std::cout <<
std::vector<
for (auto i: myVec) std::cout <<
std::cout << "\n\n";
std::vector<int> myVec2(20);
std::iota(myVec2.begin(),myVec2.e
std::cout << "myVec2: ";

Smart Pointers

Back to Basics

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.net

Smart Pointer

- A First Overview
- `std::unique_ptr` – Exclusive Ownership
- `std::shared_ptr` – Shared Ownership
- `std::weak_ptr` – Break of Cyclic References
- Performance
- Concurrency
- Function Arguments and Return Values

Smart Pointer

- A First Overview
- `std::unique_ptr` – Exclusive Ownership
- `std::shared_ptr` – Shared Ownership
- `std::weak_ptr` – Break of Cyclic References
- Performance
- Concurrency
- Function Arguments and Return Values

Overview

Smart pointers automatically manage the lifetime of its resource.

- Smart Pointers
    - Allocate und deallocate their resource in the constructor and destructor according to the RAII idiom (**R**esource **A**cquisition **I**s **I**nitialization)
    - Support automatic memory management with reference counting
    - Are C++ answer to garbage collection
    - Release the resource if the smart pointer goes out of scope
    - Are available in four versions

raii.cpp

Overview

Name	C++ Standard	Description
`std::auto_ptr` STOP	C++98	▪ Owns the resource exclusively ▪ „Moves“ its resource during a copy operation
`std::unique_ptr`	C++11	▪ Owns the resource exclusively ▪ Can not be copied ▪ Deals with non-copy objects
`std::shared_ptr`	C++11	▪ Shares a resource ▪ Supports an reference counter to the shared resource and manages it ▪ Deletes the resource if the reference counter becomes 0
`std::weak_ptr`	C++11	▪ Borrows the resource ▪ Helps to break cyclic references ▪ Doesn't change the reference counter

Smart Pointer

A First Overview

`std::unique_ptr` – Exclusive Ownership

`std::shared_ptr` – Shared Ownership

`std::weak_ptr` – Break of Cyclic References

Performance

Concurrency

Function Arguments and Return Values

std::unique_ptr

The `std::unique_ptr` exclusively manages the lifetime of its resource.

- `std::unique_ptr`
  - Is the replacement for the *deprecated* Smart Pointers `std::auto_ptr`

    ⇒ `std::unique_ptr` doesn't support copy semantic
  - Can be used in the containers and algorithms of the STL

    ⇒ Containers and algorithms can not use copy semantic
  - Has no overhead in space and time compared to a raw pointer
  - Can be parametrized with a `deleter`: `std::unique_ptr<T, Deleter>`
  - Can be specialized for arrays: `std::unique_ptr<T[]>`

std::unique_ptr

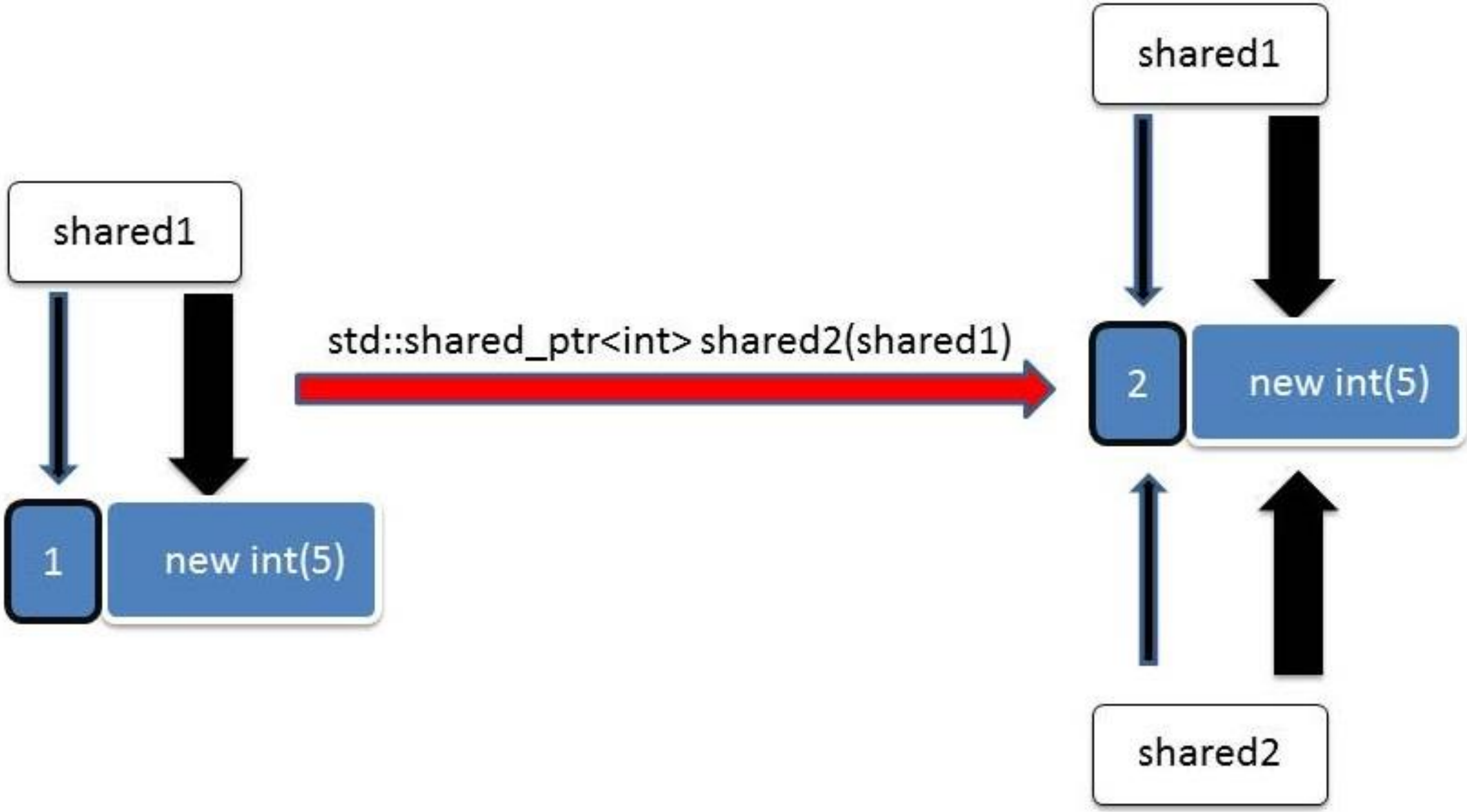
Function	Description
`uniq.release()`	Returns a pointer to the resource and releases it
`uniq.get()`	Returns a pointer to the resource
`uniq.reset(ptr)`	▪ Resets the resource to a new one ▪ Deletes the old resource
`uniq.get_deleter()`	Returns the deleter
`std::make_unique(....)`	Creates the resource and wraps it in a `std::unique_ptr`

uniquePtr.cpp

Smart Pointer

- A First Overview
- `std::unique_ptr` – Exclusive Ownership
- `std::shared_ptr` – Shared Ownership
- `std::weak_ptr` – Break of Cyclic References
- Performance
- Concurrency
- Function Arguments and Return Values

std::shared_ptr



`std::shared_ptr`

`std::shared_ptr` shares a resource and manages its lifetime.

- `std::shared_ptr`
  - Has a reference to the resource and the reference counter
  - Its C++ answer to garbage collection
  - Has *more/less* overhead in time and space such as a raw pointer
  - Deletes the resource
  - Can have a given deleter
    - `shared_ptr<int> shPtr(new int, Del());.`

  - The access to the control block of the `std::shared_ptr` is thread-safe.

std::shared_ptr

Function	Description
sha.unique()	Checks if the std:shared_ptr is the unique owner of the resource
sha.use_count()	Returns the value of the reference counter
sha.get()	Returns a pointer to the resource
sha.reset(ptr)	▪ Resets the resource ▪ Deletes eventually the resource
sha.get_deleter()	Returns the deleter
std::make_shared(....)	Creates the resource and wraps it in a std::shared_ptr

sharedPtr.cpp
sharedPtrDeleter.cpp

Smart Pointer

A First Overview

`std::unique_ptr` – Exclusive Ownership

`std::shared_ptr` – Shared Ownership

`std::weak_ptr` – Break of Cyclic References

Performance

Concurrency

Function Arguments and Return Values

`std::weak_ptr`

`std::weak_ptr` is not a classic smart pointer.

- `std::weak_ptr`
  - Owns no resource
  - Borrows the resource from a `std::shared_ptr`
  - Can not access the resource
  - Can create a `std::shared_ptr` to the resource

The `std::weak_ptr` doesn't change the reference counter

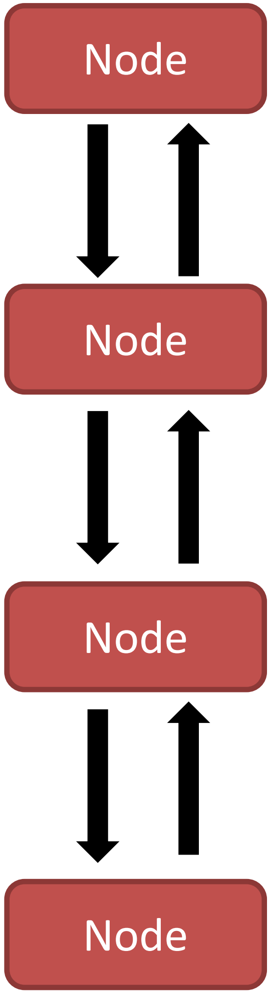
➡ Helps to break cycles of `std::shared_ptr`

std::weak_ptr

Function	Description
`wea.expired()`	Checks if the resource exists
`wea.use_count()`	Returns the value of the reference counter
`wea.lock()`	Creates a `std::shared_ptr` to the resource if available
`wea.reset()`	Releases the resource

weakPtr.cpp

Cyclic References



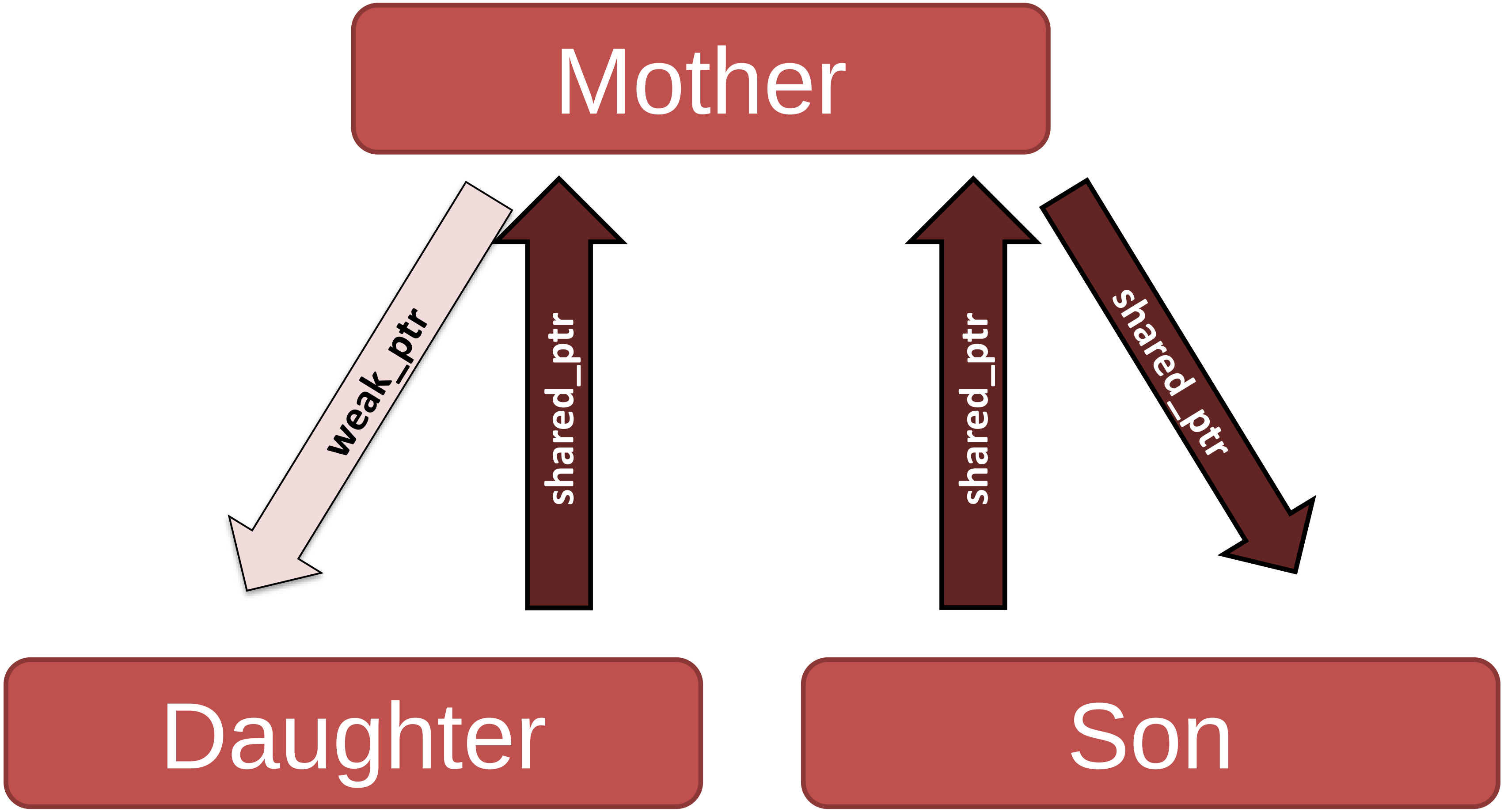
Classic problem

- If `std::shared_ptr` builds a cycle, no `std::shared_ptr` will be deleted automatically

Rescue:

- `std:weak_ptr` breaks the cycle

Cyclic References



cyclicReferences.cpp

Smart Pointers

- A First Overview
- `std::unique_ptr` – Exclusive Ownership
- `std::shared_ptr` – Shared Ownership
- `std::weak_ptr` – Break of Cyclic References
- **Performance**
- Concurrency
- Function Arguments and Return Values

Performance Comparison

```
std::chrono::duration<double> st = std::chrono::system_clock::now();
for (long long i = 0 ; i < 100000000; ++i) {
    int* tmp(new int(i));
    delete tmp;
    // std::unique_ptr<int> tmp(new int(i));
    // std::unique_ptr<int> tmp = std::make_unique<int>(i);
    // std::shared_ptr<int> tmp(new int(i));
    // std::shared_ptr<int> tmp = std::make_shared<int>(i);
}
auto dur=std::chrono::system_clock::now() - st();
std::cout << dur.count();
```

Pointer	Time	Available Since
new	2.93 s	C++98
std::unique_ptr	2.96 s	C++11
std::make_unique	2.84 s	C++14
std::shared_ptr	6.00 s	C++11
std::make_shared	3.40 s	C++11

Smart Pointer

- Further information:
  - std::unique_ptr
  - std::shared_pr
  - std::weak_ptr

Smart Pointer

A First Overview

`std::unique_ptr` – Exclusive Ownership

`std::shared_ptr` – Shared Ownership

`std::weak_ptr` – Break of Cyclic References

Performance

Concurrency

Function Arguments and Return Values

Concurrency

The management of the control block of a `std::shared_ptr` is thread-safe but not the access to the shared resource.

To share ownership between unrelated threads use a `std::shared_ptr`.

`threadSharesOwnershipSmartPtr.cpp`

Concurrency

`std::shared_ptr` contradiction in modern C++:

Use smart pointers but don‘t share.

Forget what you learned in Kindergarten.

Stop sharing. (Tony van Eerd)

Solution:

- C++11: Atomic operations for `std::shared_ptr`
- C++20: Atomic shared pointers
  - `std::atomic_shared_ptr`
  - `std::atomic_weak_ptr`

Atomic Smart Pointers

Atomic smart pointers are part of the C++20 standard.

- Partial specialization of `std::atomic`

- `std::atomic_shared_ptr`

  ➡ `std::atomic<std::shared_ptr<T>>`

- `std::atomic_weak_ptr`

  ➡ `std::atomic<std::weak_ptr<T>>`

Smart Pointer

- Further information:
  - std::atomic
  - std::atomic<std::shared_ptr>
  - std::atomic<std::weak_ptr>

Smart Pointer

A First Overview

`std::unique_ptr` – Exclusive Ownership

`std::shared_ptr` – Shared Ownership

`std::weak_ptr` – Break of Cyclic References

Performance

Concurrency

Function Arguments and Return Values

Functions

Ownership semantic for function parameters

Function Signature	`Ownership Semantic`
`func(value)`	▪ Is an independent owner of the resource ▪ Deletes the resource automatically at the end of `func`
`func(pointer*)`	▪ Borrows the resource ▪ The resource could be empty ▪ Must not delete the resource
`func(reference&)`	▪ Borrows the resource ▪ The resource could not be empty ▪ Must not delete the resource
`func(std::unique_ptr)`	▪ Is an independent owner of the resource ▪ Deletes the resource automatically at the end of `func`
`func(shared_ptr)`	▪ Is a shared owner of the resource ▪ May delete the resource at the end of `func`

Smart Pointer as Parameter

Function Signature	Semantic
`func(std::unique_ptr<int>)`	`func` takes ownership
`func(std::unique_ptr<int>&)`	`func` might reseat `int`
`func(std::shared_ptr<int>)`	`func` shares ownership
`func(std::shared_ptr<int>&)`	`func` might reseat `int`
`func(const std::shared_ptr<int>&)`	`func` might retain a reference counter

- `func(const std::shared_ptr<int>&)`
  - Adds no value to a raw pointer or a reference

Factory Method

```
int main() {
    const Window* window = createWindow(Window::Default);
}
```

Open question with a pointer interface:

- Who is the owner of the `window`?
- Who releases the resource?

Use smart pointers

- `std::unique_ptr`: exclusive ownership
- `std::shared_ptr`: shared ownership

Smart Pointer

A First Overview

`std::unique_ptr` – Exclusive Ownership

`std::shared_ptr` – Shared Ownership

`std::weak_ptr` – Break of Cyclic References

Performance

Concurrency

Function Arguments and Return Values