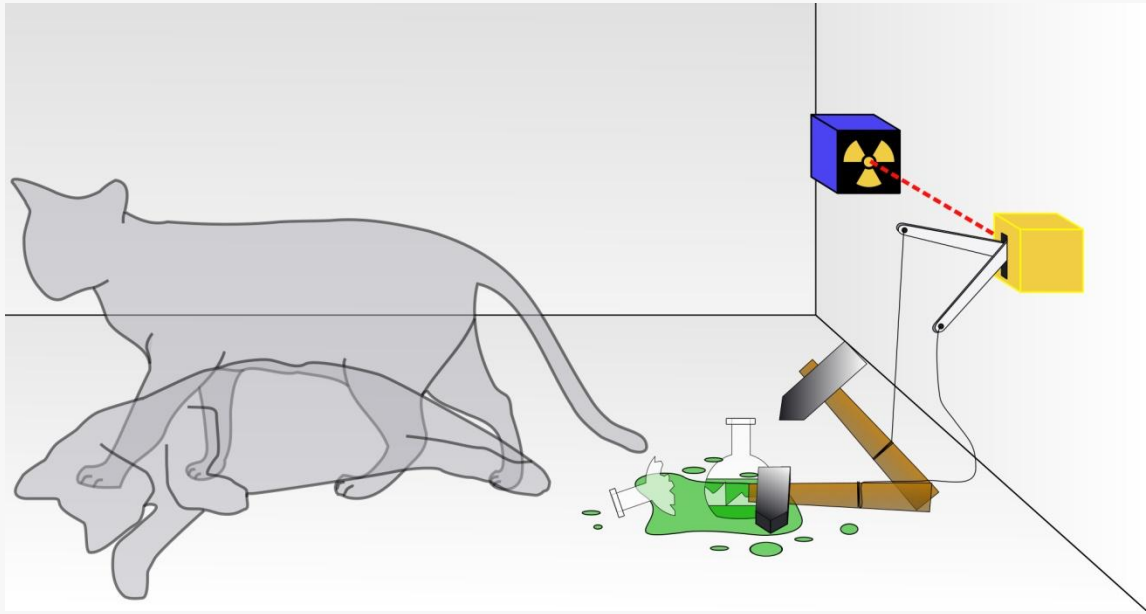
The Upcoming Concurrency TS Version 2 for Low-Latency and Lockless Synchronization

MAGED MICHAEL, MICHAEL WONG & PAUL MCKENNEY

Agenda

1. Don’t we already have a Concurrency TS?
   - Why do we need a new one?
   - implementation status
2. TS2 Hazard Pointer
   - how I learn to love C++ tricks
3. TS2 RCU
   - From C to C++ in 2500 days

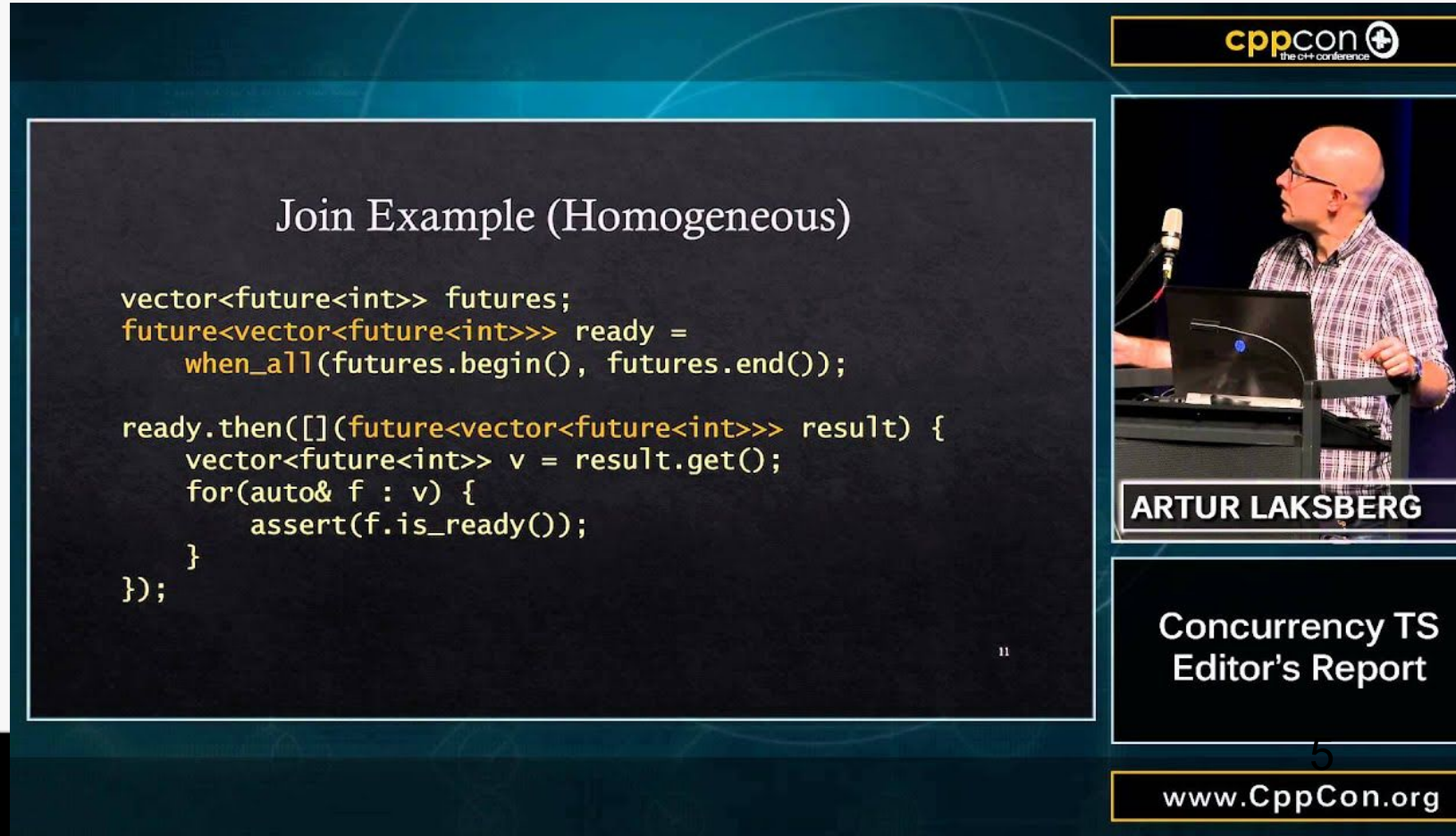


Concurrency TS1: Don’t we already have a TS?

- Produced in 2015
- Produced by the Concurrency Study Group (SG1) with input from LEWG, LWG
- Separate document and is not part of ISO C++ Standard
- Goal: Eventual Inclusion into ISO C++ Standard
- Available online: http://wg21.link/n4538
- github : https://github.com/cplusplus/concurrency-ts

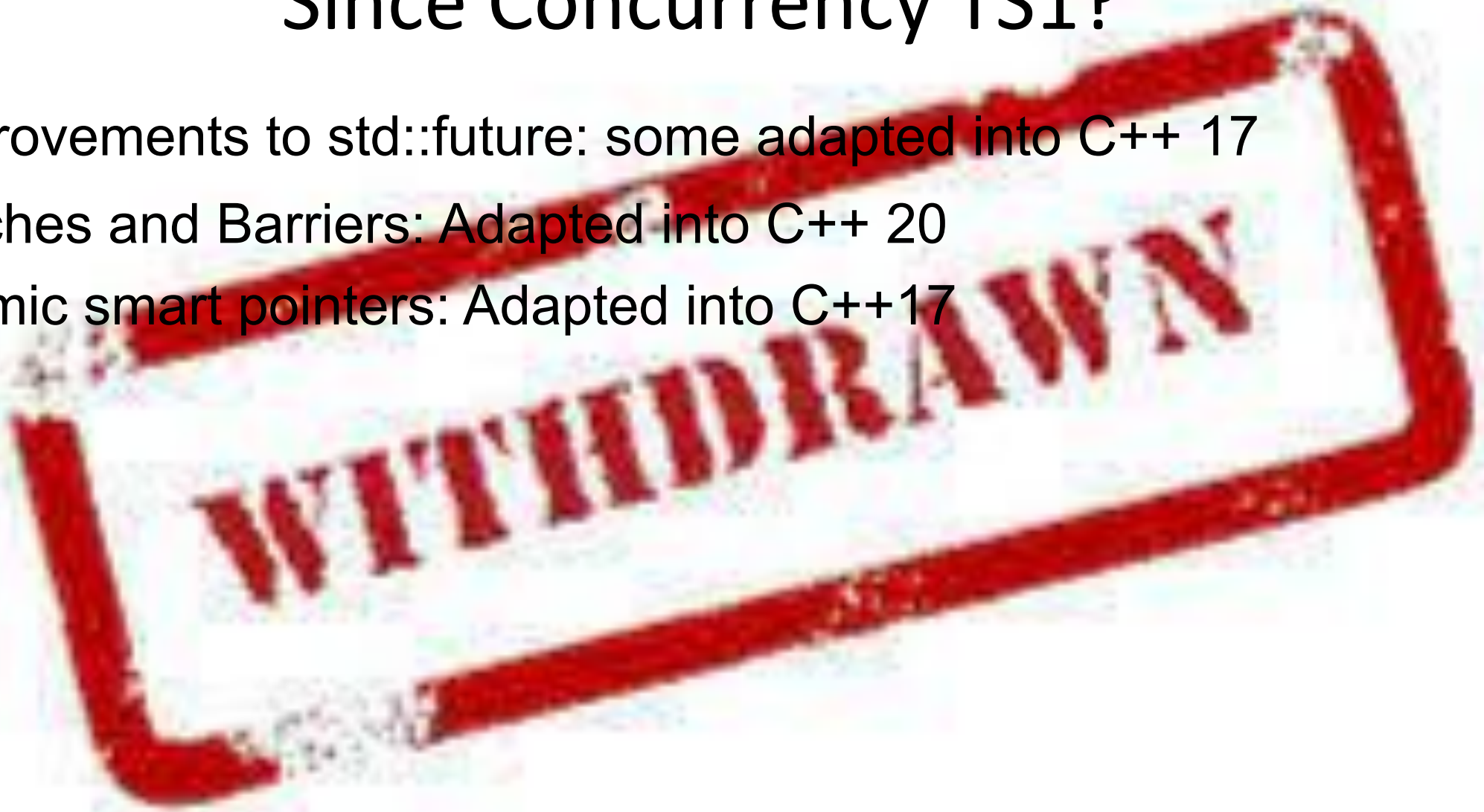
What was in Concurrency TS1?

- Improvements to std::future
- Latches and Barriers
- Atomic smart pointers



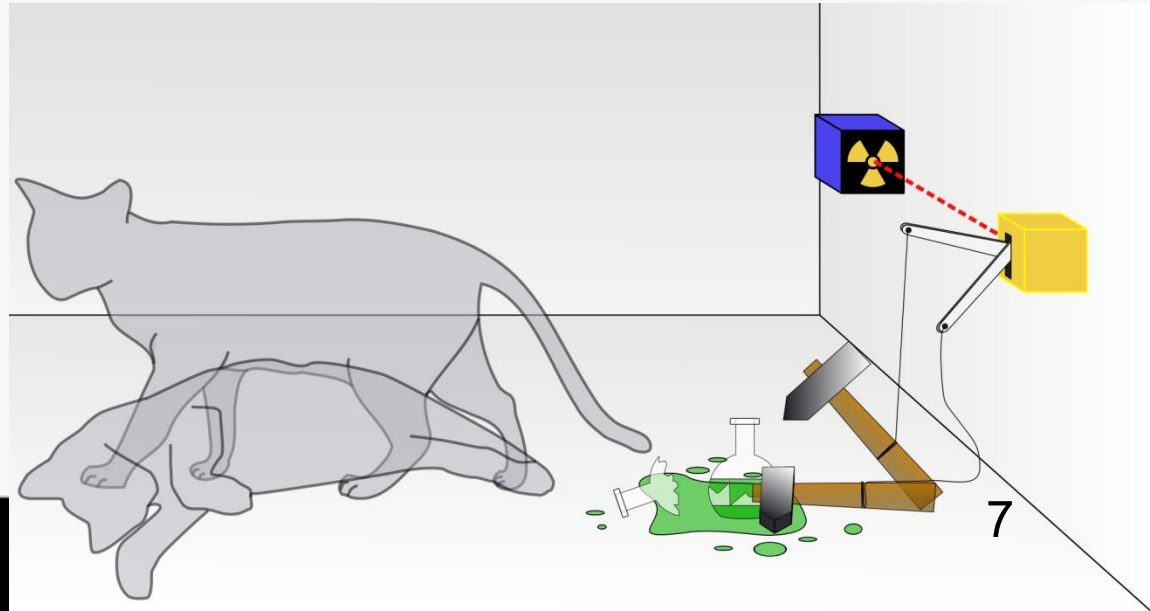
Since Concurrency TS1?

- Improvements to std::future: some adapted into C++ 17
- Latches and Barriers: Adapted into C++ 20
- Atomic smart pointers: Adapted into C++17



Talking about HP and RCU since 2014

1. Erwin Schrödinger's Zoo and Werner Heisenberg's advice
2. Increase uncertainty to get performance and scalability
3. So Procrastinate away! Use Structured Deferral
4. Shared_ptr vs atomic_shared_ptr vs hazard pointers vs Read Copy Update (RCU)
5. Hazard Pointers
6. Read Copy Update
7. A Concurrency Toolkit for C++

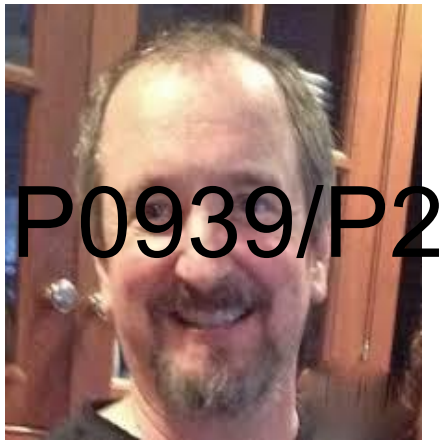


Since 2014, slow as we need to do C++17, 20

- But also we need to learn how to convert from C to C++ interface
- learn new and interesting C++ idioms
- learn new Library conventions
- work with tight schedule
- grow older, kids graduate
- changed jobs, company

To TS or not to TS: that is the question

Whether 'tis nobler in the mind to suffer.
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles.
And by opposing end them.



The role of TSes from P0939/P2000 Directions

We recommend

- Use TSs for library components.
- Don’t use TSs for a language feature unless the feature is a mostly self-contained unit.
- Don’t use a TS simply to delay; it doesn’t simplify later decision making. Have a concrete and articulated criteria for completion.

WG 21 Direction Group

TS vs IS: question TS should answer

- Is there an implementation?
- Is it a Library or Language proposal, or involve both aspects?
- Is the proposal a foundational proposal, meaning many other C++ aspects/proposal depend on it, and/or it depends on many other C++ aspects/proposals?
- Is it independent of aspects of the language.
- Are there competing design proposals?
- Is the proposal complicated or large that you fear there will be error in design decision
- Is it a research idea?
- Is there substantial invention?
- Can it be staged?
- Is there a subpart that deserves to be in IS
- Is the wording complicated or unconventional
- Will the proposal benefit from early integration (can be applied to a WP)
- Will you get feedback/testing only after TS publication or IS publication
- Is there a motivation to slow down a proposal?
- Explicitly state the acceptance criteria for the TS into IS
- Are you juggling a large number of related or dependent proposals (other proposals that depend on this proposal)?
- Are you aiming for user feedback?
- Are you aiming for implementation feedback?
- Is there a scheduling concern to make C++xx for it or its dependents?

Proposal for DG advisory

- WGs SGs decide on TS or IS route and write proposal supporting direction
- The key question:
  - WHAT ARE we hoping to LEARN through a TS must be clearly specified.
  - WHAT ARE the exit criteria of the TS to IS must be clearly specified.
- Other questions should be asked will follow to support your conclusion.
- The previous page are questions the DG may ask. And you should think about.
- We urge SGs to explicitly poll for this and their supporting reasons
- DG will offer non-binding advisory in some cases as
  - whether TS or IS route is preferred, or have you considered an SG
  - In some cases an SG vs TS vs IS continuum needs to be considered
- Please weigh our opinion as part of your decision process
- direction@lists.isocpp.org.

What is in Concurrency TS2?

- Several synchronization primitives for locked-free programming on concurrent data structures. These are cell, hazard ptr and RCU. These extend the existing shared_ptr and the proposed atomic_shared_ptr which all have safe reclamation facilities. As such we also propose moving shared_ptr and atomic<shared<ptr>> to this new location. We suspect this part may be controversial, so would ask for discussion on this topic.
  - P1121R3. Hazard Pointers: Proposed Interface and Wording for Concurrency TS 2.
  - P1122R4 - Proposed Wording for Concurrent Data Structures: Read-Copy-Update (RCU)

Concurrency TS2 in future

Concurrency TS2 is an ongoing WIP but might contain the following which has been making its way through WG21/SG1:

- Data structures such as Concurrent queues, counters,
- Asymmetric fences
- What about executors?

Plan to be in cpluplus github

- https://github.com/cplusplus/concurrency-ts2

Become an IS

- Will it still look like the TS?

Future C++ Std new clause 33

- **33: Concurrency Utilities Library**
  - **33.1 General Concepts**
    - **33.1.1 Thread Support**
    - **33.1.2 Executor Support**

- **33.2 Safe Reclamation**
  - **33.2.1 Hazard Pointers**
  - **33.2.2 RCU**
  - **33.2.3 Latest/Snapshot?**
  - **33.2.4 Asymmetric fences**

To learn or not to learn?

- What did we learn?
- What were the exit criteria?
- What is the exit vehicle?
- Will it still look like the TS in the IS (exit vehicle)?
- What is there still to learn?
- When will we stop learning?
- What is implementation status?
- Did the TS process work for us?

Hazard Pointers in Concurrency TS2, C++26, and beyond

Hazard Pointers in a Nutshell

Used to protect access to objects that may be concurrently removed.

A hazard pointer is a single-writer multi-reader pointer.

If a hazard pointer points to an object
before its removal,
then the object will not be reclaimed
as long as the hazard pointer remains unchanged

<u>Protect object A</u>
Set a hazard pointer to point to A
if A is not removed
then it is safe to use A

<u>Remove and reclaim object A</u>
Remove A
if no hazard pointers point to A
then it is safe to reclaim A

Features:

- Fast and scalable protection
- Supports arbitrarily long protection

18

Hazard Pointers TS2 Interface

Components:

- **Hazard pointers**
- **Objects protectable by hazard pointers**
- **Domain(s) to manage hazard pointers and retired objects**

Hazard Pointers TS2 Interface

```
class hazard_pointer_domain {
public:
  hazard_pointer_domain() noexcept;
  explicit hazard_pointer_domain(
      pmr::polymorphic_allocator<byte> poly_alloc) noexcept;
  hazard_pointer_domain(const hazard_pointer_domain&) = delete;
  hazard_pointer_domain& operator=(const hazard_pointer_domain&) = delete;
  ~hazard_pointer_domain();
};

hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

// For synchronous reclamation
void hazard_pointer_clean_up(
    hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
```

Hazard Pointers TS2 Interface

```
template <typename T, typename D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(
    D d = D(),
    hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
  void retire(hazard_pointer_domain& domain) noexcept;
};
```

Hazard Pointers TS2 Interface

```
class hazard_pointer {
public:
  hazard_pointer() noexcept; // Empty
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] bool empty() const noexcept;
  template <typename T> T* protect(const atomic<T*>& src) noexcept;
  template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
  template <typename T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};

hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

Usage Example

```
class Foo : public hazard_pointer_obj_base<Foo> { /* Foo members */ };

void read_and_use(const std::atomic<Foo*>& src, Func fn) { // Called frequently
  hazard_pointer h = make_hazard_pointer();
  Foo* ptr = h.protect(src);
  fn(ptr); // ptr is protected
}

void update(std::atomic<Foo*>& src, Foo* newptr) { // Called infrequently
  Foo* oldptr = src.exchange(newptr);
  oldptr->retire();
}
```

What Did We Learn in 4 Years?

- Open source: `github.com/facebook/folly` under `synchronization/Hazptr.h`
- Synchronous reclamation:
  - TS2 global cleanup is a powerful but blunt tool.
  - Folly (fast and scalable) cohort synchronous reclamation.
  - CPPCON 2021: Hazard pointer synchronous reclamation beyond Concurrency TS2
- Integrated link counting:
  - Not in TS2. Folly support for linked structures with immutable links (e.g., queues).
    Can reclaim nodes of arbitrary depth in one check of hazard pointers.
- Hazard pointers arrays optimizations
  - Not in TS2. Folly make_hazard_pointer_arrray<M>(), e.g., 4, 5, 6 ns vs 4, 8, 12 ns
- Optional dedicated thread pool for asynchronous reclamation:
  - Robustness against latency spikes and deadlock.
- Domains:
  - Robust default domain with expanded capabilities (cohorts, link counting, array optimization).
  - No customization needed in Folly so far.

Hazard Pointers Proposal for C++26

Minimalist useful subset of TS2:

- Supports asynchronous reclamation
- Compatible with external link counting and automatic retirement
- Strict subset of TS2 API and wording
- No custom domains (for now)
- No synchronous reclamation (for now)
- Can be extended

Hazard Pointers Proposal for C++26

```
class hazard_pointer_domain {
public:
  hazard_pointer_domain() noexcept;
  explicit hazard_pointer_domain(
      pmr::polymorphic_allocator<byte> poly_alloc) noexcept;
  hazard_pointer_domain(const hazard_pointer_domain&) = delete;
  hazard_pointer_domain& operator=(const hazard_pointer_domain&) = delete;
  ~hazard_pointer_domain();
};

hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

// For synchronous reclamation
void hazard_pointer_clean_up(
    hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
```

Hazard Pointers Proposal for C++26

```
template <typename T, typename D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(
    D d = D(),
    hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
  void retire(hazard_pointer_domain& domain) noexcept;
};
```

Hazard Pointers Proposal for C++26

```
class hazard_pointer {
public:
  hazard_pointer() noexcept; // Empty
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] bool empty() const noexcept;
  template <typename T> T* protect(const atomic<T*>& src) noexcept;
  template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
  template <typename T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};

hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

Hazard Pointers Proposal for C++26

```
template <typename T, typename D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(D d = D()) noexcept;
};

class hazard_pointer {
public:
  hazard_pointer() noexcept; // Empty
  hazard_pointer(hazard_pointer&&) noexcept;
  hazard_pointer& operator=(hazard_pointer&&) noexcept;
  ~hazard_pointer();
  [[nodiscard]] bool empty() const noexcept;
  template <typename T> T* protect(const atomic<T*>& src) noexcept;
  template <typename T> bool try_protect(T*& ptr, const atomic<T*>& src)
noexcept;
  template <typename T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};

hazard_pointer make_hazard_pointer();

void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

Hazard Pointers Beyond C++26

- Hazard pointer array optimization
  - In heavy use in Folly for ~4 years. Simple.
- Synchronous reclamation
  - Folly cohort synchronous reclamation: In heavy use in Folly for 3+ years.
  - Global cleanup as in TS2?
  - Other variations?
  - CPPCON 2021: Hazard pointer synchronous reclamation beyond Concurrency TS2
- Integrated link counting
  - In heavy use in Folly for ~4 years. Formal wording may not be simple.
- Domains:
  - Custom domain allocators as in TS2?
  - WiredTiger Feedback: Separate checking protection from reclamation.
  - Folly experience: Robust default domain. No custom domains needed so far.

RCU in Concurrency TS 2

C++ RCU: A Learning Experience

My previous C++ project had been in 1990

My initial attempt at RCU bindings in C++ thus used “`virtual`”

This resulted in some pointed feedback

Again With Curiously Recurring Template Pattern

Diagnostic-driven development leads to this dubious code:

```
struct foo: std::rcu_obj_base<foo> {
    int a;
};
```

Actually, RCU will be in an experimental namespace rather than `std::`, but I am being optimistic!

Again With Curiously Recurring Template Pattern

Diagnostic-driven development leads to this dubious code:

```
struct foo: std::rcu_obj_base<foo> {
    int a;
};
```

But it compiles?

Again With Curiously Recurring Template Pattern

Diagnostic-driven development leads to this dubious code:

```
struct foo: std::rcu_obj_base<foo> {
    int a;
};
```

But it compiles? And it works???

Again With Curiously Recurring Template Pattern

Diagnostic-driven development leads to this dubious code:

```
struct foo: std::rcu_obj_base<foo> {
    int a;
};
```

But it compiles? And it works???

The magic of CRTP!!!

Mutually Assured Education

- My knowledge of C++ was and is limited
- Others' knowledge of RCU was and is limited
- Therefore, lots of discussion and code samples
  - https://github.com/paulmckrcu/RCUCPPbindings Test/paulmck
  - Many thanks to my many teachers, especially those who taught in code:
    - Arthur J. O'Dwyer, Daisy Hollman, and Izzy Muerte
- And lots of discussions afterwards
  - Too many to fit on a slide, but see authors and contributors to many papers

A Little Bike-Shedding Along the Way

Wikimedia Commons User SeppVei

A Little Bike-Shedding Along the Way

- `template<T>` replaced museum-piece abstract classes ;-)
- `synchronize_rcu()` to `rcu_synchronize()` for consistency
- RAII: `rcu_reader` to a *Cpp17BasicLockable* `rcu_domain`
- Deleters may be invoked directly from a `retire` call
  - Late-breaking news: May need to inform users of this (more on this later)
- Non-intrusive `rcu_retire()` (now in Linux kernel...)

RCU RAII Readers

- As C++ developers might expect:

```
void an_rcu_reader()
{
        do_something_before_reader();
        std::unique_lock<std::rcu_domain> rdru(std::rcu_default_domain());
        do_something_within_reader();;
}

void wait_for_rcu_readers()
{
        rcu_synchronize();
}
```

As RCU users might expect:

Author: ADA&Neagoe This file is licensed under the Creative Commons Attribution-ShareAlike license versions 3.0, 2.5, 2.0, and 1.0.

RCU RAII Readers

- As C++ developers might expect, but more succinctly:

```
void an_rcu_reader()
{
        do_something_before_reader();
        std::unique_lock<std::rcu_domain> rdru();
        do_something_within_reader();;
}
```

- Except that not all the world can live within the confines of an RCU RAII reader...

RCU Non-RAII Readers

- And another fine example of diagnostic-driven development!
- Function to start an RCU reader:

```
std::unique_lock<std::rcu_domain> start_deferred_reader()
{
        std::unique_lock<std::rcu_domain> new_rdr(std::rcu_default_domain());
        return std::move(new_rdr);
}
```

- Function to end an RCU reader:

```
void end_deferred_reader(std::unique_lock<std::rcu_domain> old_rdr)
{
}
```

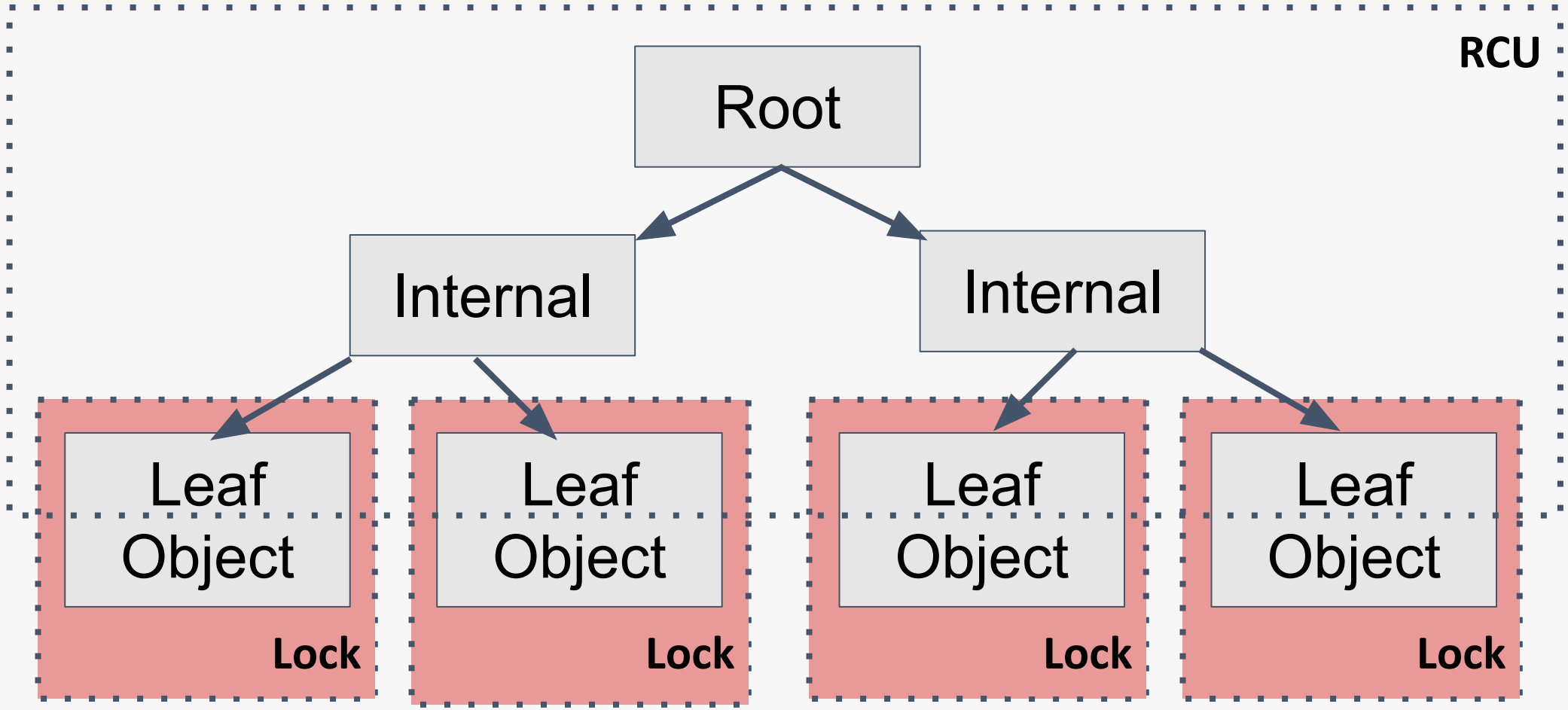
Invoking RCU Non-RAII Readers

- Whenever the spirit `std::move()`s you:

```
void an_rcu_reader()
{
        do_something_before_reader();
        auto rdr = std::move(start_deferred_reader()); // rcu_read_lock();
        do_something_within_reader();
        end_deferred_reader(std::move(rdr));           // rcu_read_unlock();
        do_something_after_reader();
}
```

- But why not just add a pair of curly braces???

Why RCU Non-RAII Readers?



Why RCU Non-RAII Readers?

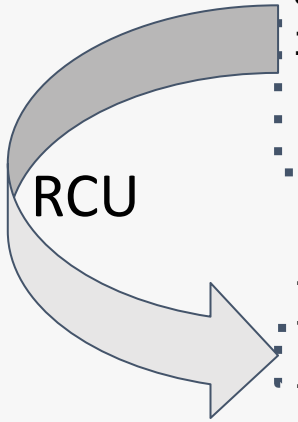
- Use RCU to protect a search structure, and locking on objects

```
void update_object(int key)
{
        auto rdr = std::move(start_deferred_reader()); // rcu_read_lock();
        auto p& = find_object(key);
        if (needs_update(p)) {
                std::lock_guard<std::mutex> guard(p.objmutex);
                end_deferred_reader(std::move(rdr));   // rcu_read_unlock();
                if (needs_update(p))
                        do_rcu_unsafe_locked_update(p);
        } else {
                end_deferred_reader(std::move(rdr));   // rcu_read_unlock();
        }
}
```

Why RCU Non-RAII Readers?

- Use RCU to protect a search structure, and locking on objects

```
void update_object(int key)
{
        auto rdr = std::move(start_deferred_reader()); // rcu_read_lock();
        auto p& = find_object(key);
        if (needs_update(p)) {
                std::lock_guard<std::mutex> guard(p.objmutex);
                end_deferred_reader(std::move(rdr));   // rcu_read_unlock();
                if (needs_update(p))
                        do_rcu_unsafe_locked_update(p);
        } else {
                end_deferred_reader(std::move(rdr));   // rcu_read_unlock();
        }
}
```



Why RCU Non-RAII Readers?

- Use RCU to protect a search structure, and locking on objects

```
void update_object(int key)
{
        auto rdr = std::move(start_deferred_reader()); // rcu_read_lock();
        auto p& = find_object(key);
        if (needs_update(p)) {
                std::lock_guard<std::mutex> guard(p.objmutex);
                end_deferred_reader(std::move(rdr));   // rcu_read_unlock();
                if (needs_update(p))
                        do_rcu_unsafe_locked_update(p);
        } else {
                end_deferred_reader(std::move(rdr));   // rcu_read_unlock();
        }
}
```

RCU

Locking

What Future Learnings Might There Be?

- QEMU developers' on deleters being invoked from rcu_retire():
  - Don't do that!!! We hate the resulting deadlocks!!!

What `rcu_retire()` deadlocks???

- If any lock is acquired by any deleter, that lock cannot be held across any call to `.retire()` or `rcu_retire()`!

```
void hapless_retire_invoker(Foo *p)
{
        std::lock_guard<std::mutex> guard(mymutex);
        rcu_retire(p);
        // Which might invoke deleters.
        // And if any of those deleters acquire mymutex, game over!!!
}
```

What Future Learnings Might There Be?

- QEMU developers' on deleters being invoked from rcu_retire():
  - Don't do that!!! We hate the resulting deadlocks!!!
  - But some environments don't have much choice
  - Perhaps a static function? If it returns `false`, no such deadlocks!

```
bool rcu_deleters_from_retire(rcu_domain& dom = rcu_default_domain()) noexcept;
```

What Future Learnings Might There Be?

- QEMU developers' on deleters being invoked from rcu_retire():
  - Don't do that!!! We hate the resulting deadlocks!!!
  - But some environments don't have much choice
  - Perhaps a static function? If it returns `false`, no such deadlocks!
    ```
    bool rcu_deleters_from_retire(rcu_domain& dom = rcu_default_domain()) noexcept;
    ```
  - Maybe `rcu_retire()`? Type trait saying beg/borrow/steal thread? ...

What Future Learnings Might There Be?

- QEMU developers' on deleters being invoked from rcu_retire():
  - Don't do that!!! We hate the resulting deadlocks!!!
  - But some environments don't have much choice
  - Perhaps a static function? If it returns `false`, no such deadlocks!
    ```
    bool rcu_deleters_from_retire(rcu_domain& dom = rcu_default_domain()) noexcept;
    ```
  - Maybe `rcu_retire()`? Type trait saying beg/borrow/steal thread? ...
- Additional unique_lock/lock_guard constructors for RCU?
- Some users might want a rough count of outstanding deleters
- Multiple instances of `rcu_domain`? Later...
- And there is still `memory_order_consume`...

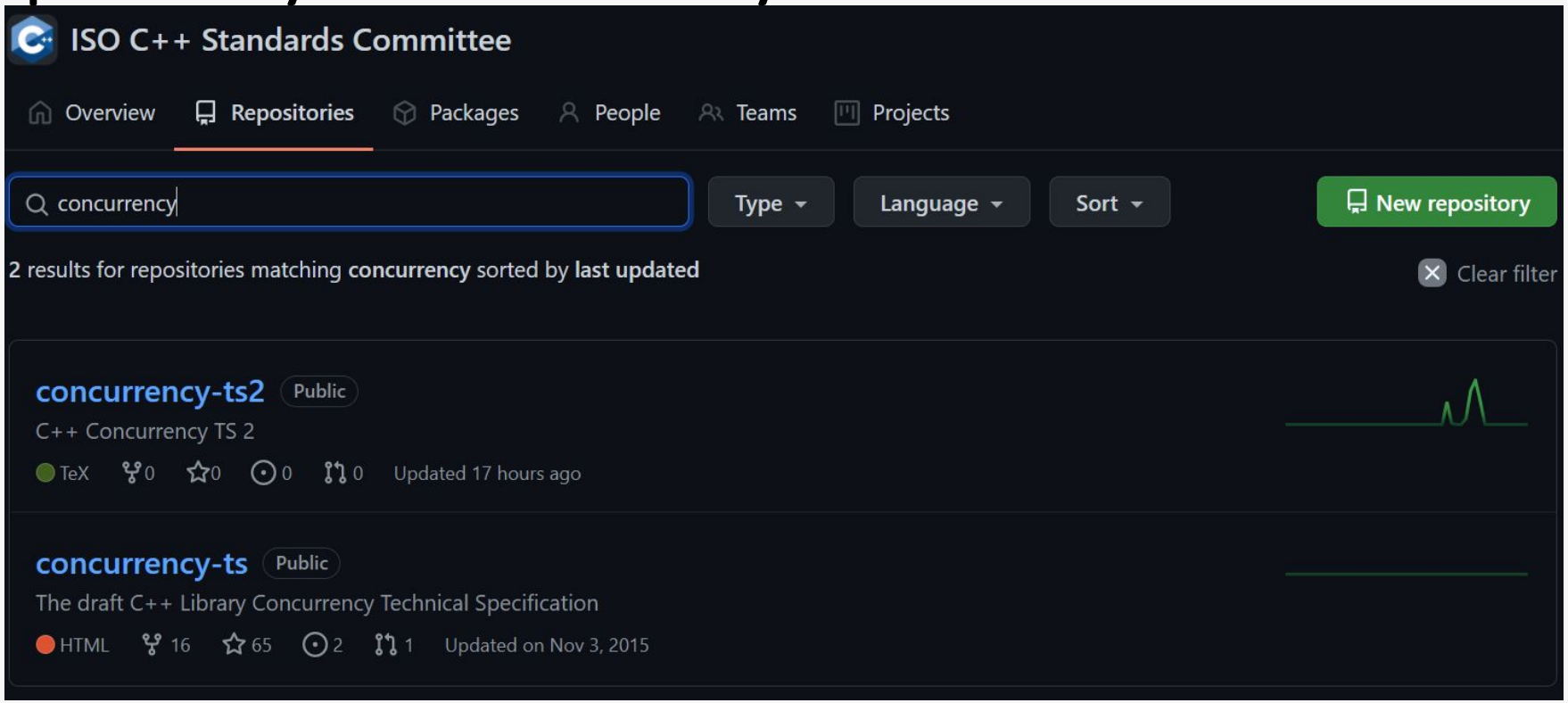
What Future Learnings Might There Be?

- QEMU developers' on deleters being invoked from rcu_retire():
  - Don't do that!!! We hate the resulting deadlocks!!!
  - But some environments don't have much choice
  - Perhaps a static function? If it returns `false`, no such deadlocks!
    ```
    bool rcu_deleters_from_retire(rcu_domain& dom = rcu_default_domain()) noexcept;
    ```
  - Maybe `rcu_retire()`? Type trait saying beg/borrow/steal thread? ...
- Additional unique_lock/lock_guard constructors for RCU?
- Some users might want a rough count of outstanding deleters
- Multiple instances of `rcu_domain`? Later...
- And there is still `memory_order_consume`...
- None of which are on critical path to IS

Final Words

The IRONY: it is not lost on us

SG1 Concurrency SG will have 2 concurrency TSes in the github repository concurrently



What is in Concurrency TS2?

- Several synchronization primitives for locked-free programming on concurrent data structures. These are cell, hazard ptr and RCU. These extend the existing shared_ptr and the proposed atomic_shared_ptr which all have safe reclamation facilities. As such we also propose moving shared_ptr and atomic<shared<ptr>> to this new location. We suspect this part may be controversial, so would ask for discussion on this topic.
  - P1121R3. Hazard Pointers: Proposed Interface and Wording for Concurrency TS 2.
  - P1122R4 - Proposed Wording for Concurrent Data Structures: Read-Copy-Update (RCU)

BACKUP