Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 11 – Advanced Library Design

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Thomas Corbat / Felix Morgner
Rapperswil, 17.05.2020
FS2019

Agenda

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

Goals

- **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

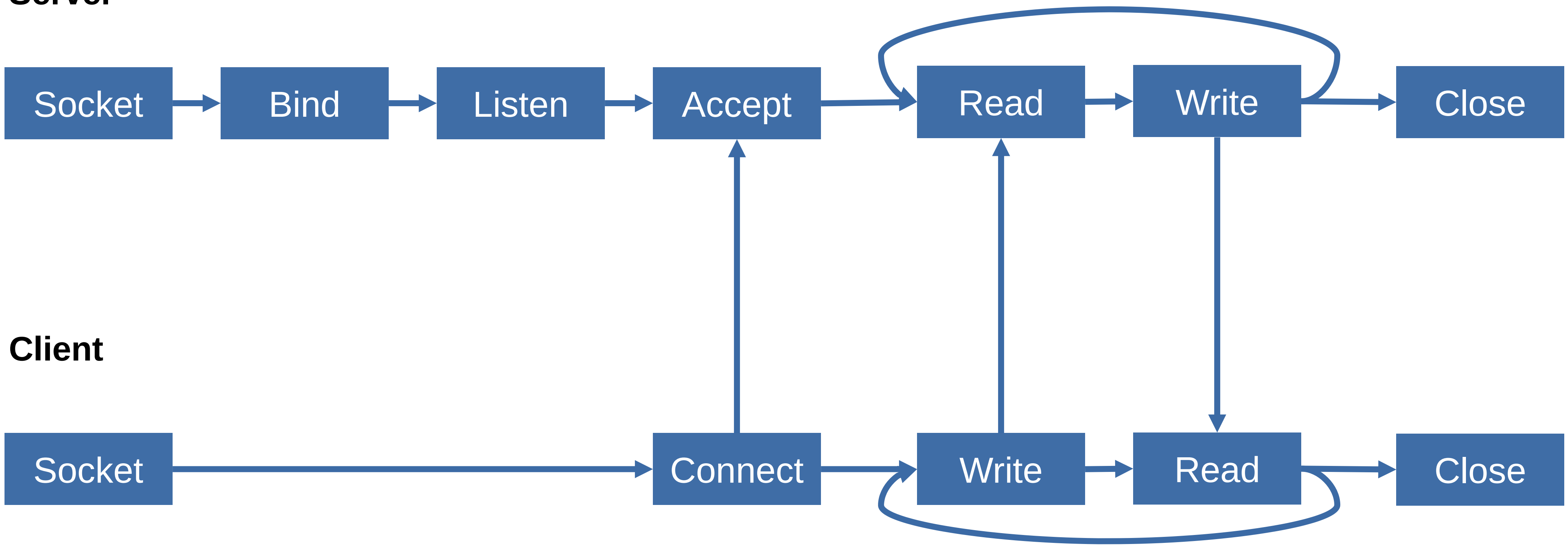
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Connection-Oriented Communication Pattern Using Sockets

- **Server**

- **Client**



Distributed Systems: Principles and Paradigms, 2nd Edition Andrew S. Tanenbaum

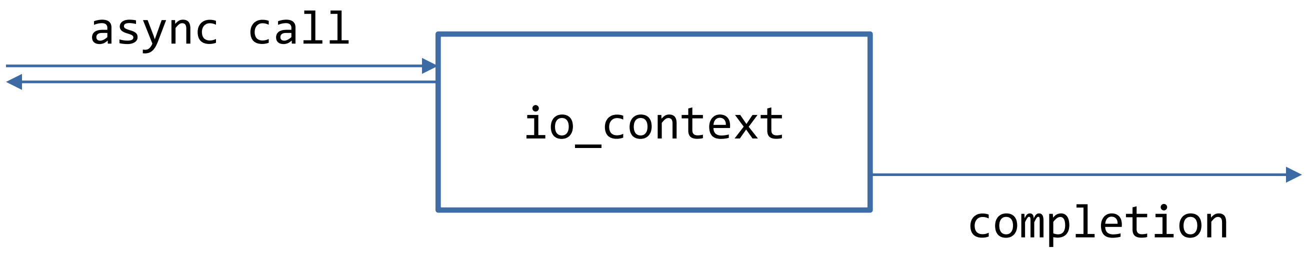
- **Transmit / Receive functions need sources or destinations buffers**
  - ASIO generally does not manage memory for you!
  - Fixed size buffers using `asio::buffer()`
    - Must provide at least as much memory as you would like to read
    - Can use several standard containers as a backend
    - Pointer + 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`

Interlude: Advanced Reading

- **`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

Asynchronous Read/Write on Sockets

- **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 executer associated with the stream's `asio::io_context`**
- **A completion handler is called when the operation is done**



Session with Asynchronous IO

- **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**

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

private:
  void read();
  void write(std::string data);

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

- **Why `enable_shared_from_this`?**

Strands to the Rescue!

- **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

Cevelop

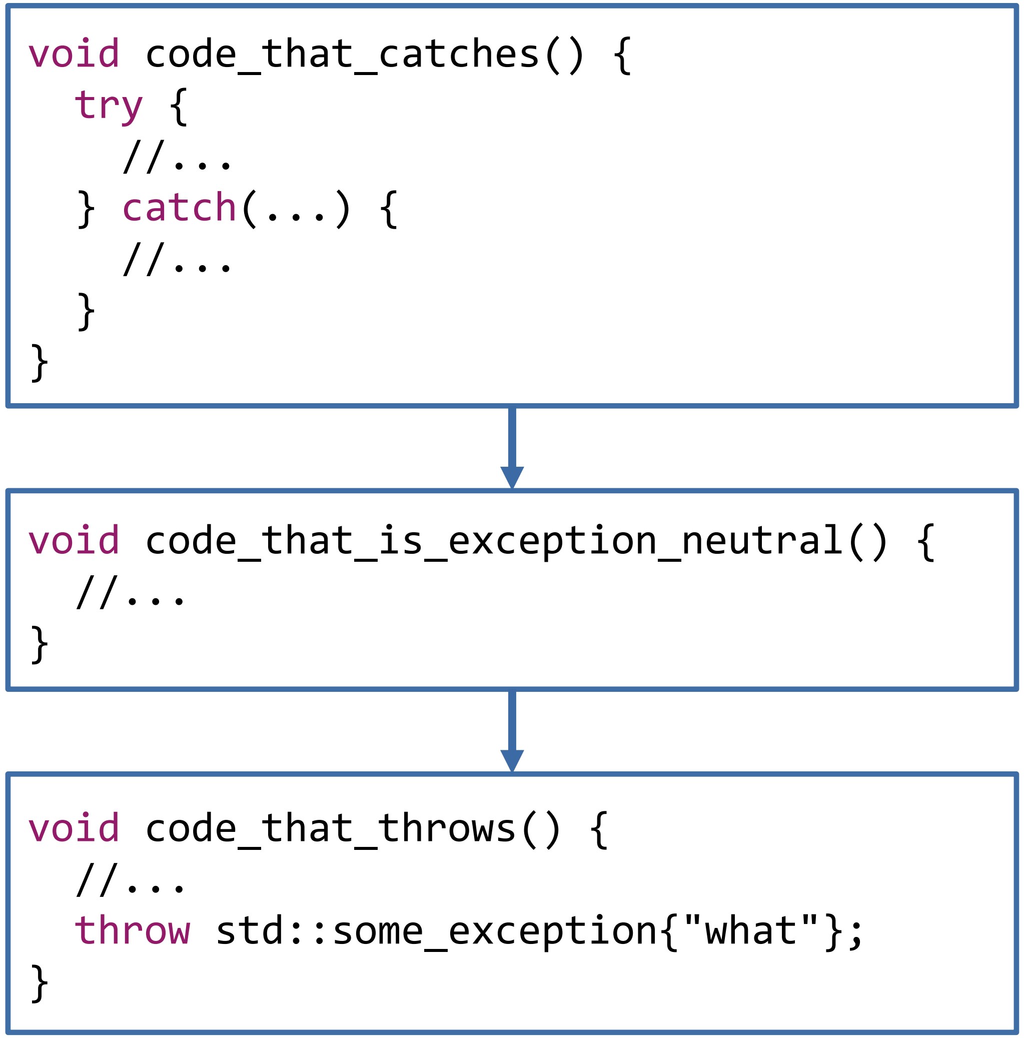
Your C++ deserves it

OST

Ostschweizer
Fachhochschule

Code Impacted by Exceptions

- **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"};
}
```

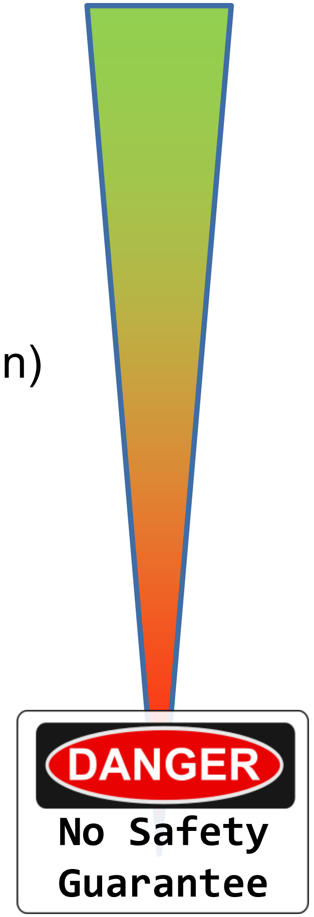
Why Is Exception Safety Important?

- **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

Exception While Unwinding Stack

Exception Safety Levels of an Operation/Function

- **`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!**



Dave Abrahams: http://www.boost.org/community/exception_safety.html

No Guarantee

- **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;
}
```

Basic Guarantee of an Operation

- **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?

Basic Guarantee of an Operation (Bounded Buffer `push_many`)

- **`push()` could fail**
  - If in the middle of the pushs no memory is leaked, but the buffer only contains some of the pushed elements

```cpp
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_++;
}
```

Strong Guarantee

- **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

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

Copy-Swap Idiom

noexcept / No-Throw Guarantee

- **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
```

Overview of Exception Safety Guarantees

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

noexcept Keyword

- **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>) {
  //...
}
```

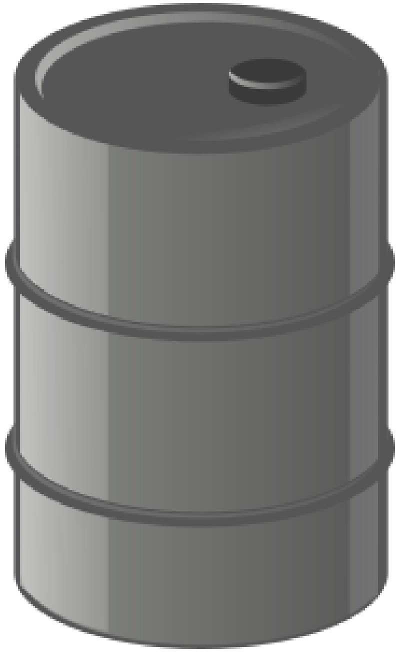
```
void main() {
  std::cout << "is function() noexcept? " <<
    noexcept(function()) << '\n';
}
```

Explosive Example

```cpp
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;
};
```



Member Functions That Should Not Throw

- **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
```

Standard Library Helpers

- **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);
```

is_nothrow_constructible	is_nothrow_move_constructible	is_nothrow_move_assignable
is_nothrow_default_constructible	is_nothrow_assignable	is_nothrow_destructible
is_nothrow_copy_constructible	is_nothrow_copy_assignable	is_nothrow_swappable

http://en.cppreference.com/w/cpp/utility/move_if_noexcept

noexcept Example

```
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;
  }
};
```

Wide and Narrow Contracts

- **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.

Wide and Narrow Contract Example

- **`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." };
  }
}
```

Other Forces of `noexcept`

- **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'
```

Implementing Move Constructors and Assignment Operators (Recap)

- **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);
}
```

Wrap Up

- **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

Cevelop

Your C++ deserves it

OST

Ostschweizer
Fachhochschule

Opaque Types (Incomplete Types)

- **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);
}
```

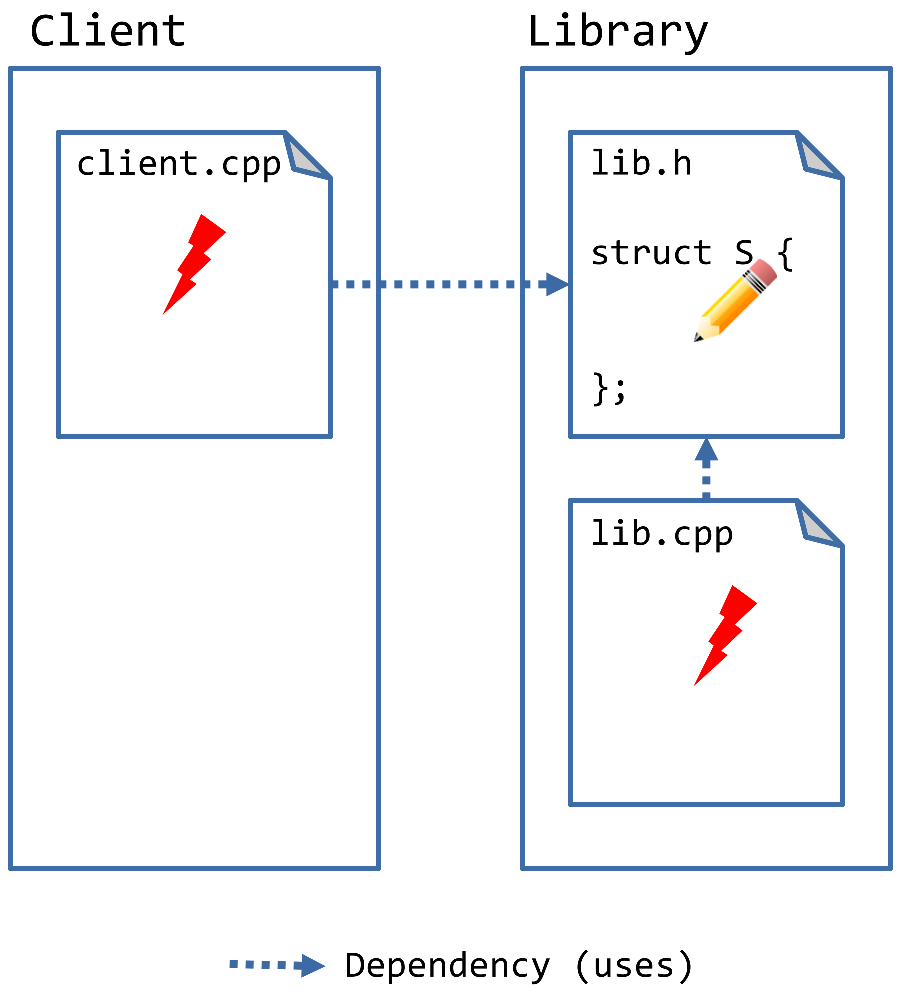
DANGER

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;
}
```

Pointer to IMPLementation Idiom

- **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/)**



PIMPL Example

- **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;
  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);
  //...
};
```

PIMPL with `std::shared_ptr<class Impl>`

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

`Wizard.h`

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

`WizardImpl.cpp (Wizard Members)`

```cpp
//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);
}
```

PIMPL with `std::shared_ptr<class Impl>`

- **`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);
  //...
};
```

PIMPL with `std::shared_ptr<class Impl>`

- **`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"};
}
```

PIMPL with std::unique_ptr<class Impl>

- **Expected required change?**

Wizard.h

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

Wizard.h

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

WizardImpl.cpp

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

WizardImpl.cpp

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

PIMPL with `std::unique_ptr<class Impl>`

- **Won't compile!**

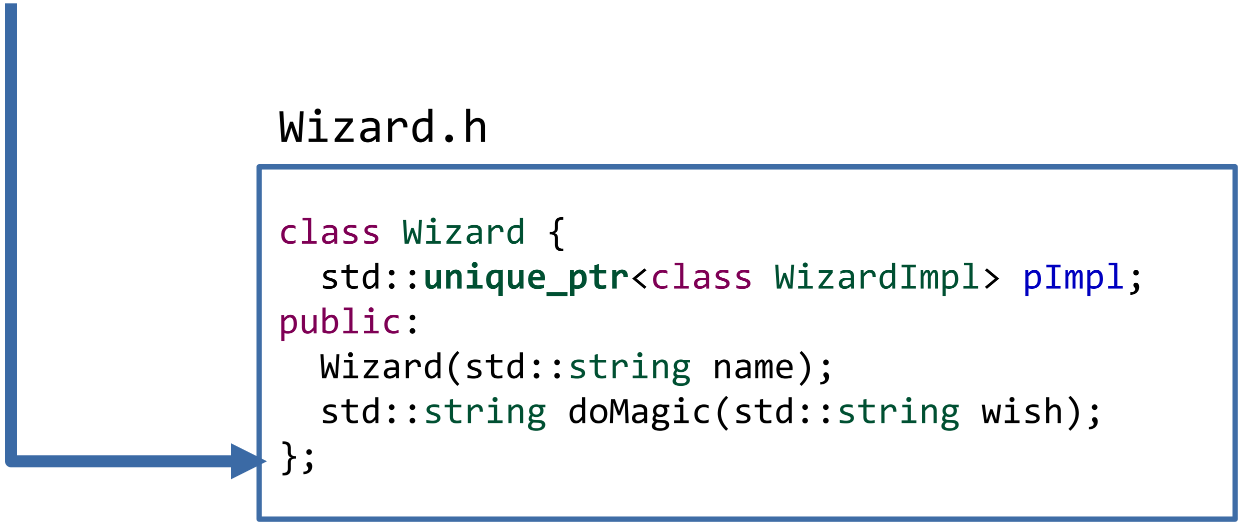
Compiler says:
"NO!"

```
.../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**

PIMPL with `std::unique_ptr<class Impl>`

- **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?**

PIMPL with `std::unique_ptr<class Impl>`

- **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;
```

Design Decisions with PIMPL Idiom

- **How should objects be copied?**

No Copying – Only Moving	`std::unique_ptr<class Impl>` • Declare destructor & `=default` • Declare move operations & `=default`
Shallow Copying (Sharing the implementation)	`std::shared_ptr<class Impl>`
Deep Copying (Default for C++)	`std::unique_ptr<class Impl>` • with DIY copy constructor (use copy constructor of Impl)

- **Can pImpl == nullptr?**
  - IMHO: never!
- **Can you inherit from PIMPL class?**
  - Better don't

Summary

- **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**