

Department I - C Plus Plus

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

Week 11 – Advanced Library Design

Thomas Corbat / Felix Morgner

Rapperswil / St. Gallen, 16.05.2024

FS2024



- **Recap Week 10**
- **Exception Safety**
- **PIMPL Idiom**

- **Participants should ...**
  - know how to distinguish between the different exception safety levels
  - know how to decide when a function can be noexcept
  - be able to hide implementations with the PIMPL idiom

## Recap Week 10



## • Server



## • Client



- **Transmit / Receive functions need sources or destinations buffers**

- ASIO generally does not manage memory for you!
- Fixed size buffers using `asio::buffer()`
  - Must provide at least as much memory as you would like to read
  - Can use several standard containers as a backend
  - Pointer + Size combinations are also available
- Dynamically sized buffers using `asio::dynamic_buffer()`
  - For use with `std::string` and `std::vector`
- Streambuf buffers using `asio::streambuf`
  - Works with `std::istream` and `std::ostream`

- **asio::read also allows you to specify completion conditions**

- `asio::transfer_all()` – Default behavior, transfer all available data or until the buffer is full
- `asio::transfer_at_least(std::size_t bytes)` – Read at least bytes number of bytes (may transfer more)
- `asio::transfer_exactly(std::size_t bytes)` – Read exactly bytes number of bytes

- **asio::read\_until allows you to specify conditions on the data being read**

- Simple matching of characters or strings
- More complex matching using `std::regex`
- Also allows you to specify a callable object
  - Expects `std::pair<iterator, bool> operator()(iterator begin, iterator end)`
- May read more! You need to work with the number of bytes returned by the call

- **Async read operations**

- `asio::async_read`
- `asio::async_read_until`
- `asio::async_read_at`

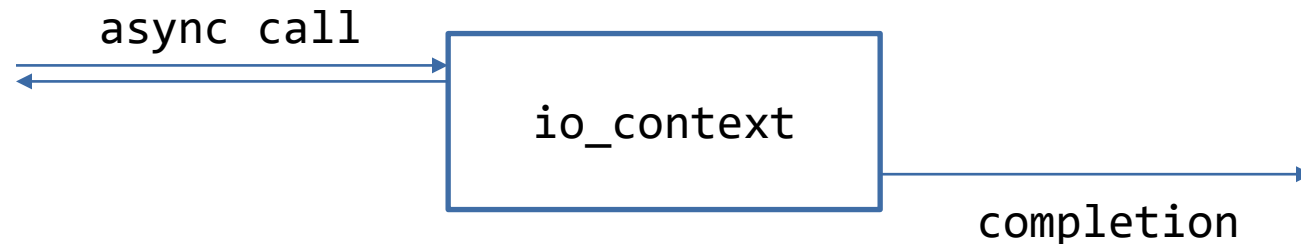
- **Async write operations**

- `asio::async_write`
- `asio::async_write_at`

- **They return immediately**

- **The operation is processed by the executor associated with the stream's `asio::io_context`**

- **A completion handler is called when the operation is done**





- **Constructor**

- Stores the socket with the client connection

- **start() initiates the first async read**

- **read() invokes async reading**

- **write() invokes async writing**

- Called by the handler in read

- **The fields store the data of the session**

- **Why enable\_shared\_from\_this?**

```
struct Session
    : std::enable_shared_from_this<Session> {
    explicit Session(asio::ip::tcp::socket socket);
    auto start() -> void {
        read();
    }

private:
    auto read() -> void;
    auto write(std::string data) -> void;

    asio::streambuf buffer{};
    std::istream input{&buffer};
    asio::ip::tcp::socket socket;
};
```

- **Strands are a mechanism to ensure sequential execution of handlers**

- Implicit Strands

- if only one thread calls `io_context.run()`
- or program logic ensures only one operation is in progress at a time

- Explicit Strands

- Objects of type `asio::strand<...>`
- Created using `asio::make_strand(executor)`
- Or `asio::make_strand(execution_context)`
- Applied to handlers using `asio::bind_executor(strand, handler)`

## 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?

```
auto code_that_catches() -> void {  
    try {  
        //...  
    } catch(...) {  
        //...  
    }  
}
```



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

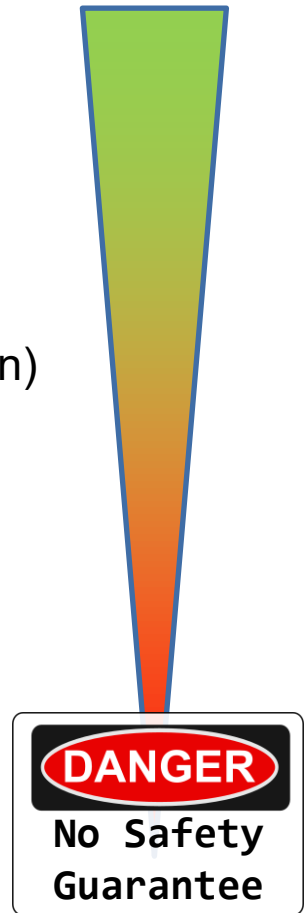


```
auto code_that_throws() -> void {  
    //...  
    throw std::some_exception{"what"};  
}
```

- **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!

```
auto & 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 auto make_buffer(const int size, TYPE&&...param) -> BoundedBuffer<value_type> {
    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

```
auto push_many() -> void { }

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

auto push(value_type const & elem) -> void {
    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

```
auto & 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)

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

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

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

// but not:
auto std::vector<T>::push_back(T const&) -> void;
auto std::vector<T>::pop_back() -> void;
// as well as emplace, insert, resize, erase
auto swap(vector&) -> void; //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>))

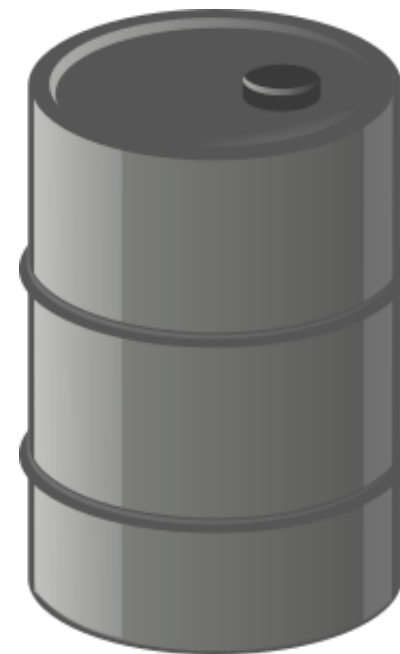
```
auto function() noexcept -> void {  
    //...  
}  
  
template<typename T>  
auto function(T t) noexcept(<expression>) {  
    //...  
}
```

```
auto main() -> int {  
    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)} {
    }
    auto 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:  
auto 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)) {
    }

    auto get() noexcept -> T &{
        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
auto size() const _GLIBCXX_NOEXCEPT -> size_type
{
    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 {};  
  
auto barrater() noexcept -> void {  
    throw Ball{};  
}  
  
auto main() try -> int {  
    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);  
}
```

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

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

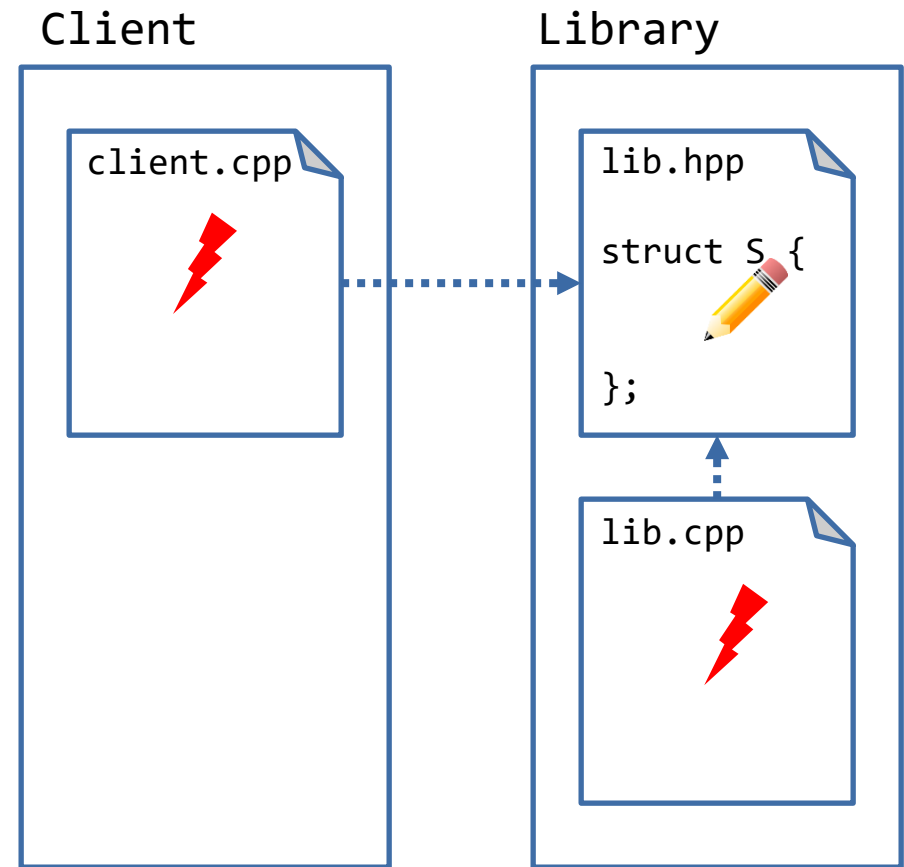
```
template<typename T>
auto makeOpaque(T * ptr) -> void * {
    return ptr;
}
template<typename T>
auto ptrCast(void * p) -> T * {
    return static_cast<T*>(p);
}
auto main() -> int {
    int i{42};
    void * const pi {makeOpaque(&i)};
    cout << *ptrCast<int>(pi) << endl;
}
```

**DANGER**

Unsafe



- **Problem: 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**
  - → You must not change header files your client relies upon
- **Put in the "exported" header file a class consisting of a "Pointer to IMPLementation" + all public members**
- **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 "muggles"

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

- 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);  
}
```

- **WizardImpl class declaration (in WizardImpl.cpp)**

WizardImpl.cpp

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

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

- **WizardImpl implementation**

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

`WizardImpl.cpp`

```
auto WizardImpl::doMagic(std::string const &wish) -> std::string {  
    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.hpp

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



Wizard.hpp

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

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!"

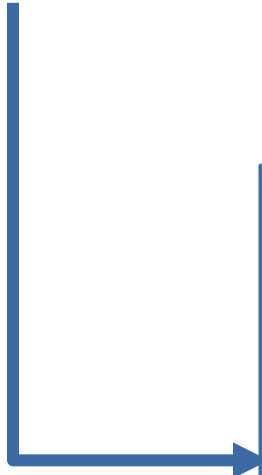
```
.../unique_ptr.h: In instantiation of 'void std::default_delete<_Tp>::operator()(_Tp*) const [with _Tp = WizardImpl]':  
.../unique_ptr.h:239:17:   required from 'std::unique_ptr<_Tp, _Dp>::~~unique_ptr() [with _Tp = WizardImpl; _Dp =  
std::default_delete<WizardImpl>]'  
.../Wizard.h:6:7:   required 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.hpp



```
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.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

- **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**