Document number: P1121R3
Date: 2021-04-09
Project: Programming Language C++, WG21
Authors: Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Mathias Stearn, Jens Maurer
Email: maged.michael@gmail.com, michael@codeplay.com, paulmck@kernel.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, redbeard0531+isocpp@gmail.com, jens.maurer@gmx.net
Reply to: maged.michael@gmail.com, michael@codeplay.com, paulmck@kernel.org

Hazard Pointers: Proposed Interface and Wording for Concurrency TS 2

1 Introduction

This paper contains proposed interface and wording for hazard pointers [1], a technique for safe deferred reclamation. This wording is based on N4700 draft [2]. An implementation is in the Folly open source library [3]

The proposal was voted to be forwarded by SG1 to LEWG in Rapperswil in June 2018, and was voted to be forwarded by LEWG to LWG in San Diego in November 2018.

Drafts of this paper were reviewed by LWG on 2021-03-19, 2021-03-26, and 2021-04-09. This paper was voted on 2021-04-09 by LWG to be forwarded to plenary for inclusion in Concurrency TS 2.

1.1 Motivation

Under optimistic concurrency, threads[1] may use shared resources concurrently with other threads that may make such resources unavailable for further use. Care must be taken to reclaim such resources only after they are guaranteed that no threads will subsequently use them.

More specifically, concurrent dynamic data structures that employ optimistic concurrency allow threads to access dynamic objects concurrently with threads that may remove such objects. Without proper precautions, it is generally unsafe to reclaim the removed objects, as they may be accessed subsequently by threads that hold references to them. Solutions for the safe reclamation problem can also be used to prevent the ABA problem, a common problem under optimistic concurrency.

There are several methods for safe deferred reclamation. The main methods aside from automatic garbage collection are reference counting, RCU (read-copy-update), and hazard pointers. Each method has its pros and cons and none of the methods provides the best features in all cases. Therefore, it is desirable to offer users the opportunity to choose the most suitable methods for their use cases.

2 History/Changes from Previous Release

Until the June 2018 Rapperswil meeting the interface and wording for hazard pointers were presented together with those for RCU (Read-Copy-Update) [4] in P0566 [5] with associated Bugzilla Bug #382. For the history of P0566 see the last revision P0566R5 (pre-Rapperswil). Earlier interface proposals are in P0233 [6]. This paper is a revision of the hazard pointer related parts of P0566. The RCU related parts are now in P1122, and the chapter headings from P0566 are now in P0940.

2021-04 Changes in [P1121R3] from [P1121R2] (after review by LWG)

- Pervasive changes. The following are the most significant changes.
- Added general design information and examples.

[1] Throughout this document, we use to term *thread* to refer to any thread of execution, including language-level threads, processes, and signal handlers.

- Changed the term epoch to protection epoch and made the use of the term more precise.
- Made the specification and use of the base object type and derived types more precise. Added the term hazard-protectable.
- Consolidated the definition of possibly-reclaimable.
- Separated the domain default constructor from the explicit one with an optional parameter. Made the domain constructors `noexcept`.

2021-01 Changes in [P1121R2] from [P1121R1] (pre-Kona)

- Updates are based on feedback from LWG for D1122R3 (RCU).
- Add `namespace std::experimental::inline concurrency_v2`.
- Change `class` to `struct`.
- Make `hazard_pointer_clean_up noexcept`.
- Make `retire noexcept`.
- Several additions to the wording for `retire`.

2019-01 Changes in [P1121R1] (pre-Kona) from [P1121R0] (pre-San Diego)

- Removed Section 3 of P1121R1, which provided detailed background for LEWG review in San Diego.
- Changed instances of "Requires" to "Mandates" and "Expects" according to N4762 [structure.specifications].

2018-11 LEWG Review in San Diego

- LEWG voted to approve the API changes proposed in P1121R1 and to forward the proposal to LWG for wording review towards inclusion in Concurrency TS 2.

2018-10 Changes in [P1121R0] (pre-San Diego) from [P0566R5] (pre-Rapperswil)

- Edited the wording of hazard_pointer_obj_base retire(): Added a clarifying note. Removed instances of the word "then". Added the word "reclaim" to clarify the meaning of "expression".
- Changed "hazptr" to "hazard_pointer" in class and function names.
- Changed the class name "hazptr_holder" to "hazard_pointer".
- Changed the member function name "reset_protected" to "reset_protection".
- Changed the free function name "hazptr_cleanup" to "hazard_pointer_clean_up".

2018-06 LEWG Review in Rapperswil

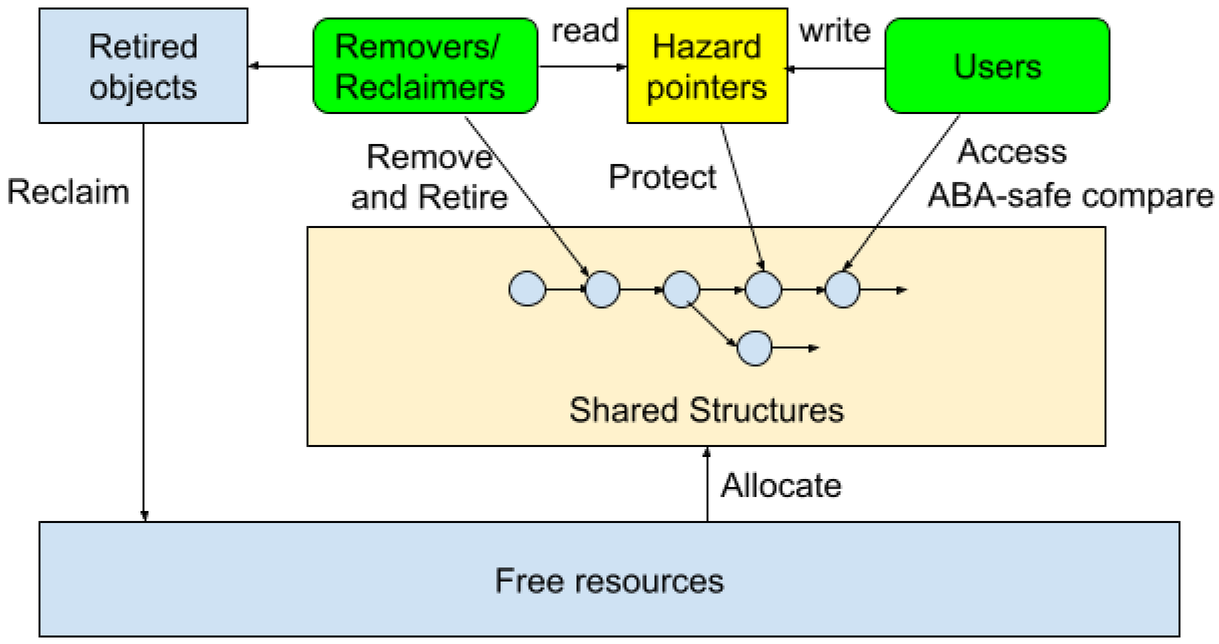
- See details in Section 3 of P1121R0.

2018-06 SG1 Review in Rapperswil

- SG1 voted to forward the proposal to LEWG, provided that the following changes to the wording of hazptr_obj_base retire are made: Add a clarifying note. Remove instances of the word "then". Add the word "reclaim" to clarify the meaning of "expression".

3. Hazard Pointers

A hazard pointer is a single-writer multi-reader pointer that can be owned by at most one thread at any time. Only the owner of the hazard pointer can set its value, while any number of threads may read its value. A thread that is about to access dynamic objects optimistically acquires ownership of a set of hazard pointer to protect such objects from being reclaimed. The owner thread sets the value of a hazard pointer to point to an object in order to indicate to concurrent threads — that may remove such object — that the object is not yet safe to reclaim.



Hazard pointers are owned and written by threads that act as users/protectors (i.e., protect removable objects from unsafe reclamation in order to use such objects) and are read by threads that act as removers/reclaimers (i.e., may remove and try to reclaim objects). Removers retire removed objects to the hazard pointer library (i.e., pass the responsibility for reclaiming the objects to the library code rather than normally by user code). The set of protector and remover threads may overlap, so the same thread may write to its own hazard pointers to protect objects and read the hazard pointers including those of other threads when attempting to reclaim retired objects.

The key rule of the hazard pointers method is that **a retired object can be reclaimed only after it is determined that no hazard pointers have been pointing continuously to it from a time before its retirement**.

3.1. Basic Mechanism

Protection:

- By setting a hazard pointer HP to the address of an object A, the owner of the hazard pointer is telling all threads: “if you (collectively) remove object A after I have set HP to the address of A, and then retire A, then don’t reclaim A as long as HP continues to point to A”.

Deferred reclamation:

- After accumulating a number of retired objects (for the sake of amortization):
    - Extract the set of retired objects.
    - Read (no atomicity needed) the values all hazard pointers and keep a private set of non-null values.
    - For each retired object, lookup its address in the private set of values read from hazard pointers:
        - If not found, reclaim the object.
        - If found, put the object back in the set of retired objects.

3.2. Domains

The hazard pointers method allows the presence of multiple hazard pointer domains, where the safe reclamation of resources in one domain does not require checking all the hazard pointers in different domains. It is possible for the same thread to participate in multiple domains concurrently. A domain can be specific to one or more resources, or can encompass all sharing among multiple processes in a system.

3.3. Main Structures and Operations

The main structures of the hazard pointers method are:

- **Hazard pointers:** pointer-sized variables.
- **Retired objects** awaiting reclamation.
- **Container structures** for hazard pointer records and removed objects.

The key operations are:

- **Allocate** a hazard pointer.

- **Acquire** ownership of a hazard pointer.
- **Set** the value of a hazard pointer to protect an object.
- **Clear** the value of a hazard pointer.
- **Release** ownership of a hazard pointer.
- **Retire** a removed object.
- **Read** the value of a hazard pointer.

The rationale for using a polymorphic allocator for the allocation of hazard pointers (at least for the TS proposal) is to avoid the virality of allocator template parameters and to allow custom domains to use custom allocation (e.g., a fixed number of preallocated hazard pointers).

3.4. Pros and Cons

The main advantages of the hazard pointers method are that:

1. The number of removed objects that are not yet reclaimed is bounded.
2. Readers do not interfere with each other or with writers
3. Cache friendly access patterns.
4. Constant time complexity for traversal and expected amortized constant time for determination of safe reclamation per retired object.

4. Comparison of Deferred Reclamation Methods

	Reference Counting	**Split Reference Counting**	**RCU**	**Hazard Pointers**
Unreclaimed objects	**Bounded**	**Bounded**	Unbounded	**Bounded**
Contention among readers	Can be very high	Can be very high	**No contention**	**No contention**
Traversal speed	Atomic updates	Atomic updates	**No or low overhead**	Low overhead
Reference acquisition	**Unconditional**	**Unconditional**	**Unconditional**	Conditional
Automatic reclamation	**Yes**	**Yes**	No	No
Blocking while protecting objects	**Yes**	**Yes**	No or complicated	**Yes**

Advantages

5 Use Examples

5.1 Copy-on-Write

```
struct Block : hazard_pointer_obj_base<Block> { V val_; ... };

atomic<Block*> block_;

U reader_op() {
  hazard_pointer h = make_hazard_pointer();
  Block* p = h.protect(block_);
  return f(p); // safe to access *p
} // RAII end of protection

void writer(Block* newb) { // May be called concurrently with readers
  Block* oldb = block_.exchange(newb);
  oldb->retire(); // reclaim *oldb when safe
}
```

5.2 Search Ordered Single-Writer Singly-Linked List

```
struct Node : hazard_pointer_obj_base<Node> {
  T elem_;
  atomic<Node*> next_;
  Node(T e, Node* n) : elem_(e), next_(n) {}
};

atomic<Node*> head_{nullptr};

bool contains(const T& val) const {
  /* Two hazard pointers for hand-over-hand traversal */
  hazard_pointer hptr_prev = make_hazard_pointer();
  hazard_pointer hptr_curr = make_hazard_pointer();
  while (true) {
    atomic<Node*>* prev = &head_;
    Node* curr = prev->load(std::memory_order_acquire);
    while (true) {
      if (!curr) return false;
      if (!hptr_curr.try_protect(curr, *prev)) break;
      Node* next = curr->next_.load(std::memory_order_acquire);
      if (prev->load(std::memory_order_acquire) != curr) break;
      if (curr->elem_ >= val) return curr->elem_ == val;
      prev = &(curr->next_);
      curr = next;
      swap(hptr_curr, hptr_prev);
    }
  }
}
// For more details see
https://github.com/facebook/folly/blob/master/folly/synchronization/example/H
azptrSWMRSet.h
```

6 Proposed wording

? Hazard Pointers [hazptr]

1. A hazard pointer is a single-writer multi-reader pointer that can be owned by at most one thread at any time. Only the owner of the hazard pointer can set its value, while any number of threads may read its value. The owner thread sets the value of a hazard pointer to point to an object in order to indicate to concurrent threads — that may delete such an object — that the object is not yet safe to delete.
2. A class type T is *hazard-protectable* if it has exactly one public base class of type `hazard_pointer_obj_base<T,D>` for some D and no base classes of type `hazard_pointer_obj_base<T',D'>` for any other combination `T'`, `D'`. An object is *hazard-protectable* if it is of hazard-protectable type.
3. The span between creation and destruction of a hazard pointer *h* is partitioned into a series of *protection epochs*; in each protection epoch, *h* either is *associated with* a hazard-protectable object, or is *unassociated*. Upon creation, a hazard pointer is unassociated. Changing the association (possibly to the same object) initiates a new protection epoch and ends the preceding one.
4. A hazard pointer *belongs to* exactly one *domain*.
5. An object of type `hazard_pointer` is either empty or *owns* a hazard pointer. Each hazard pointer is owned by exactly one object of type `hazard_pointer`.
   [*Note:* An empty `hazard_pointer` object is different from a `hazard_pointer` object that owns an unassociated hazard pointer. An empty `hazard_pointer` object does not own any hazard pointers. — *end note*]

6. An object x of hazard-protectable type T is *retired* to a domain with a deleter of type D when the member function hazard_pointer_obj_base<T,D>::retire is invoked on x. Any given object x shall be retired at most once.
7. A retired object x is *reclaimed* by invoking its deleter with a pointer to x.
8. A hazard-protectable object x is *definitely reclaimable* in a domain *dom* with respect to an evaluation A if:
    a. x is not reclaimed, and
    b. x is retired to *dom* in an evaluation that happens before A, and
    c. for all hazard pointers h that belong to *dom*, the end of any protection epoch where h is associated with x happens before A.
9. A hazard-protectable object x is *possibly reclaimable* in domain *dom* with respect to an evaluation A if:
    a. x is not reclaimed; and
    b. x is retired to *dom* in an evaluation R and A does not happen before R; and
    c. for all hazard pointers h that belong to *dom*, A does not happen before the end of any protection epoch where h is associated with x; and
    d. for all hazard pointers *h* belonging to *dom* and for every protection epoch E of *h* during which *h* is associated with x:
        i. A does not happen before the end of E, and
        ii. if the beginning of E happens before x is retired, the end of E strongly happens before A, and
        iii. if E began by an evaluation of try_protect with argument src, label its atomic load operation L. If there exists an atomic modification B on src such that L observes a modification that is modification-ordered before B, and B happens before x is retired, the end of E strongly happens before A. [Note: In typical use, a store to src sequenced before retiring x will be such an atomic operation B.]

        [Note: The latter two conditions convey the informal notion that a protection epoch that began before retiring x, as implied either by the happens-before relation or the coherence order of some source, delays the reclamation of x. -- end note]

[Example— The following example shows how hazard pointers allow updates to be carried out in the presence of concurrent readers. The object of type hazard_pointer in print_name protects the object *ptr from being reclaimed by ptr->retire until the end of the protection epoch.

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

// called often and in parallel!
void print_name() {
  hazard_pointer h = make_hazard_pointer();
  Name* ptr = h.protect(name); /* Protection epoch starts  */
  /* ... safe to access *ptr ... */
} /* Protection epoch ends. */

// called rarely, but possibly concurrently with print_name
void update_name(Name* new_name) {
  Name* ptr = name.exchange(new_name);
  ptr->retire();
}
```
—end example]

?.1 Header <hazard_pointer> synopsis [hazptr.syn]

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

  // ?.2, class hazard_pointer_domain
  class hazard_pointer_domain;

  // ?.3, Default hazard_pointer_domain
  hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

  // ?.4, Clean up
  void hazard_pointer_clean_up(
    hazard_pointer_domain& domain = hazard_pointer_default_domain())
    noexcept;

  // ?.5, class template hazard_pointer_obj_base
  template <typename T, typename D = default_delete<T>>
    class hazard_pointer_obj_base;

  // ?.6, class hazard_pointer
  class hazard_pointer;

  // ?.7, Construct non-empty hazard_pointer
  hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

  // ?.8, Hazard pointer swap
  void swap(hazard_pointer&, hazard_pointer&) noexcept;

}
```

?.2 Class hazard_pointer_domain [hazptr.domain]
? 2.1 General [hazptr.domain.general]

1. The number of unreclaimed possibly-reclaimable objects retired to a domain is bounded. The bound is implementation-defined. [Note: The bound can be independent of other domains and can be a function of the number of hazard pointers belonging to the domain, the number of threads that retire objects to the domain, and the number of threads that use hazard pointers belonging to the domain. -- end note]
2. Concurrent access to a domain does not incur a data race ([intro.races)].

```
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();
};
```

?.2.2 `Member functions` [hazptr.domain.mem]

```
hazard_pointer_domain() noexcept;
```

1. Effects: Equivalent to
   `hazard_pointer_domain({});`

```
explicit hazard_pointer_domain(
  pmr::polymorphic_allocator<byte> poly_alloc) noexcept;
```

1. Remarks: All allocation and deallocation related to hazard pointers belonging to this domain use a copy of poly_`alloc`.

```
~hazard_pointer_domain();
```

1. Preconditions: All hazard pointers belonging to `*this` have been destroyed.
2. Effects: Reclaims all objects retired to this domain that have not yet been reclaimed.

?.3 Default `hazard_pointer_domain` [hazptr.domain.default]

```
hazard_pointer_domain& hazard_pointer_default_domain() noexcept;
```

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

2. Remarks: The default domain has an unspecified allocator and has static storage duration. The initialization of the default domain strongly happens before this function returns; the sequencing is otherwise unspecified.

?.4 Clean up [hazptr.cleanup]

```
void hazard_pointer_clean_up(
  hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
```

1. Effects: May reclaim possibly-reclaimable objects retired to `domain`.
2. Postconditions: All definitely-reclaimable objects retired to `domain` have been reclaimed.
3. Synchronization: The completion of the deleter for each reclaimed object synchronizes with the return from this function call.

?.5 Class template `hazard_pointer_obj_base` [hazptr.base]

```
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;
protected:
  hazard_pointer_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 `hazard_pointer_obj_base` is undefined.
3. `D` shall meet the requirements for Cpp17DefaultConstructible and Cpp17MoveAssignable.
4. `T` may be an incomplete type.

```
void retire(
  D d = D(),
  hazard_pointer_domain& domain = hazard_pointer_default_domain()) noexcept;
```

1.

2. Mandates: `T` is a hazard-protectable type.
3. Preconditions: `*this` is a base class subobject of an object `x` of type T. `x` is not retired. Move-assigning `D` from `d` does not throw an exception. The expression `d(addressof(x))` has well-defined behavior and does not throw an exception.
4. Effects: Move-assigns `d` to `deleter`, thereby setting it as the deleter of `x`, then retires `x` to `domain`.
5. Invoking the `retire` function may reclaim possibly-reclaimable objects retired to `domain`.

```
void retire(hazard_pointer_domain& domain) noexcept;
```

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

?.6 Class `hazard_pointer` [hazptr.holder]

```
class hazard_pointer {
public:
  hazard_pointer() noexcept;

  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;
};
```

?.6.1 Constructors [hazptr.holder.ctor]

```
hazard_pointer() noexcept;
```

1. Postconditions: `*this` is empty.

```
hazard_pointer(hazard_pointer&& other) noexcept;
```

1. Postconditions: If `other` is empty, `*this` is empty. Otherwise, `*this` owns the hazard pointer originally owned by `other`; `other` is empty.

?.6.2 Destructor [hazptr.holder.dtor]

```
~hazard_pointer();
```

1. Effects: If `*this` is not empty, destroys the hazard pointer owned by `*this`, thereby ending its current protection epoch.

?.6.3 Assignment [hazptr.holder.assign]

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

1. Effects: If `this == &other` is `true`, no effect. Otherwise, if `*this` is not empty, destroys the hazard pointer owned by `*this`, thereby ending its current protection epoch.
2. Postconditions: If `other` was empty, `*this` is empty. Otherwise, `*this` owns the hazard pointer originally owned by `other`. If `this != &other` is `true`, `other` is empty.
3. Returns: `*this.`

?.6.4 Member functions [hazptr.holder.mem]

```
[[nodiscard]] bool empty() const noexcept;
```

1. Returns: `true` if and only if `*this` is empty.

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

1. Effects: Equivalent to

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

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

1. Mandates: T is a hazard-protectable type.
2. Preconditions: `*this` is not empty.
3. Effects:
   a. Initializes a variable `old` of type `T*` with the value of `ptr`.
   b. Evaluates the function call `reset_protection(old)`.

   c. Assigns the value of `src.load(std::memory_order_acquire)` to `ptr`.
   d. If `old == ptr` is `false`, evaluates the function call `reset_protection()`.
4. Returns: `old == ptr`. [*Note:* It is possible for `try_protect` to return `true` when `ptr` is a null pointer. — *end note*]
5. Complexity: Constant.

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

1. Mandates: `T` is a hazard-protectable type.
2. Preconditions: `*this` is not empty.
3. Effects: If `ptr` is a null pointer value, invokes `reset_protection()`. Otherwise, associates the hazard pointer owned by `*this` with `*ptr`, thereby ending the current protection epoch.

```
void reset_protection(nullptr_t = nullptr) noexcept;
```
1. Preconditions: `*this` is not empty.
2. Postconditions: The hazard pointer owned by `*this` is unassociated.

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

1. Effects: Swaps the hazard pointer ownership of this object with that of `other`.
   [*Note:* The owned hazard pointers, if any, remain unchanged during the swap and continue to be associated with the respective objects that they were protecting before the swap, if any. No protection epochs are ended or initiated. — *end note*]
2. Complexity: Constant.

?.7 make_hazard_pointer [hazptr.make]
```
hazard_pointer make_hazard_pointer(
  hazard_pointer_domain& domain = hazard_pointer_default_domain());
```

1. Effects: Constructs a hazard pointer belonging to `domain`.
2. Returns: A `hazard_pointer` object that owns the newly-constructed hazard pointer.
3. Throws: Any exception thrown by the allocator of `domain`.

?.8 hazard_pointer specialized algorithms [hazptr.holder.special]
```
void swap(hazard_pointer& a, hazard_pointer& b) noexcept;
```

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

Acknowledgements

The authors thank Keith Bostic, Olivier Giroux, Pablo Halpern, Davis Herring, Lee Howes, Bronek Kozicki, Nathan Myers, Xiao Shi, Tim Song, Viktor Vafeiadis, Tony Van Eerd, Dave Watson, Anthony Williams and other members of SG1, SG14, LEWG, and LWG for useful discussions and suggestions that helped improve this paper and its earlier versions.

References

[1] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects." *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (2004): 491-504.

[2] N4700 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4700.pdf

[3] Hazard Pointer Implementation:
https://github.com/facebook/folly/blob/master/folly/synchronization/Hazptr*

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

[5] P0566 Proposed Wording for Concurrent Data Structures: Hazard Pointer and ReadCopyUpdate (RCU). http://wg21.link/P0566

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