

Department I - C Plus Plus

## Modern and Lucid C++ Advanced for Professional Programmers

Week 12 – Advanced Library Design

Prof. Peter Sommerlad / Thomas Corbat  
Rapperswil, 09.05.2019  
FS2019



## Recap Week 11




```
struct ConcurrentCounter {
    void increment() {
        std::scoped_lock lock{m};
        ++value;
    }

    int current() const {
        std::scoped_lock lock{m};
        return value;
    }

private:
    mutable std::mutex m{};
    int value{};
};
```

```
template <typename T, typename MUTEX = std::mutex>
struct ThreadsafeQueue {
    using guard = std::lock_guard<MUTEX>;
    using lock = std::unique_lock<MUTEX>;
    void push(T const & t) {
        guard lk{mx};
        q.push(t);
        notEmpty.notify_one();
    }
    T pop() {
        lock lk{mx};
        notEmpty.wait(lk, [this] { return !q.empty(); });
        T t = q.front();
        q.pop();
        return t;
    }
private:
    mutable MUTEX mx{};
    std::condition_variable notEmpty{};
    std::queue<T> q{};
};
```



- Can everything be used as template argument for `std::atomic<T>`?

- T must be trivially copyable

- Member Operations (all atomic)

<code>void store(T)</code> set the new value	<code>T load()</code> get the current value	<code>T exchange(T)</code> set a new value and get the previous
<code>bool compare_exchange_weak(T &amp; expected, T desired)</code> compare expected with current value, if equal replace the current value with desired, otherwise replace expected with current value. May spuriously fail (even when current value == expected). <code>compare_exchange_strong</code> cannot fail spuriously, but might be slower		

- Specializations like `std::atomic<int>` also provide atomic operators like `++`, `--`, `+=`, etc.

- **In this lecture you should learn...**
  - ... to distinguish between the different exception safety levels
  - ... to decide when a function can be noexcept
  - ... how to hide implementations with the pimpl idiom

# Exception Safety



- **There is code that handles exceptions**
  - Does it handle all possible exceptions?
- **There is code that throws exceptions**
- **There is exception neutral code**
  - Does not throw exceptions
  - Does not catch exceptions
  - It just forwards exceptions thrown in called code
- **Exception neutral code is probably the most common kind you will deal with**
  - Can you neglect exceptions in exception neutral code?

```
void code_that_catches() {  
    try {  
        //...  
    } catch(...) {  
        //...  
    }  
}
```



```
void code_that_is_exception_neutral() {  
    //...  
}
```



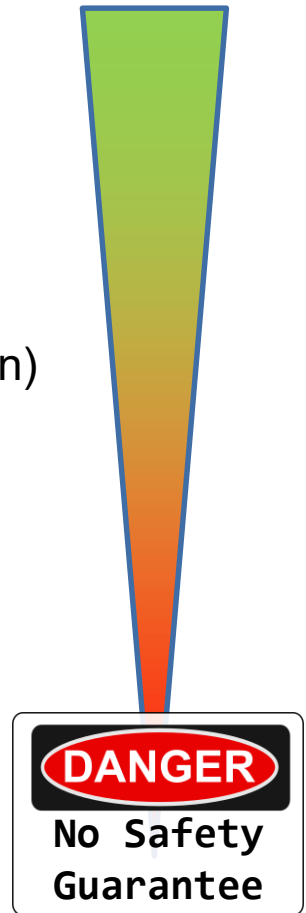
```
void code_that_throws() {  
    //...  
    throw std::some_exception{"what"};  
}
```



- **In generic code that manages resources or data structures**
  - It might call user-defined operations from template arguments explicitly or implicitly
  - It must not garble its data structures
  - It must not leak resources (esp. memory!) – RAII helps
- **Generic code must also be usable to not make user-provided code suffer**
  - Responsibility goes in both directions
- **Deterministic lifetime model of C++ requires it**
  - When an exception is thrown, “stack unwinding” ends the lifetime of temporary and local objects
  - Throwing an exception while another exception is “in flight” in the same thread causes the program to `std::terminate()`
  - Better do not throw on stack unwinding from an exception



- **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!**



- You do not want to go there
- Invalid or corrupted data when an exception is thrown
  - better never catch and let main terminate
  - often unintentional, but happens
  - undefined behavior is lurking
- Very easy to achieve!

```
BoundedBuffer & operator=(BoundedBuffer const & other) {  
    if (m_container != other.m_container) {  
        m_capacity = other.m_capacity;  
        // what if this allocation throws?  
        m_container = new char[sizeof(T) * m_capacity];  
        m_position = 0;  
        m_size = 0;  
  
        for (auto const & element : other){  
            this->push(element); // what if a copy throws?  
        }  
    }  
    return *this;  
}
```



- No resource leaks
- No garbled internal data structure (invariants hold)
- But
  - Operation request could be only half-done

```
template<typename...TYPE>
static BoundedBuffer<value_type> make_buffer(const int size, TYPE&&...param) {
    int const number_of_arguments = sizeof...(TYPE);
    if (number_of_arguments > size)
        throw std::invalid_argument{"Invalid argument"};
    BoundedBuffer<value_type> buffer{size};
    buffer.push_many(std::forward<TYPE>(param)...);
    return buffer;
}
```

Is push\_many() safe?

- **`push()` could fail**

- If in the middle of the pushes no memory is leaked, but the buffer only contains some of the pushed elements

```
void push_many() { }

template<typename FIRST, typename...REST>
void push_many(FIRST && first, REST&&...rest) {
    push(std::forward<FIRST>(first));
    push_many(std::forward<decltype(rest)>(rest)...);
}

void push(value_type const & elem) {
    if(full()) throw std::logic_error{"full"};
    auto pointer = reinterpret_cast<value_type*>(dynamic_container_) + tail_;
    new (pointer) value_type{elem}; // might throw due to copy
    tail_ = (tail_ + 1) % (capacity() + 1);
    elements_++;
}
```

- **Transaction semantic**

- operation succeeds, or
- operation fails with an exception and has no effects

- **Can be hard to achieve**

- when multiple effects have to happen in sequence and something can go wrong in the middle
- doable with 2 effects, when the second one can not throw an exception or when one can undo at least one of the effects

```
BoundedBuffer & operator=(BoundedBuffer const & other) {  
    if (this != &other) {  
        BoundedBuffer copy {other}; // might throw  
        swap(copy); // mustn't throw  
    }  
    return *this;  
}
```

Copy-Swap Idiom

- **A function will never throw an exception**
- **And it will be successful**
  - Any failure is handled internally and compensated for
  - Or no failures are possible
- **How?**
  - Very hard, up to impossible if resource requests are required, i.e., memory allocation
  - Even if it doesn't happen in practical cases, it might happen in theory and in the field
  - All possible argument values must be considered valid (wide contract)

```
bool std::vector<T>::empty() const noexcept;
size_type std::vector<T>::size() const noexcept;
size_type std::vector<T>::capacity() const noexcept;

T * std::vector<T>::data() noexcept;

// all iterator factories begin(), end()...
void std::vector<T>::clear() noexcept;

// but not:
void std::vector<T>::push_back(T const&);
void std::vector<T>::pop_back();
// as well as emplace, insert, resize, erase
void swap(vector&); //until C++17
```

	Invariant OK	All or Nothing	Will Not Throw
No Guarantee	X	X	X
Basic Guarantee	✓	X	X
Strong Guarantee	✓	✓	X
No-Throw Guarantee	✓	✓	✓



- **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>))

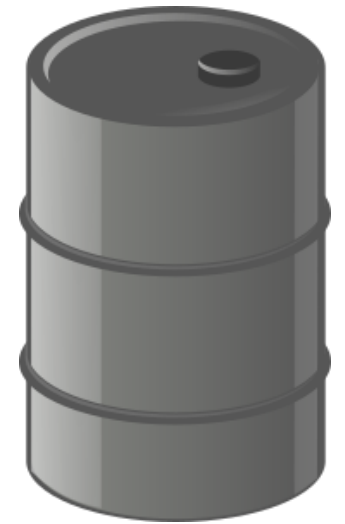
```
void function() noexcept {  
    //...  
}  
  
template<typename T>  
void function(T t) noexcept(<expression>) {  
    //...  
}
```

```
void main() {  
    std::cout << "is function() noexcept? " <<  
        noexcept(function()) << '\n';  
}
```

```
template <unsigned ChanceToExplode>
struct Liquid;

using Nitroglycerin = Liquid<75>;
using JetFuel = Liquid<10>;
using Water = Liquid<0>;

template <typename Liquid>
struct Barrel {
    Barrel(Liquid && content)
        : content{std::move(content)} {
    }
    void poke() noexcept(noexcept(std::declval<Liquid>().shake())) {
        content.shake();
    }
private:
    Liquid content;
};
```



- **Destructors must not throw when used during stack unwinding**
- **Move construction and move assignment better not throw**
- **swap should not throw**
  - `std::swap` requires non-throwing move operations
- **Copying might throw, when memory needs to be allocated**

```
// g++ library std::vector:  
void swap(vector & __x) _GLIBCXX_NOEXCEPT
```

- It may be hard for a library type (container) to implement its move operations correctly if the element type does not support noexcept-move.

- What could we do instead?

- `std::move_if_noexcept`

```
template <typename T>
constexpr typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept(T & x);
```

<code>is_nothrow_constructible</code>	<code>is_nothrow_move_constructible</code>	<code>is_nothrow_move_assignable</code>
<code>is_nothrow_default_constructible</code>	<code>is_nothrow_assignable</code>	<code>is_nothrow_destructible</code>
<code>is_nothrow_copy_constructible</code>	<code>is_nothrow_copy_assignable</code>	<code>is_nothrow_swappable</code>

```
template<typename T>
class _box {
    T value;
public:
    explicit _box(T const &t) noexcept(noexcept(T(t))) :
        value(t) {
    }

    explicit _box(T && t) noexcept(noexcept(T(std::move_if_noexcept(t)))) :
        value(std::move_if_noexcept(t)) {
    }

    T & get() noexcept {
        return value;
    }
};
```

- **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.

- `vector::size()` is `noexcept` as it has a wide contract and cannot fail
- **Constructor of `BoundedBuffer` must not be declared `noexcept`**
  - Exception is thrown if capacity is 0 and allocate might throw

```
// wide contract
size_type size() const _GLIBCXX_NOEXCEPT
{
    return size_type(this->_M_impl._M_finish - this->_M_impl._M_start);
}

// narrow contract:
explicit BoundedBuffer(size_type capacity)
    : startIndex { 0 }, nOfElements { 0 }, capacity { capacity }, values { allocate(capacity) } {
    if (capacity == 0) {
        throw std::invalid_argument { "size must be > 0." };
    }
}
```

- **The compiler might optimize a call of a noexcept function better**
  - It is not required to provide the infrastructure of unwinding the stack properly for the non-existing exception case
- **However, the compiler will not provide an in-depth analysis whether your code adheres to your exception specification**
  - If you throw an exception from a noexcept function (directly or indirectly) `std::terminate()` will be called

```
struct Ball {};  
  
void barrater() noexcept {  
    throw Ball{};  
}  
  
int main() try {  
    barrater();  
} catch(Ball const & b) {  
    std::cout << "caught the ball!";  
}  
}
```

This application has requested the Runtime to terminate it in an unusual way.  
Please contact the application's support team for more information.  
terminate called after throwing an instance of 'Ball'



- A swap operation should be **noexcept**
  - If it is you can rely on it to implement the move constructor

```
BoundedBuffer(BoundedBuffer && other) noexcept :  
    startIndex {0},  
    nOfElements {0},  
    bufferCapacity {0},  
    values_memory {nullptr} {  
    swap(other);  
}
```

```
void swap(BoundedBuffer & other) noexcept {  
    std::swap(startIndex, other.startIndex);  
    std::swap(nOfElements, other.nOfElements);  
    std::swap(bufferCapacity, other.bufferCapacity);  
    std::swap(values_memory, other.values_memory);  
}
```

- See code for P0052R2 standard submission (Eric Niebler: “It is hard...”)

```
unique_resource & operator=(unique_resource && that)
    noexcept(is_nothrow_delete_v &&
        std::is_nothrow_move_assignable<R>::value &&
        std::is_nothrow_move_assignable<D>::value) {
    if(&that == this) return *this;
    reset();
    if (std::is_nothrow_move_assignable<detail::_box<R>>::value) {
        deleter = _move_assign_if_noexcept(that.deleter);
        resource = _move_assign_if_noexcept(that.resource);
    } else if (std::is_nothrow_move_assignable<detail::_box<D>>::value) {
        resource = _move_assign_if_noexcept(that.resource);
        deleter = _move_assign_if_noexcept(that.deleter);
    } else {
        resource = _as_const(that.resource);
        deleter = _as_const(that.deleter);
    }
    execute_on_destruction = std::exchange(that.execute_on_destruction, false);
    return *this;
}
```

- **Exception Safety is an important consideration**
  - Especially when designing generic code
  - Do it consciously
- **Make your Destructor and Move operations `noexcept(true)`**
- **Ensure invariants, even in case of exceptions (basic guarantee)**
- **If really pedantic, rely on `noexcept` expressions to “compute” the `noexcept` value of your functions, if there is a chance that they can be `noexcept(true)`**

# PIMPL Idiom




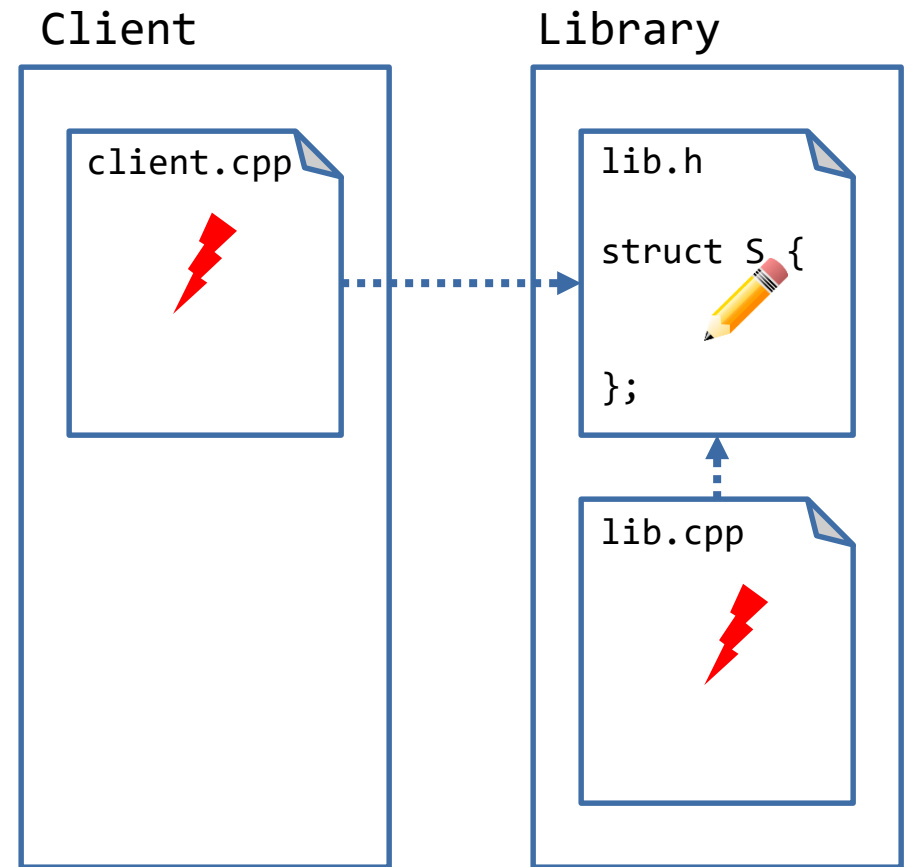
- **Name known (declared) but not the content (structure)**
  - Introduced by a forward declaration
- **Can be used for pointers and references**
  - but not dereference values without definition (access members)
- **C only uses pointers**
  - `void *` is the universally opaque pointer in C
- **`void *` 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
void foo(S & s) {
    foo(s);
    //S s{}; //Invalid
}
struct S{}; //Definition
int main() {
    S s{};
    foo(s);
}
```

**⚠ DANGER****Unsafe**

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

- **Problem: even minor/internal changes in a class' definition require clients to re-compile**
  - E.g. changing a type of a private member variable
- **Compilation "Firewall"**
  - Allow changes to implementation without the need to re-compile users
- **It can be used to shield client code from implementation changes, e.g., when you want to provide a binary library as a DLL/shared library for clients and want to be able to update the library without having the client code to be re-compiled**
  -  You must not change header files your client relies upon
- **Put in the "exported" header file a class consisting of a "Pointer to IMPLementation" plus all public member functions to be used**
- **Read self-study material! ([http://herbsutter.com/gotw/\\_100/](http://herbsutter.com/gotw/_100/))**



.....➔ Dependency (uses)

- All internals and details are exposed to those interacting with class Wizard
- Makes changes hard and will require recompile

Should not be  
shown to "no-majs"

```
class Wizard { // all magic details visible
    std::string name;
    MagicWand wand;
    std::vector<Spell> books;
    std::vector<Potion> potions;
    std::string searchForSpell(std::string const & wish);
    Potion mixPotion(std::string const & recipe);
    void castSpell(Spell spell);
    void applyPotion(Potion phial);
public:
    Wizard(std::string name = "Rincewind") :
        name{name}, wand{} {
    }
    std::string doMagic(std::string const & wish);
    //...
};
```

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

#### `Wizard.h`

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

#### `WizardImpl.cpp` (Wizard Members)

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



- WizardImpl class declaration (in WizardImpl.cpp)

WizardImpl.cpp

```
#include "Wizard.h"
#include "WizardIngredients.h"
#include <vector>
#include <algorithm>

class WizardImpl {
    std::string name;
    MagicWand wand;
    std::vector<Spell> books;
    std::vector<Potion> potions;
    std::string searchForSpell(std::string const & wish);
    Potion mixPotion(std::string const & recipe);
    void castSpell(Spell spell);
    void applyPotion(Potion phial);
public:
    WizardImpl(std::string name) : name{name}, wand{}{}
    std::string doMagic(std::string const & wish);
    //...
};
```

- **WizardImpl implementation**

- in `WizardImpl.cpp`
- Example member function `WizardImpl::doMagic`

`WizardImpl.cpp`

```
std::string WizardImpl::doMagic(std::string const &wish) {  
    auto spell = searchForSpell(wish);  
    if (!spell.empty()) {  
        castSpell(spell);  
        return "wootsh";  
    }  
    auto potion = mixPotion(wish);  
    if (!potion.empty()) {  
        applyPotion(potion);  
        return "zapp";  
    }  
    throw std::logic_error{"magic failed"};  
}
```

- Expected required change?

Wizard.h

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



Wizard.h

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

WizardImpl.cpp

```
//Implementation of Wizard  
Wizard::Wizard(std::string name):  
    pImpl{std::make_shared<WizardImpl>(name)} {  
}
```



WizardImpl.cpp

```
//Implementation of Wizard  
Wizard::Wizard(std::string name):  
    pImpl{std::make_unique<WizardImpl>(name)} {  
}
```

- **Won't compile!**



Compiler says:  
"NO!"

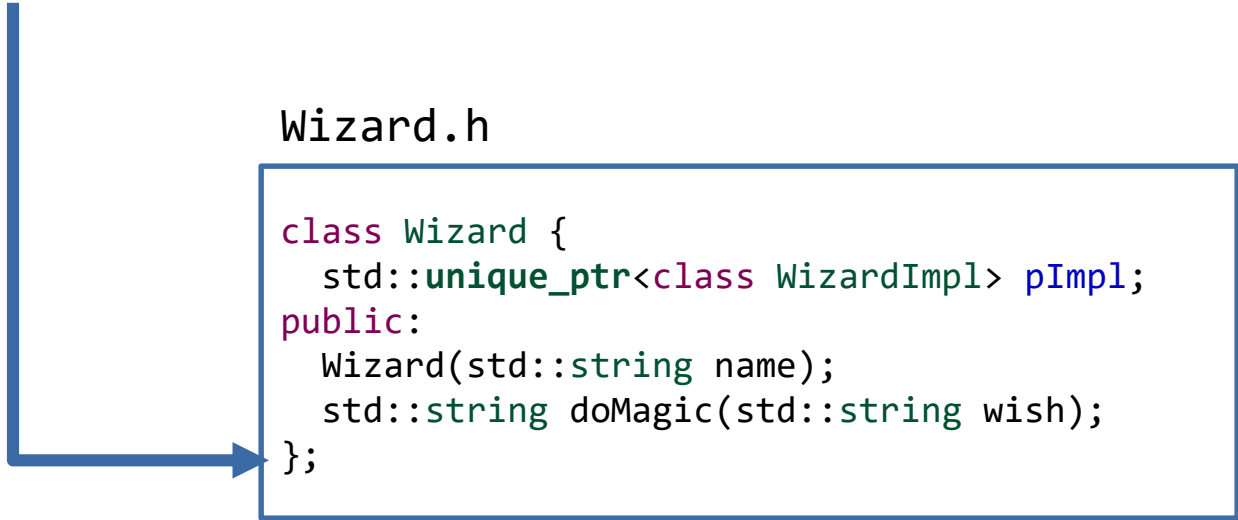
```
.../unique_ptr.h: In instantiation of 'void std::default_delete<_Tp>::operator()(_Tp*) const [with _Tp = WizardImpl]':  
.../unique_ptr.h:239:17:   required from 'std::unique_ptr<_Tp, _Dp>::~~unique_ptr() [with _Tp = WizardImpl; _Dp =  
std::default_delete<WizardImpl>]'  
.../Wizard.h:6:7:   required from here  
.../unique_ptr.h:74:22: error: invalid application of 'sizeof' to incomplete type 'WizardImpl'  
    static_assert(sizeof(_Tp)>0,
```

- **`std::unique_ptr` has 2 template parameters:**
  - pointee type
  - deleter for pointee type
- **The default deleter cannot delete an incomplete type**

- **Definition of implicitly declared Destructor**

- [special]/1 states: ... An implicitly-declared special member function is declared at the closing `}` of the class-specifier.

Wizard.h



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

- **At this point WizardImpl is incomplete**
- **What can we do?**

- Define the destructor of Wizard after the definition of WizardImpl

Wizard.h

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

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

- **Write code that is as exception-safe as possible**
- **In generic code exceptions can occur in code that depends on the template arguments**
- **Lower limit is the basic guarantee, unless it is code you have absolute control of and only you can call it**
- **The Pimpl idiom can be applied to hide implementation details and reduce static dependencies and hide implementations**



What Else?

Not covered in the lecture

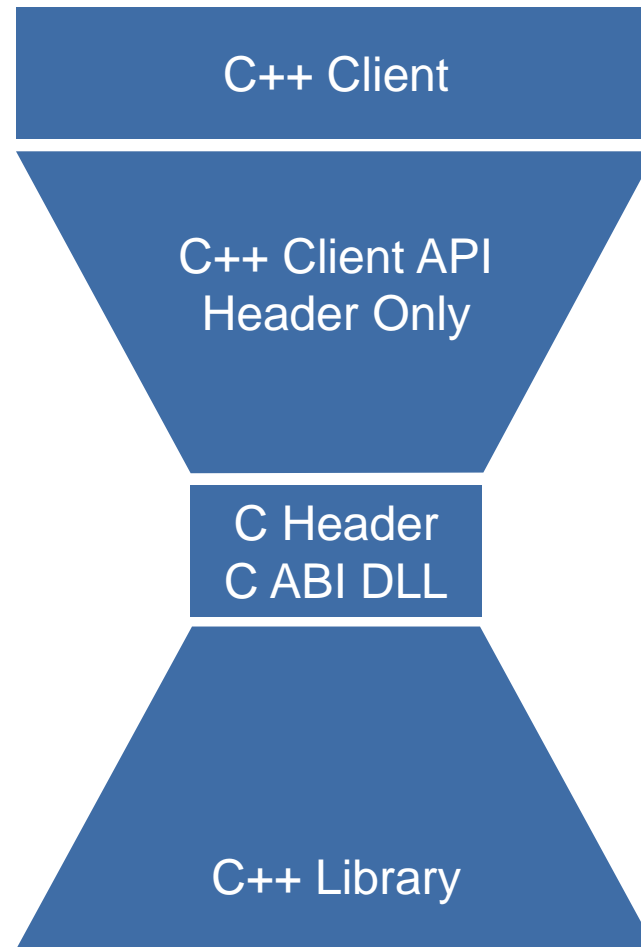


# Hourglass Interfaces



- **DLL APIs work best (and corss platform compatible) with C only**
  - We ignore the Windows burdon 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**

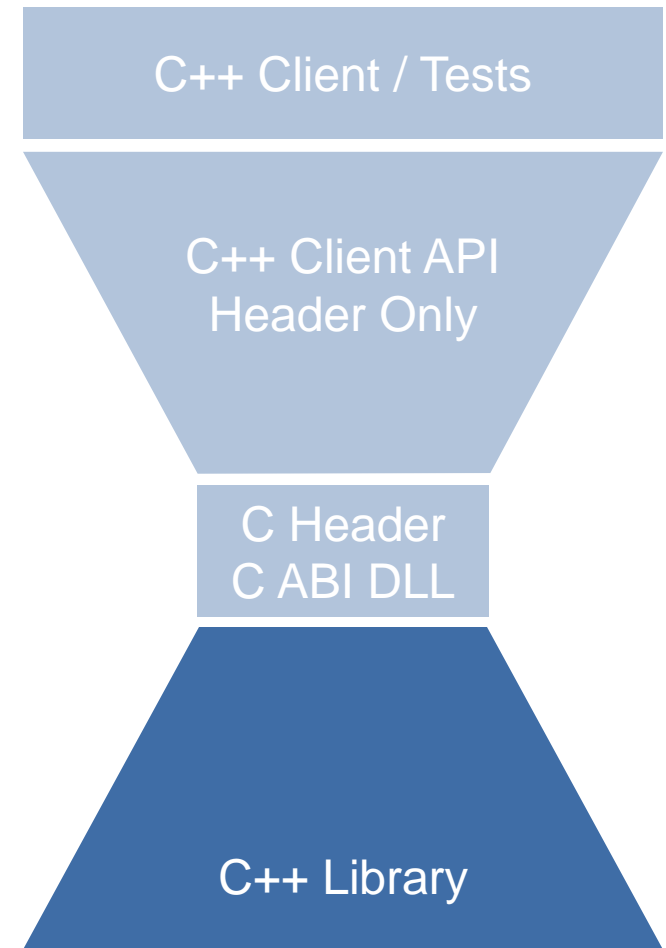
- **Shape of an hourglass**



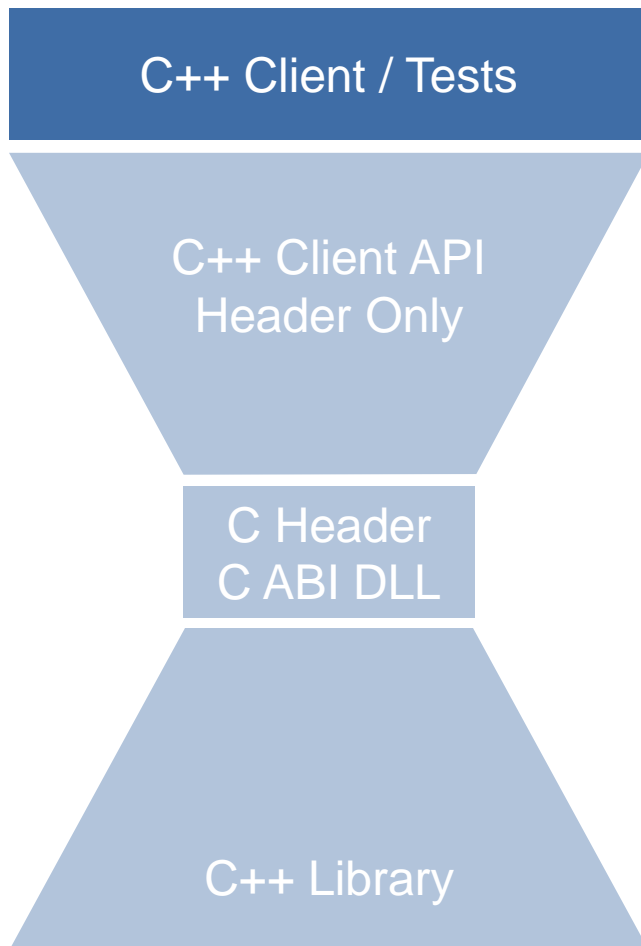
- **Let's add some functionality to our Wizard**

- `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

```
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();  
    }  
};
```



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



```
using wizard_client::Wizard;

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

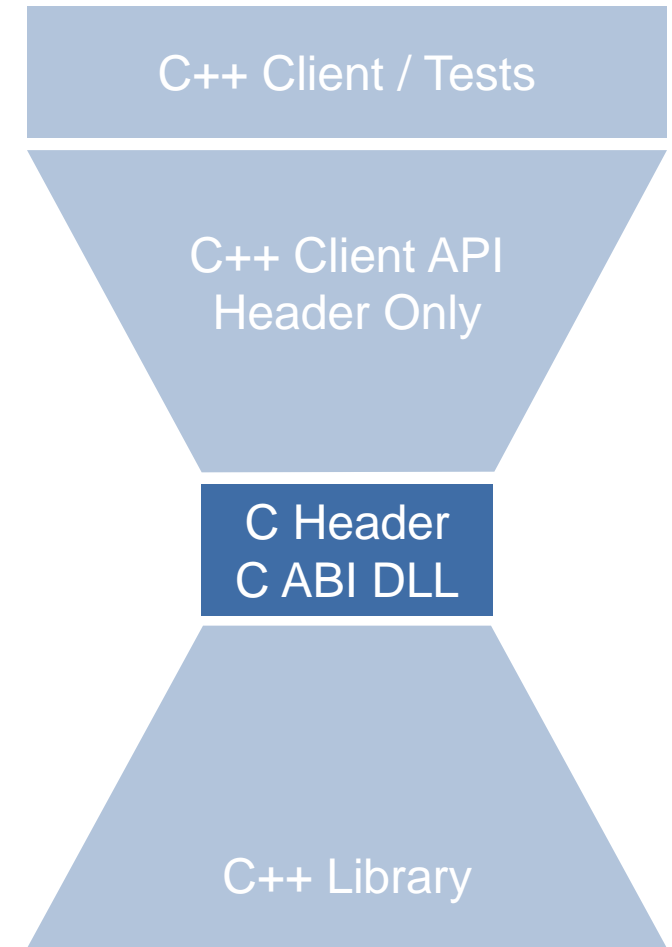
void canCreateWizardWithName() {
    Wizard const magician{ "Petrosilius Zwackelmann" };
    ASSERT_EQUAL("Petrosilius Zwackelmann", magician.getName());
}

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 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.h

```
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();  
    }  
};  
}
```

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

- 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 createWizard(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;
}

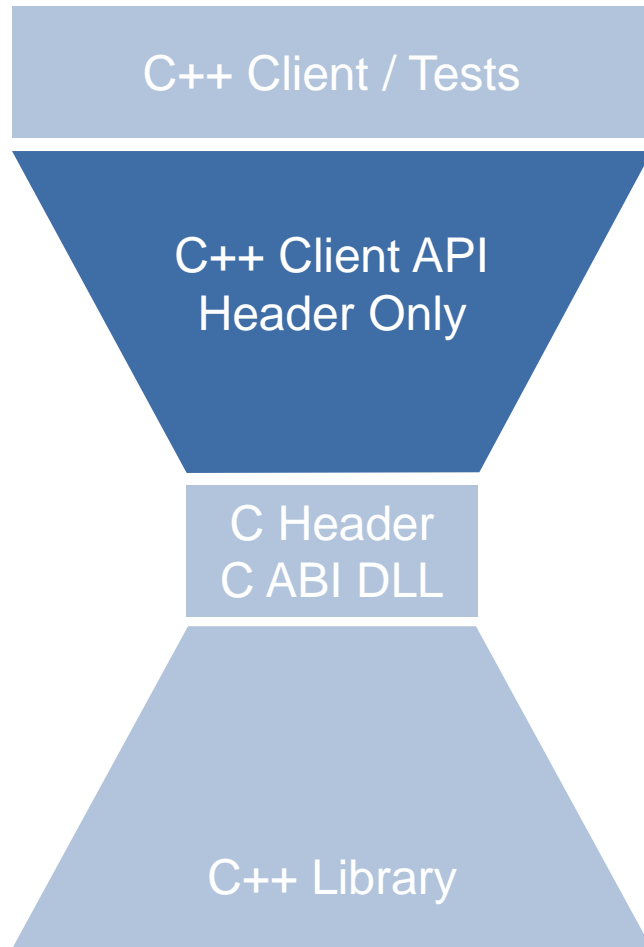
wizard createWizard(const char * name,
                    error_t * out_error) {
    wizard result = nullptr;
    translateExceptions(out_error, [&] {
        result = new Wizard{name};
    });
    return result;
}
```

- **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

```
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

```
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

```
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

```
#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
- 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)