Document number: P0566R5
Date: 20180506 (pre-Rapperswil)
Project: Programming Language C++, WG21, SG1,SG14, LEWG, LWG
Authors: Michael Wong, Maged M. Michael, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher
Email: michael@codeplay.com, maged.michael@acm.org, paulmck@linux.vnet.ibm.com, gromer@google.com, ahh@google.com, arthur.j.odwyer@gmail.com, dshollm@sandia.gov, jfbastien@apple.com, hboehm@google.com, davidtgoldblatt@gmail.com, frank.birbacher@gmail.com
Reply to: michael@codeplay.com

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

1 Introduction

This is proposed wording for Hazard Pointers [P0233] and Read-Copy-Update [P0461]. Both are techniques for safe deferred resource reclamation for optimistic concurrency, useful for lock-free data structures. Both have been progressing steadily through SG1 based on years of implementation by the authors, and are in wide use in MongoDB (for Hazard Pointers) and Linux OS (RCU).

We 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. They can be split on request or on subsequent presentation.

This wording is based on n4618 draft [N4618]

2 History/Changes from Previous Release

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

- 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

- 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

Hazard Pointer and RCU are proposed additions to the C++ standard library, for the concurrency TS. It has been approved for addition through multiple SG1/SG14 sessions.
As hazard pointer and rcu are related, both being utility structures for deferred reclamation of concurrent data structures, we chose to do the wording together so that the similarity in structure and wording can be more apparent. They could be separated on request.

As both techniques are related to a concurrent shared pointer, it could be appropriate to be in Clause 20 with smart pointer, or Clause 30 with thread support, or even entirely in a new clause 31 labelled Concurrent Data Structures Library. However, we also believe Clause 20 does not seem appropriate as it does not cover the kind of concurrent data structures that we anticipate, while clause 30 is just about Threads, mutex, condition variables, and futures but does not cover data structures. 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.

4 Proposed wording

? Concurrent Data Structures Library [concur.data]

1. The following subclauses describe components to create and manage concurrent data structures, perform lock-free or lock-based concurrent execution, and synchronize concurrent operations.
2. If a data structure is to be accessed from multiple threads, then the program must be designed to ensure that any changes are correctly synchronized between threads.This clause describes data structures that have such synchronization built in, and do not require external locking.

?.1 Concurrent Data Structures Utilities [concur.util]

1. This component provides utilities for lock-free operations that can provide safe memory access, safe memory reclamation, and ABA safety.

?.1.1 Concurrent Deferred Reclamation Utilities [concur.reclaim]

1. The following subclauses describe low-level utilities that enable the user to schedule objects for destruction, while ensuring that they will not be destroyed until after all concurrent accesses to them have completed. These utilities are summarized in Table 1. These differ from `shared_ptr` in that they do not reclaim or retire their objects automatically, rather it is under user control, and they do not rely on reference counting.

Table 1 - Concurrent Data Structure Deferred Reclamation Utilities Summary

	Subclause	Header(s)	Feature Test Macro
?.1.1.2	Hazard Pointers	<hazptr>	__cpp_hazard_pointers
?.1.1.3	Read-Copy-Update	<rcu>	__cpp_read_copy_update

?.1.1.1 Concurrent Deferred Reclamation Utilities General [concur.reclaim.general]

Highly scalable algorithms often weaken mutual exclusion so as to allow readers to traverse linked data structures concurrently with updates. Because updaters reclaim (e.g., destroy) objects removed from a given structure, it is necessary to prevent objects from being reclaimed while readers are accessing them: Failure to prevent such accesses constitute use-after-free bugs. Hazard pointers and RCU are two techniques to prevent this class of bugs. Reference counting (e.g., `atomic_shared_pointer`) and garbage collection are two additional techniques.

? Hazard Pointers [hazptr]

1. The lifetime of each hazard pointer is split into a series of nonoverlapping epochs, with each epoch associated with a particular pointer to a hazptr_obj_base instance (or `NULL`). Consecutive epochs associated with the same instance are treated as distinct epochs. The initial epoch of each hazard pointer is associated with NULL.
2. Certain ways of starting a hazard pointer epoch associated with a pointer to an object will defer reclamation of that object until the end of the epoch.
3. The hazard pointer library allows for multiple hazard pointer domains, where the reclamation of objects in one domain is not affected by the hazard pointers in different domains. It is possible for the same thread to participate in multiple domains concurrently.
4. Hazard pointers are not directly exposed by this interface. Operations on hazard pointers are exposed through the `hazptr_holder` class. Each instance of `hazptr_holder` owns and operates on at most one hazard pointer. Each call to `protect`, `try_protect`, or `reset_protected` begins a new epoch and ends the previous one for the owned hazard pointer. Non-empty construction begins an epoch associated with NULL, and destruction of a non-empty hazptr_holder ends its epoch.
5. A hazard pointer domain contains a set of hazard pointers. A domain is responsible for reclaiming objects retired to it (by calling `hazptr_obj_base retire`), when such objects are not protected by hazard pointers that belong to this domain (including when this domain is destroyed).
6. A `hazptr_obj_base` O is *definitely reclaimable* in domain D at program point P if:
    a. there is a call to O.`retire`(reclaim, D), and it happens before P,
    b. [definitely not protected] For each epoch E in D associated with O, the end of E happens before P.
7. A `hazptr_obj_base` O is *possibly reclaimable* in domain D at program point P if
    a. There is a call to O.`retire`(reclaim, D) and P does not happen before the call,
    b. [possibly not protected] For each epoch E in D associated with O, P does not happen before the end of E.

[Note— The following example shows how hazard pointers allow updates to be carried out in the presence of concurrent readers. Each `hazptr_holder` instance in `print_name` is used through the call to `get_protected` to start an epoch associated with an object `*ptr` to protect the object from being reclaimed by `ptr->retire` until the end of the epoch.

```
struct Name : public hazptr_obj_base<Name> { /* details */ };
std::atomic<Name*> name;

// called often and in parallel!
void print_name() {
  std::hazptr_holder h = std::make_hazptr();
  Name* ptr = h.protect(name);
  /* ... safe to access *ptr ... */
```

```
}

// called rarely
void update_name(Name* new_name) {
  Name* ptr = name.exchange(new_name);
  ptr->retire();
}
```

—end note]

Header <hazptr> synopsis

```
namespace std {
namespace experimental {

// ?.1, Class hazptr_domain:
class hazptr_domain;

// ?.2, Default hazptr_domain:
hazptr_domain& default_hazptr_domain() noexcept;

// ?.?, Cleanup
void hazptr_cleanup(hazptr_domain& domain = default_hazptr_domain());

// ?.3, Class template hazptr_obj_base:
template <typename T, typename D = std::default_delete<T>>
  class hazptr_obj_base;

// ?.4, class hazptr_holder
class hazptr_holder;

// ?.5, make_hazptr
hazptr_holder make_hazptr(hazptr_domain& domain = default_hazptr_domain());

// ?.6, hazptr_holder swap
void swap(hazptr_holder&, hazptr_holder&) noexcept;

} // namespace experimental
} // namespace std
```

?.1 Class hazptr_domain [hazptr.domain]

1. The number of unreclaimed possibly reclaimable objects retired to a domain is bounded. The bound is implementation-defined. The bound is independent of other domains and may be a function of the number of hazard pointers in the domain, the number of threads that retire objects to the domain, and the number of threads that use hazard pointers that belong to the domain.

```
class hazptr_domain {
 public:

  // ?.1.1 constructor:
  explicit hazptr_domain(
      std::pmr::polymorphic_allocator<byte> poly_alloc = {});

  // disable copy and move constructors and assignment operators
  hazptr_domain(const hazptr_domain&) = delete;
  hazptr_domain(hazptr_domain&&) = delete;
  hazptr_domain& operator=(const hazptr_domain&) = delete;
  hazptr_domain& operator=(hazptr_domain&&) = delete;

  // ?.1.2 destructor:
  ~hazptr_domain();
 private:
  std::pmr::polymorphic_allocator<byte> alloc_;  // exposition only
};
```

?.1.1 `hazptr_domain` constructors [hazptr.domain.constructor]

```
explicit hazptr_domain(
      pmr::polymorphic_allocator<byte> poly_alloc = {});
```

1. Effects: Sets `alloc_` to `poly_alloc`.
2. Throws: Nothing.
3. Remarks: All allocation and deallocation of hazard pointers in this domain will use `alloc_`.

?.1.2 `hazptr_domain` destructor [hazptr.domain.destructor]

```
~hazptr_domain();
```

1. Requires: The destruction of all hazard pointers in this domain (including hazard pointers with epochs associated with NULL) and all `retire()` calls that take this domain as argument must happen before the destruction of the domain.
2. Effects: Deallocates all hazard pointer storage used by this domain. Reclaims any remaining objects that were retired to this domain.

3. Complexity: Linear in the number of objects retired to this domain that have not been reclaimed yet plus the number of hazard pointers contained in this domain.

?.2 Default `hazptr_domain`
[hazptr.default_domain]

```
hazptr_domain& default_hazptr_domain() noexcept;
```

1. Returns: A reference to the default `hazptr_domain`.

?.2 Cleanup
[hazptr.cleanup]

```
void hazptr_cleanup(hazptr_domain& domain = default_hazptr_domain());
```

1. Effects: For a set of hazptr_obj_base objects O in domain for which O.retire(reclaim, domain) has been called, ensures that O has been reclaimed. The set contains all definitely reclaimable objects at the point of cleanup, and may contain some possibly reclaimable objects.
2. Synchronization: The end of evaluation of each reclaim function of objects in the set synchronizes with the return from this call.

[*Note:* To avoid deadlock, this function must not be called while holding resources that may be required by such expressions. — *end note*]

?.3 Class template `hazptr_obj_base` [hazptr.base]
The base class template of objects to be protected by hazard pointers.

```
template <typename T, typename D = std::default_delete<T>>
class hazptr_obj_base {
 public:
  // retire
  void retire(D reclaim = {},
      hazptr_domain& domain = default_hazptr_domain());
  void retire(hazptr_domain& domain);
 protected:
  hazptr_obj_base() = default;
};
```

1. If this template is instantiated with a `T` argument that is not publicly derived from `hazptr_obj_base<T,D>` for some `D`, the program is ill-formed.
2. A client-supplied template argument `D` shall be a function object type 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.

3. 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.
4. A program may not add specializations of this template.

```
void retire(
    D reclaim = {}, hazptr_domain& domain = default_hazptr_domain());
```

1. Requires: `D` shall satisfy the requirements of MoveConstructible and such construction shall not exit via an exception. The *reclaim expression* `reclaim(static_cast<T*>(this))` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions.
2. Effects: Registers the expression `reclaim(static_cast<T*>(this))` to be evaluated asynchronously. For every hazard pointer in the domain, for epoch E associated with the value `static_cast<T*>(this)`:
   a.If the beginning of E happens before this call, then the end of E strongly happens before the evaluation of the reclaim expression.
   b. If E began as part of an evaluation of try_protect(ptr, src) returning true (for some src and ptr == `static_cast<T*>(this)`), let its associated atomic load operation be labelled A. If there exists an atomic modification B on src such that A observes a modification that is modification-ordered before B, and B happens before this call, then the end of E strongly happens before the evaluation of the expression. [Note: in typical use, a store to src sequenced before this call will be such atomic operation B.]

   The reclaim expression will be evaluated only once, and it will be evaluated by the evaluation of a `retire()` or `hazptr_cleanup()` operation on `domain`.

   This function may also evaluate any number of reclaim expressions for hazptr_obj_base objects possibly reclaimable in domain.

   [*Note:* To avoid deadlock, this function must not be called while holding resources required by such expressions. — *end note*]

```
void retire(
    hazptr_domain& domain);
```

1. Effects: Equivalent to
   `retire({}, domain);`

?.4 class hazptr_holder [hazptr.holder]

A `hazptr_holder` acts as a local handle on a hidden hazard pointer, which can be used to protect at most a single `hazptr_obj_base` object at a time. Every object of type `hazptr_holder` is either empty or *owns* exactly one hazard pointer. hazptr_holder and has no

protection against data races other than what is specified for the library generally ([res.on.data.races]).

Every hazard pointer is owned by exactly one hazptr_holder.

```
class hazptr_holder {
 public:
  // ?.4.1, Constructors
  hazptr_holder() noexcept;
  hazptr_holder(hazptr_holder&&) noexcept;

  // disallow copy operations
  hazptr_holder(const hazptr_holder&) = delete;
  hazptr_holder& operator=(const hazptr_holder&) = delete;

  // ?.4.2, destructor
  ~hazptr_holder();

  // ?.4.3, move assignment
  hazptr_holder& operator=(hazptr_holder&&) noexcept;

  // ?.4.4, empty
  bool empty() const noexcept;

  // ?.4.5, protect
  template <typename T>
    T* protect(const atomic<T*>& src) noexcept;

  // ?.4.6, try_protect
  template <typename T>
    bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;

  // ?.4.7, reset_protected
  template <typename T>
  void reset_protected(const T* ptr) noexcept;
  void reset_protected(nullptr_t = nullptr) noexcept;

  // ?.4.8, swap
  void swap(hazptr_holder&) noexcept;
};
```

?.4.1 `hazptr_holder` constructors [hazptr.holder.constructors]

```
hazptr_holder() noexcept;
```

1. Effects: Constructs an empty hazptr_holder.

```
hazptr_holder& hazptr_holder(hazptr_holder&& other) noexcept;
```

1. Effects: If `other` is empty, constructs an empty `hazptr_holder`. Otherwise, constructs a hazptr_holder that owns the pointer originally owned by `other`. `other` becomes empty.

?.4.2 `hazptr_holder` destructor [hazptr.holder.destructor]

```
~hazptr_holder();
```

1. Effects: If `*this` is not empty, destroys the owned hazard pointer which ends its current epoch.

?.4.3 hazptr_holder assignment [hazptr.holder.assignment]

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

1. Effects: If `this == &other`, no effect. Otherwise, if `other` is not empty, `*this` takes ownership of the hazard pointer originally owned by `other`, and `other` becomes empty. If `*this` was not empty before the call, destroys the owned hazard pointer which ends its current epoch
2. Returns: `*this.`

?.4.4 hazptr_holder empty [hazptr.holder.empty]

```
bool empty() const noexcept;
```

1. Returns: `true` if and only if hazptr_holder is empty. [*Note:* An empty `hazptr_holder` is different from a nonempty `hazptr_holder` that owns a hazard pointer with epoch associated with null value. An empty `hazptr_holder` does not own any hazard pointers. — *end note*]

?.4.5 hazptr_holder protect [hazptr.holder.protect]

```
template <typename T>
  T* protect(const atomic<T*>& src) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Equivalent to

```
T* ptr = src.load(memory_order_relaxed);
while (!try_protect(ptr, src)) {}
return ptr;
```

?.4.6 hazptr_holder try_protect [hazptr.holder.try_protect]

```
template <typename T>
```

```
bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects:
    a. Ends the owned hazard pointer's current epoch, and starts a new one.
    b. Performs an atomic acquire load on `src`. If `src == ptr`, then the hazard pointer's new epoch is associated with the value `ptr`, and `try_protect()` returns true. Otherwise, the new epoch is associated with `NULL`, and `try_protect()` returns false.
    c. Sets ptr to the value read from src.
3. Returns: The result of the comparison. [*Note:* It is possible for `try_protect` to return `true` when `ptr` is a null pointer. — *end note*]
4. Complexity: Constant.

?.4.7 hazptr_holder reset_protected [hazptr.holder.reset]

```
template <typename T>
void reset_protected(const T* ptr) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Ends the owned hazard pointer's current epoch, and begins a new one associated with `ptr`.

```
void reset_protected(nullptr_t = nullptr) noexcept;
```

1. Requires: `*this` is not empty.
2. Effects: Ends the owned hazard pointer's current epoch, and begins a new one associated with NULL..

?.4.8 hazptr_holder swap[hazptr.holder.swap]

```
void swap(hazptr_holder& other) noexcept;
```

1. Effects: Swaps the owned hazard pointer and the domain of this object with those of the other object. [*Note:* The owned hazard pointers, if any, remain unchanged during the swap and continue to protect the respective objects that they were protecting before the swap, if any. — *end note*]
2. Complexity: Constant.

?.5 make_hazptr [hazptr.make]

```
hazptr_holder make_hazptr(hazptr_domain& domain = default_hazptr_domain());
```

1. Effects: Constructs a hazard pointer from `domain`, and returns a `hazptr_holder` that owns it.
2. Throws: Any exception thrown by `domain.alloc_.allocate()`.

?.6 hazptr_holder specialized algorithms [hazptr.holder.special]

```
void swap(hazptr_holder& a, hazptr_holder& b) noexcept;
```

1. Effects: Equivalent to `a.swap(b)`.

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

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 an RAII class `std::rcu_reader`.
3. In one common use case (example shown below), RCU linked-structure updates are divided into two segments.
   [Note— The following example shows how RCU allows updates to be carried out 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 *new_name) {
  std::string *s = name.exchange(new_name, std::memory_order_acq_rel);
  std::rcu_retire(s);
}
```

   —end note]
   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. 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 make it work. 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. Both types of retire functions arrange to invoke the 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 previous (*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.

Header <rcu> synopsis

```
namespace std {
namespace experimental {

// ?.2, class template rcu_obj_base
template<typename T, typename D = default_delete<T>>
    class rcu_obj_base;

// ?.2.2, class rcu_reader: RCU reader as RAII
class rcu_reader;
void swap(rcu_reader& a, rcu_reader& b) noexcept;

// ?.2.3 function rcu_synchronize
void rcu_synchronize() noexcept;

// ?.2.4 function rcu_barrier
void rcu_barrier() noexcept;

// ?.2.5 function template rcu_retire
template<typename T, typename D = default_delete<T>>
void rcu_retire(T* p, D d = {});
} // namespace experimental
} // namespace std
```

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

Objects of type `T` to be protected by RCU inherit from `rcu_obj_base<T>`. Note that `rcu_obj_base<T>` has no non-default constructors or destructors.

```
template<typename T, typename D = default_delete<T>>
    class rcu_obj_base {
public:
  // ?.2.1.1, rcu.base.retire: Retire a removed object and pass the
  // responsibility for reclaiming it to the RCU library.
  void retire(
      D d = {});
};
```

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.

?.2.1.1, rcu_obj_base retire [rcu.base.retire]

```
   void retire(
      D d = {}) noexcept;
```

1. Effects: Equivalent to `rcu_retire(this, d)`
2. Throws: nothing. [*Note:* This implies it may not allocate memory via `operator new`. *end note*]

?.2.2, class rcu_reader [rcu.reader]

This class provides RAII RCU readers.

```
// ?.2.2, class template rcu_readers
class rcu_reader {
public:
  // ?.2.k2.1, rcu_reader: RAII RCU readers
   rcu_reader() noexcept;
   rcu_reader(std::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, class template rcu_reader constructors [rcu.reader.cons]

```
rcu_reader() noexcept;
```

1. Effects: Creates an active `rcu_reader` that is associated with a new RCU read-side critical section.
2. Postconditions: For each retire-function (`std::rcu_obj_base::retire` or `std::rcu_retire`) invocation such that this constructor does not happen before (C++Std [intro.races]) that retire-function invocation, prevents the corresponding deleter tfrom being invoked.

```
rcu_reader(std::defer_lock_t) noexcept;
```

1. Effects: Creates an inactive `rcu_reader`.

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

1. Effects: Creates an active `rcu_reader` that is associated with the RCU read-side critical section that was associated with `other`. If `this` was already associated with an RCU read-side critical section, that critical section ends as described in the destructor. The `rcu_reader other` becomes inactive.

?.2.2.2, class template rcu_reader assignment [rcu.reader.assign]

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

1. Effects: If `this` is active, the corresponding RCU read-side critical section ends as described in the destructor. In either case, `this` become active and holds the RCU read-side critical section corresponding to `other`, and `other` becomes inactive.

?.2.2.4, class template rcu_reader swap [rcu.reader.swap]

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

1. Effects: Swaps `this` and `other`, thus swapping their RCU read-side critical section states.

```
void swap(rcu_reader& a, rcu_reader& b) noexcept; // free function
```

1. Effects: Swaps `a` and `b` thus swapping their RCU read-side critical section states.

?.2.3, function rcu_synchronize [rcu.synchronize]

```
void rcu_synchronize() noexcept;
```

1. Effects: Equivalent to:
```
std::atomic<bool> b;
rcu_retire(&b, [](std::atomic<bool> *b) {
  b->store(true);
});
while (!b->load());
```
2. Throws: nothing.

?.2.4, function rcu_barrier [rcu.barrier]

```
void rcu_barrier() noexcept;
```

1. Effects: For each invocation of a retire function (`std::rcu_obj_base::retire` or `std::rcu_retire`) that happens before this call, blocks until the corresponding deleter has completed. May additionally wait for the completion of the corresponding deleter of any retire function call that does not happen after this call.
2. Synchronization: The completion of each such deleter strongly happens before the return from `rcu_barrier`.

?.2.5, function template rcu_retire [rcu.retire]

```
template<typename T, typename D = default_delete<T>>
void rcu_retire(T* p, D d = {});
```

1. Requires: `D` shall satisfy the requirements of MoveConstructible and such construction shall not exit via an exception. The expression `d(p)` shall be well-formed, shall have well-defined behavior, and shall not throw exceptions. The object referenced by `p` must not be passed to any other invocation of a retire function.
2. Effects: Causes `d(p)` to be invoked later at an unspecified point on unspecified execution agents. Guarantees that for each instance R of `rcu_reader`, one of two things hold:
   - `rcu_retire` strongly happens before R's constructor
   - R's destructor strongly happens before the invocation of the deleter.

   [Note: If R's constructor happens before `rcu_retire`, then R's destructor strongly happens before the deleter. If the deleter happens before R's destructor, then `rcu_retire` strongly happens before R's constructor. --- end note]

5. Acknowledgements

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

6. References

Hazptr implementation:
https://github.com/facebook/folly/blob/master/folly/experimental/hazptr/hazptr.h

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