSTANCE.printUnicorn(u);  
u.dispose();
```

- **Opaque struct types should inherit from `Pointer`**
  - Provide a constructor using the `create...()` function
- **Managing lifetime is not trivial**
  - Using `dispose...()` API functions in finalizers is not recommended
  - Either provide a `dispose` method on your Java type
  - Or implement `AutoClosable` and use your objects with `try-with-resources`



```
extern "C" {  
  
    char * getData(int * size);  
  
    void freeData(char * data);  
  
}
```

```
public interface CplaLib extends Library {  
    CplaLib INSTANCE = (CplaLib) Native.load("cpla", CplaLib.class);  
  
    Pointer getData(IntByReference size);  
    void freeData(Pointer data);  
}
```



```
IntByReference size = new IntByReference();  
Pointer data = CplaLib.INSTANCE.getData(size);  
byte[] javaData = data.getByteArray(0, size.getValue());  
CplaLib.INSTANCE.freeData(data);  
  
for(byte b : javaData) {  
    System.out.println(b);  
}
```

- **Use IntByReference to retrieve the size of the buffer**
  - Requires that the API supports it!
- **getBytesArray() copies the data from the buffer**
- **Make sure to free the buffer**
  - Either using an API free...() functions
  - Or `Native.free()`
    - Tends to crash on Windows for some reason

- **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
- **API stability can be important**
  - PIMPL idiom helps with avoiding client re-compiles
  - 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
- **JNA provides a convenient mechanism to work with native code from Java**