Document number: P1122R4
Date: 20210514
Project: Programming Language C++, WG21, LEWG & LWG
Authors: Paul McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński,, Jens Maurer
Email: paulmckrcu@fb.com, michael@codeplay.com, maged.michael@acm.org, gromer@google.com, andrewhhunter@gmail.com, arthur.j.odwyer@gmail.com, dshollm@sandia.gov, jfbastien@apple.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com, erik@rigtorp.se, tomaszkam@gmail.com, jens.maurer@gmx.net
Reply to: paulmckrcu@fb.com

Proposed Wording for Concurrent Data Structures: Read-Copy-Update (RCU)

1. Introduction

This paper is a successor to the RCU portion of P0566R5, in response to LEWG's Rapperswil 2018 request that the two techniques be split into separate papers.

This is proposed wording for Read-Copy-Update [P0461], which is a technique for safe deferred resource reclamation for optimistic concurrency, useful for lock-free data structures. Both RCU and hazard pointers have been progressing steadily through SG1 based on years of implementation by the authors, and are in wide use in MongoDB (for Hazard Pointers), Facebook, and Linux OS (RCU).

We originally decided to do both papers' wording together to illustrate their close relationship, and similar design structure, while hopefully making it easier for the reader to review together for this first presentation. As noted above, they have been split on request.

This wording is based P0566r5, which in turn was based on that of on n4618 draft [N4618]

2. History/Changes from Previous Release

2021-05-14 [P1122R4] after virtual LWG meeting

- Rework wording defining nesting of `lock` and `unlock`.
- Clarify consequences of exceptions.
- Explicitly allow non-deleter expressions for `retire` and `rcu_retire`.
- Clarify effects of constructing/initializing/assigning expressions passed to `retire` and `rcu_retire`.
- Clarify what can and cannot cause a data race.
- Clarify `rcu_synchronize` synchronization.
- Added TS-experience question asking whether a given object may be passed to multiple invocations of `rcu_retire`.

2021-04-30 [D1122R4] after virtual LWG meeting

- Change `struct` to `class` as a consequence of adding `private` and `protected` members.
- Align with hazard-pointer wording in P1121R3.
- Return the single global `rcu_domain` from an `rcu_default_domain` function call instead of a global object in order to avoid constructor-ordering problems.

2020-08-28 [D1122R3] after virtual LWG meeting

- Apply initial feedback from the meeting.
- Awaiting LEWG decision on reuse of lock_guard and unique_lock:
    - Just add `rcu_reader` to the list in 32.5.3.2 [[thread.mutex.requirements.mutex]? This would require adding an always-successful "try" variant of the counterpart to the Linux kernel's `rcu_read_lock()`, but that is straightforward.
    - Or is there a class similar to mutex that implements `Cpp17BasicLockable`? A quick search through the current working draft fails to find one. The downside of introducing (say) a basic_mutex is that it seems to require replicating 32.5.3.2 [thread.mutex.requirements.mutex], which seems a bit heavy weight.
    - Whether or not deleters may be invoked from `rcu_retire()`, `retire()`, or `rcu_domain::unlock()` is implementation defined. Allowing such invocation allows implementations to create fewer threads, but also means that a deleter cannot acquire any resource held across any invocation of `rcu_retire()`, `retire()`, or `rcu_domain::unlock()` destructor without risking deadlock. Some implementations (for example, the Linux kernel) may therefore choose to guarantee that deleters will be invoked from a separate thread in order to provide users more flexibility in resource acquisition while still avoiding deadlock.
    - Switch from rcu_reader to rcu_domain, which implements `Cpp17BasicLockable`. **LEWG might wish to review this change**.
    - Fine-tuning of the various clauses.
- Significant changes against P1122R2 are noted below:
    - Change `class` to `struct`.
    - Provide a global domain without the ability for users to create domains.
    - Namespace `std::experimental::inline concurrency_v2.`
    - Use `Cpp17BasicLockable` to provide RAII RCU readers.
    - Follow `unique_ptr` wording for deleters.

2020-08-25 [D1122R3] preparation for virtual LWG meeting

- Remove "RAII" from wording because it is not defined in the standard.
- Change bogus "jstring" to "string"

2018-11-06 [P1122R2] post-San Diego meeting

- Adjusted sections to new naming schema introduced at the Rapperswil Meeting 2018 for C++20. (Requires->(Mandates (compile-time), Expects(contract)), Remarks->Constraints, Postconditions->Ensures as documented in the [structure.specifications] section of N4762).
- Improve wording (courtesy of Geoff Romer).
- Markup reflects changes since P1122R0.

2018-07-06 [D1122R1] pre-San Diego meeting

- Added list of open issues to be addressed by TS implementation experience.

2018-06-07 [P1122R0] post-Rapperswil meeting

- Extracted the RCU portion of P0566R5 per request by LEWG.
- Add the `explicit` keyword to the `defer_lock_t` constructor of `rcu_reader` per request by LEWG.
- Add the destructor description per request by LEWG.
- Andrej Krzemienski of LEWG: Have `reset` to pre-destruct an `rcu_reader`? Defer to TS experience because we don't know this use case will actually appear.
- Anthony Williams via email: Allow the deleter to be specified at construction time in addition to at retire time. Defer to TS experience.
- Remove swap declaration, thus relying on default definition per request by LEWG.
- Moved Preamble to D0940R2.
- Archival version: https://docs.google.com/document/d/1wls6q2mE60I5uZFug5U5i21N75j--s0wFOJA3e7MuKY/edit

2018-03-12 [P0566R5] pre-Rapperswil meeting

- Updated RCU ordering guarantees for readers and deleters.
- Remove `noexcept` from `rcu_reader` destructor.
- Drop the detailed description of the `rcu_reader` destructor.
- Word the `retire` member function based on the `rcu_retire` free function.
- Word the `synchronize_rcu` free function based on `rcu_retire`.
- Rename `synchronize_rcu` to `rcu_synchronize`, as requested by LEWG in JAX.
- Confirmed that `rcu_synchronize` has SC fence semantics, and added a section to the RCU litmus-tests paper (P0868R1)
- Added feature test macros `__cpp_hazard_pointers` and `__cpp_read_copy_update`.
- Added wording constraining deleters.
- Hazard pointer changes:
    - Changed the introduction and the wording for hazptr_cleanup(), hazptr_obj_base retire(), hazptr_holder try_protect() to consider the lifetime of each hazard pointer as a series of epochs to facilitate specifying memory ordering (based on JAX evening session).
    - Changed the name of hazptr_holder get_protected() to protect(), as instructed by LEWG.
    - Changed the default constructor for hazptr_holder to return an empty hazptr_holder, as instructed by LEWG.

    - Removed the hazptr_holder member function make_empty(), by implication of changing the default constructor.
    - Added the free function make_hazptr(), which constructs a non-empty hazptr_holder, to replace the functionality of the hazptr_holder constructor.
    - Changed the name of hazptr_holder reset() to reset_protected() for clarity.

2017-11-08 [P0566R4] pre-JAX meeting

- Full RCU wording review was done at this meeting. A repeat HP wording done at this meeting for any small design deltas, although HP was approved to move to LEWG in Toronto
- Three related bugzillas tracking this:
    - 382 C++ Concurre parallel@lists.isocpp.org CONF --- Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU) Tue 23:01
    - 291 C++ Library fraggamuffin@gmail.com SG_R --- Hazard Pointers 2017-07-06
    - 376 C++ Concurre maged.michael@acm.org SG_R --- Hazard Pointers Mon 22:58
- Rewrote the RCU preamble to give a better introduction to RCU's concepts and use cases, including adding example code.
- Updated the ordering guarantees to be more like the C++ memory model and less like the Linux kernel memory model. (There is still some refining that needs to be done, and this is waiting on an RCU litmus-test paper by Paul E. McKenney, now available as P0868R1.)
- Removed the `lock` and `unlock` member functions from the `rcu_reader` class. These member functions are not needed because `rcu_reader` directly provides the needed RAII functionality.
- Numerous additional wording changes were made, none of which represent a change to the design, implementation, or API.
- Added some authors.
- Hazard pointer wording changes:
    - Added `hazptr_cleanup()` free function, a stronger replacement for `hazptr_barrier()`. There was no consensus in Albuquerque on the requirements for a such a function. The decision on whether to provide one and its semantics was left to the authors.
    - Significant rewrite of the wording for `hazptr_obj_base::retire()` to address the issues with memory ordering raised in Toronto.
    - Rewrite of the wording for `hazptr_holder::try_protect()` for clarity.
    - Other minor editorial changes and corrections.

2017-10-15 [P0566R3] pre-ABQ meeting

- Changed the syntax for the polymorphic allocator passed to the constructor of `hazptr_domain`. The constructor is no longer constexpr.
- Added the free function `hazptr_barrier()` that guarantees the completion of reclamation of all objects retired to a domain.
- Changed the syntax of constructing empty `hazptr_holder`-s.
- Changed the syntax of the `hazptr_holder` member function that indicated whether a `hazptr_holder` is empty or not.
- Added a note that an empty `hazptr_holder` is different from a `hazptr_holder` that owns a hazard pointer with null value.
- Added a note to clarify that it acceptable for `hazptr_holder` `try_protect` to return true when its first argument has null value.
- Update RCU presentation to reduce member-function repetition.
- Fix RCU s/Void/void/ typo
- Remove RCU’s std::nullptr_t in favor of the new-age std::defer_lock_t.
- Remove RCU’s barrier() member function in favor of free function based on BSI comment

2017-07-30 [P0566R2] post-Toronto meeting

- Allow hazptr_holder to be empty. Add a move constructor, empty constructor, move assignment operator, and a bool operator to check for empty state.
- A call by an empty hazptr_holder to any of the following is undefined behavior: reset(), try_protect() and get_protected().
- Destruction of an hazptr_holder object may be invoked by a thread other than the one that constructed it.
- Add overload of `hazptr_obj_base` `retire()`.

2017-06-18 [P0566R1] pre-Toronto meeting

- Addressed comments from Kona meeting
- Removed Clause numbering 31 to leave it to the committee to decide where to inject this wording
- Renamed `hazptr_owner` `hazptr_holder`.
- Combined `hazptr_holder` member functions `set()` and `clear()` into `reset()`.
- Replaced the member function template parameter `A` for `hazptr_holder` `try_protect()` and `get_protected` with `atomic<T*>`.
- Moved the template parameter `T` from the class `hazptr_holder` to its member functions `try_protect()`, `get_protected()`, and `reset()`.
- Added a non-template overload of `hazptr_holder::reset()` with an optional `nullptr_t` parameter.

- Removed the template parameter `T` from the free function `swap()`, as `hazptr_holder` is no longer a template.
- Almost complete rewrite of the hazard pointer wording.

----

3. Guidance to Editor

RCU is a proposed addition to the C++ standard library, for the concurrency TS. It has been approved for addition through multiple SG1/SG14 sessions.

As RCU (and Hazard Pointers) are related to a concurrent shared pointer, we are looking at building a new clause 33 for Concurrency Utilities Library through P0940 [P0940].In P0940, we plan to introduce subclause on Safe Reclamation which will support RCU, Hazard Pointer, as well as a other similar features.

We will not make any assumption for now as to the placement of this wording and leave it to SG1/LEWG/LWG to decide and have used ? as a Clause placeholder. We believe that Hazard Pointers and RCU should appear in the same section.

4. Proposed wording

?.1 Read-Copy Update (RCU) [rcu]

RCU is a synchronization mechanism that can be used for linked data structures that are frequently read, but seldom updated. RCU does not provide mutual exclusion, but instead allows the user to schedule specified actions such as deletion at some later time.

A class type `T` is *rcu-protectable* if it has exactly one public base class of type `rcu_obj_base<T,D>` for some `D` and no base classes of type `rcu_obj_base<X,Y>` for any other combination `X`, `Y`. An object is rcu-protectable if it is of rcu-protectable type.

An invocation of `unlock` `U` on an `rcu_domain` `dom` *corresponds* to an invocation of `lock` `L` on `dom` if `L` is sequenced before `U` and either

- no other invocation of `lock` on `dom` is sequenced after `L` and before `U` or
- every invocation of unlock `U’` on `dom` such that `L` is sequenced before `U’` and `U’` is sequenced before `U` corresponds to an invocation of lock `L’` on `dom` such that `L` is sequenced before `L’` and `L’` is sequenced before `U’`.

[Note: This pairs nested locks and unlocks on a given domain in each thread.]

A *region of RCU protection* on a domain `dom` starts with a `lock` `L` on `dom` and ends with its corresponding `unlock` `U`.

Given a region of RCU protection `R` on a domain `dom` and given an evaluation `E` that scheduled another evaluation `F` in `dom`, if `E` does not strongly happen before the start of `R`, the end of `R` strongly happens before evaluating `F`.

The evaluation of a scheduled evaluation is potentially concurrent with any other such evaluation. Each scheduled evaluation is evaluated at most once.

Header <rcu> synopsis [rcu.syn]

```
namespace std::experimental::inline concurrency_v2 {

  // ?.2.1, class template rcu_obj_base
  template<class T, class D = default_delete<T>>
    class rcu_obj_base;

  // ?.2.2, class rcu_domain
  class rcu_domain;

  // ?.2.3 rcu_default_domain
  rcu_domain& rcu_default_domain() noexcept;

  // ?.2.4 rcu_synchronize
  void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;

  // ?.2.5 rcu_barrier
  void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;

  // ?.2.6 rcu_retire
  template<class T, class D = default_delete<T>>
    void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
}
```

?.2.1, class template `rcu_obj_base` [rcu.base]

Objects of type `T` to be protected by RCU inherit from a specialization `rcu_obj_base<T,D>`.

```
template<class T, class D = default_delete<T>>
class rcu_obj_base {
public:
  // ?.2.1.1, rcu.base.retire
  void retire(D d = D(), rcu_domain& dom = rcu_default_domain()) noexcept;
protected:
  rcu_obj_base() = default;
private:
  D deleter;  // exposition only
};
```

1. A client-supplied template argument `D` shall be a function object type ([function.object]) for which, given a value d of type `D` and a value `ptr` of type `T*`, the expression `d(ptr)` is valid and has the effect of disposing of the pointer as appropriate for that deleter.
2. The behavior of a program that adds specializations for `rcu_obj_base` is undefined.
3. D shall meet the requirements for Cpp17DefaultConstructible and Cpp17MoveAssignable.
4. `T` may be an incomplete type.
5. If `D` is trivially copyable, all specializations of `rcu_obj_base<T,D>` are trivially copyable.

```
void retire(D d = D(), rcu_domain& dom = rcu_default_domain()) noexcept;
```

1. Mandates: `T` is an rcu-protectable type.
2. Preconditions: `*this` is a base class subobject of an object `x` of type `T`. The member function `rcu_obj_base<T,D>::retire` was not invoked on `x` before. The assignment to

deleter does not throw an exception. The expression deleter(addressof(x)) has well-defined behavior and does not throw an exception.

3. Effects: Evaluates deleter = std::move(d) and schedules the evaluation of the expression deleter(addressof(x)) in the domain dom.
4. Remarks: It is implementation-defined whether or not scheduled evaluations in dom can be invoked by the retire function. [Note: If such evaluations acquire resources held across any invocation of retire on dom, deadlock can occur. -- end note.]

?.2.2, class rcu_domain [rcu.domain]

This class meets the requirements of Cpp17BasicLockable [thread.req.lockable.basic] and provides regions of RCU protection, for example, as follows:
std::scoped_lock<rcu_domain> rlock(rcu_default_domain());

```
// ?.2.2, class rcu_domain
class rcu_domain {
public:
  // ?.2.2.1, rcu_domain: RCU domain, or instance
  rcu_domain(const rcu_domain&) = delete;
  rcu_domain& operator=(const rcu_domain&) = delete;

  void lock() noexcept;
  void unlock() noexcept;
};
```

The functions lock and unlock establish (possibly nested) regions of RCU protection.

?.2.2.1, lock [rcu.domain.lock]

```
void lock() noexcept;
```

1. Effects: Opens a region of RCU protection.
2. Remarks: Calls to the function lock do not introduce a data race (16.4.6.10) involving *this.

?.2.2.2, unlock [rcu.domain.unlock]

```
void unlock() noexcept;
```

1. Preconditions: A call to the function lock that opened an unclosed region of RCU protection is sequenced before the call to unlock.
2. Effects: Closes the unclosed region of RCU protection that was most recently opened.

3. Remarks: It is implementation-defined whether or not scheduled evaluations in `*this` can be invoked by the `unlock` function. [Note: If such evaluations acquire resources held across any invocation of `unlock` on `*this`, deadlock can occur.] Calls to the function `unlock` do not introduce a data race (16.4.6.10) involving `*this`. [Note: Evaluation of scheduled evaluations can still cause a data race. -- end note.]

?.2.3, rcu_default_domain [rcu.default.domain]

```
  rcu_domain& rcu_default_domain() noexcept;
```

1. Returns: A reference to the default object of type `rcu_domain`. A reference to the same object is returned every time this function is called.

?.2.4 rcu_synchronize [rcu.synchronize]

```
   void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;
```

1. Effects: If the call to `rcu_synchronize` does not strongly happen before the `lock` opening an RCU protection region `R` on `dom`, blocks until the `unlock` closing `R` happens.
2. Synchronization: The `unlock` closing `R` strongly happens before the return from `rcu_synchronize`.

?.2.5, rcu_barrier [rcu.barrier]

```
   void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;
```

1. Effects: May evaluate any scheduled evaluations in `dom`. For any evaluation that happens before the call to `rcu_barrier` and that schedules an evaluation `E` in `dom`, blocks until `E` has been evaluated.
2. Synchronization: The evaluation of any such `E` strongly happens before the return from `rcu_barrier`.

?.2.6, template rcu_retire [rcu.retire]

```
template<class T, class D = default_delete<T>>
void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
```

1. Mandates: is`_move_constructible_v<D>` is true.
2. Preconditions: `D` meets the Cpp17MoveConstructible and Cpp17Destructible requirements. The expression `d1(p)`, where `d1` is defined below, is well-formed and its evaluation does not exit via an exception.
3. Effects: May allocate memory. It is unspecified whether the memory allocation is performed by invoking `operator new`. Initializes an object `d1` of type `D` from

`std::move(d)`. Schedules the evaluation of `d1(p)` in the domain `dom`. [Note: If `rcu_retire` exits via an exception, no evaluation is scheduled. -- end note]
4. Throws: Any exception that would be caught by a handler of type `bad_alloc`. Any exception thrown by the initialization of `d1`.
5. Remarks: It is implementation-defined whether or not scheduled evaluations in `dom` can be invoked by the `rcu_retire` function. [Note: If such evaluations acquire resources held across any invocation of `rcu_retire` on `dom`, deadlock can occur.]

5. Issues Requiring TS Implementation Experience

We expect that implementation experience will help shed light on the following open issues:

1. Should the standard provide RCU domains, such that readers in one domain have no effect on updaters in any other domain? Although these have proven useful in other environments (e.g., Linux-kernel SRCU), they are rarely used and it has usually taken some years in any given environment for the need for them to arise. In addition, RCU can take advantage of extremely effective batching optimizations that are partially or even wholly defeated by excessive use of domains. These considerations have resulted in the initial proposal to exclude RCU domains. (The discussions covering this topic have been primarily face to face.)
2. Should the `rcu_obj_base` class provide an additional constructor that takes a deleter argument, thus allowing the deleter to be omitted from the call to `retire()` or `rcu_retire()`? (Note that this result in an extra store at constructor time when the deleter is specified at retire time.) Should this class delete copy constructors? (See the isocpp-parallel email thread in May 2018 with subject line “Feedback on Proposed wording for RCU (p0566R5)”.)
3. Should the type specifier for the deleter more closely resemble that of `unique_ptr`? (See the isocpp-parallel email thread in May 2018 with subject line “Feedback on Proposed wording for RCU (p0566R5)”.) The following options have been suggested:
    a. “D must be DefaultConstructible (for the constructors without an explicit deleter) (23.11.1.2.1p1 and p5)”, or
    b. “D must be constructible from std::forward<decltype(D)>(d), where d is the supplied deleter (which probably means MoveConstructible, but also accounts for reference types) (23.11.1.2.1p9-12)”
4. Should the `[[no_unique_address]]` attribute be applied to the deleter to take advantage of the C++20 equivalent of empty base optimization (EBO)? (See the isocpp-parallel email thread in May 2018 with subject line “Feedback on Proposed wording for RCU (p0566R5)”.)
5. What constraints need to be placed on the context in which deleters are invoked? If there are background threads, what control do users need over the time at which they are spawned and terminated? Should it be possible to construct an implementation with no background threads, and, if so, in what context do the destructors run? What

additional constraints (e.g., deadlock avoidance) need to be placed on users of implementations with no background threads? (See the isocpp-parallel email thread in July 2018 with subject line “Background threads and P0561”.)

6. Is special handling required for Microsoft dynamic-link libraries (DLLs), particularly surrounding special handling of background threads and pending deleters when DLLs are unloaded? Similarly, is special handling required to support clean shutdown of applications? (See the isocpp-parallel email thread in July 2018 with subject line “Background threads and P0561”.)
7. Can all major platforms support `latest<T>` instances with static storage duration? (See the isocpp-parallel email thread in July 2018 with subject line “Background threads and P0561”.)
8. Clean shutdown when deleters invoke rcu_retire(). Note that we can have cross-retires where hazard-pointer deleters invoke rcu_retire() and the corresponding RCU deleters retire a hazard pointer. Note that users can prevent clean shutdown by cascading retires forever. Should high-quality implementations do a best-effort attempt to shut down (even in the absence of rcu_barrier()), given that malicious users can prevent this. Note that this not unprecedented: Users can also prevent clean shutdown via infinite loops and various other forms of infinite recursion. However, even without infinite recursion, it is possible for a shutdown-time issues involving shutdown-time destructors using RCU after RCU itself has destructed (the Linux kernel avoids this by never destructing RCU). We hope that TS implementation experience will help us identify good strategies for best-effort shutdown. (Off-list discussion between Paul, Maged, and Michael.)
9. Should there be a polling interface having the functionality of rcu_synchronize()? For example, there might be an rcu_synchronize_start() free function that returned a cookie that could be passed to a boolean free function named rcu_synchronize_is_done().
10. Semantics of `rcu_retire`: Should it be OK to pass a single object to multiple invocations of `rcu_retire`? To an invocation of the `retire` member function and to one or more invocations of `rcu_retire`?

We therefore are not working to resolve these before the concurrency TS2 is created, but rather expecting that implementation experience based on the TS will shed light on the various possible resolutions.

6. Acknowledgements

The authors thank Olivier Giroux, Pablo Halpern, Lee Howes, Xiao Shi, Viktor Vafeiadis, Dave Watson, Daniel Kruegler, Tomsz Kaminski and other members of SG1 and LEWG for useful discussions and suggestions that helped improve this paper and its earlier versions.

7. References

RCU implementation: https://github.com/paulmckrcu/RCUCPPbindings (See Test/paulmck)

[N4618] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4618.pdf

[P0233] Hazard Pointers: Safe Resource Reclamation for Optimistic Concurrency
http://wg21.link/P0233

[P0461] Proposed RCU C++ API http://wg21.link/P0461

[P0566] Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU) http://wg21.link/P0566

[P0940] Concurrency TS is growing: Utilities and Data Structures http://wg21.link/P0940

8. Appendix

8.1 Removed Text

This is a holding area for text that has been removed, but might be needed elsewhere.

8.1.1. From Proposed Wording Section ?.1

1. RCU is a synchronization mechanism that can be used for linked data structures that are frequently read, but seldom updated. RCU does not provide mutual exclusion, but instead allows the user to defer specified actions to a later time at which there are no longer any RCU *read-side critical sections* that were executing at the time the deferral started. Threads executing within an RCU read-side critical section are called *readers*.
2. RCU read-side critical sections are designated using a class `std::rcu_reader`.
3. In one common use case (example shown below), RCU linked-structure updates are divided into two segments. The first segment is restricted to updates that do not disrupt readers (for example, removal from a linked structure). The second segment is disruptive to readers (for example, deleting items removed in the first segment), and must therefore be deferred until pre-existing readers (which might still be accessing the removed items) have ended.
   [Note— The following example shows how RCU allows updates to be carried out in two segments in the presence of concurrent readers. The reader function executes in one thread and the update function in another. The `rcu_reader` instance in `print_name` protects the referenced object `name` from being deleted by `rcu_retire` until the reader

has completed.

```
std::atomic<std::string *> name;

// called often and in parallel!
void print_name() {
  std::rcu_reader rr;
  std::string *s = name.load(std::memory_order_acquire);
 /* ...use *s... */
}

// called rarely
void update_name(std::string_view &nn) {
  // The first segment of update:
  std::string *new_name = new std::string(nn);
  std::string *s = name.exchange(new_name, std::memory_order_acq_rel);
  // The second segment of update is the deferred deleter.
  std::rcu_retire(s);
}
```

—end note]

Again, the first segment can be safely executed while RCU readers are concurrently traversing the same part of the linked structure, for example, removing some objects from a linked list. The second segment cannot be safely executed while RCU readers are accessing the removed objects; for example, the second segment typically deletes the objects removed by the first segment, in this case in a deferred fashion courtesy of `std::rcu_retire`. RCU can also be used to prevent RCU readers from observing transient atomic values, also known as the A-B-A problem.

4. A class `T` can inherit from `std::rcu_obj_base<T>` to inherit the `retire` member function and the intrusive machinery required to track retired objects. Alternatively, any class `T` can be passed to the `std::rcu_retire` free function template, whether it inherits from `std::rcu_obj_base<T>` or not. The free function is expected to have performance and memory-footprint advantages, but unlike the member function can potentially allocate. A deleter may be specified in the class template or to the retire function, and both types of retire functions arrange to invoke this deleter at a later time, when it can guarantee that no *read-side critical section* is still accessing (or can later access) the deleted data.
5. A `std::rcu_synchronize` free function blocks until all preexisting or concurrent *read-side critical sections* have ended. This function may be used as an alternative to the retire functions, in which case the `rcu_synchronize` follows the first (removal) segment of the update and precedes the second (deletion) segment of the update.
6. A `std::rcu_barrier` free function blocks until all happens-before [intro.multithreading] calls to `std::rcu_retire` have invoked and completed their deleters. This is helpful, for

instance, in cases where deleters have observable effects, or when it is desirable to bound undeleted resources, or when clean shutdown is desired.

8.1. Proposed Text

The current definition of `rcu_domain` addresses the feedback that the `rcu_reader` class not implement RAII on its own, but rather meet the requirements of `Cpp17BasicLockable` so as to leverage `lock_guard` [thread.lock.guard] and `scoped_lock` [thread.lock.scoped]. This seems to require that `lock` and `unlock` member functions be provided and that `scoped_lock` never be passed multiple types if one of those types is `rcu_reader`. This restriction is no problem because `rcu_reader` does not normally participate in deadlock cycles. The above wording is based on class mutex [thread.mutex.class].

Given definitions for the member functions, and given a constructor, this passes godbolt for `lock_guard, scoped_lock and unique_lock`.

The remainder of this appendix contains the old P1122R2 definition of rcu_reader.

?.2.2, class rcu_reader [rcu.reader]

This class provides RCU readers.

```
// ?.2.2, class rcu_reader
class rcu_reader {
public:
  // ?.2.2.1, rcu_reader: RCU readers
   rcu_reader() noexcept;
   explicit rcu_reader(defer_lock_t) noexcept;
   rcu_reader(const rcu_reader&) = delete;
   rcu_reader(rcu_reader&& other) noexcept;
   rcu_reader& operator=(const rcu_reader&) = delete;
   rcu_reader& operator=(rcu_reader&& other) noexcept;
   ~rcu_reader();
};
```

?.2.2.1, Construction and destruction [rcu.reader.cons]

```
  rcu_reader() noexcept;
```

1. Effects: Creates an active `rcu_reader`.
2. Synchronization: For each retire-function (`rcu_obj_base::retire` or `rcu_retire`) invocation such that this constructor does not happen after (C++Std [intro.races]) that

retire-function invocation, invocation of the corresponding deleter strongly happens after invocation of this object's destructor.

```
explicit rcu_reader(defer_lock_t) noexcept;
```

1. Effects: Creates an inactive `rcu_reader`.

```
rcu_reader(rcu_reader&& other) noexcept;
```

1. Effects: Creates an active `rcu_reader`. `other` becomes inactive.

```
~rcu_reader();
```

1. Effects: If the `rcu_reader` is active, causes it to become inactive.

?.2.2.2, Assignment [rcu.reader.assign]

```
rcu_reader& operator=(rcu_reader&& other) noexcept;
```

1. Effects: If `this` is active, it becomes inactive. In either case, `this` becomes active if and only if `other` was active, and `other` becomes inactive.