Authors: Maged M. Michael, Michael Wong, Paul McKenney, Arthur O'Dwyer, David Hollman,
Geoffrey Romer, Andrew Hunter
Project: Programming Language C++, SG14/SG1 Concurrency, LEWG

Hazard Pointers

Safe Resource Reclamation for Optimistic Concurrency

1.   Introduction

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. See paper P0232R0 (Concurrency ToolKit for Structured Deferral/Optimistic Speculation)[3] for a detailed comparative analysis of these methods along with atomic shared pointers which is based on an earlier paper by Paul McKenney [1]. This proposal focuses on the hazard pointer method [2].

We propose adding hazard pointers as a library as part of a collection of Concurrency ToolKit methods (P0232R0).

1.1 History

2017-10-15 R6:
- Updated the discussion of the performance of hazard pointers in Sections 2.3, 3.8. and 7 to reflect recent potential performance gains by using `sys_membarrier()`.
- Changed the syntax for the polymorphic allocator passed to the constructor of `hazptr_domain`. The constructor is no longer constexpr.

---

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

- 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.

2017-07-30 R5:
- ● Allow hazptr_holder to be empty. Add a move constructor, empty state constructor, move assignment operator, and a bool operator to check for empty state.
- ● A call to reset(), try_protect(), or get_protected() is undefined behavior if the hazptr_holder is empty.
- ● Add overload of hazptr_obj_base retire().

2017-06-18 R4:
- Reverted the generic template parameter of hazptr_owner::try_protected() and hazptr_owner::get_protected to std::atomic to avoid re-specifying atomics in the draft wording.
- Renamed hazptr_owner to hazptr_holder.
- Changed hazptr_holder::set(const T* ptr) to reset(const T* ptr) and hazptr_holder::clear() to reset(nullptr_t = nullptr).
- Made hazptr_holder a class and not a class template, while making get_protected(), try_protect(), and reset(const T*) member function templates.
- Updated the reference in Section 5 to the corresponding draft standard wording (P0566R1).
- Updated the information about implementation options and a prefered performant implementation in Section 9.

2017-02-06: R3:
- Replaced std::atomic with a generic template parameter to allow different atomic types. Updated Sections 8 and 10 accordingly.
- Added a reference in Section 5 to draft standard wording (P0566R0).

2016-10-17:R2
- Renamed haz_ptr_control_block hazptr_domain
- Renamed haz_ptr_guard hazptr_owner
- Renamed haz_ptr_obj hazptr_obj_base
- hazptr_domain constructor takes an optional argument, a memory_resource for allocating and deallocating hazard pointers.
- hazptr_obj_base has two template parameters: object type and deleter type
- Renamed and moved the reclaim() member function template of haz_ptr_control_block to be the member function retire() of the hazptr_obj_base.
- Removed all optional thread-specific parts of the interface for clarity at this point.

- Renamed haz_ptr_guard::protect() hazptr_owner::try_protect()
- Added the function get_protected() to hazptr_owner.
- Added a free function template swap() for swapping hazptr_owner instances.

2016-09-21: Review of a partial update.at SG14 CPPCON

2016-05-30:R1
- Renamed haz_ptr haz_ptr_guard and made it a class template
- Using default allocator by default instead of malloc and free
- Changed the haz_ptr_control_block from default constructor to template c'tor
- Moved reclaim() and rem_policy to haz_ptr_control_block
- Made set() take T* as parameter instead of void*
- Removed allocation and deallocation function objects from haz_ptr_guard c'tor
- haz_ptr_obj became a class template
- Added noexcept and const wherever applicable
- Removed examples and some optional (nice to have) functions and parameters until a core interface is approved

2016-03-04: 1st review by SG1; positive support to continue work; but interface needs to be patterned to C++, as well as other comments; reviewed at SG14 GDC 2016 with Jeffrey, Hans, Michael, Lee, JF

2016-02-12: R0 with initial proposal

1.2 Planned Changes

The following is a list of possible changes before the Albuquerque meeting:
- Refinement of the null constructor for hazptr_holder.
- Refinement of overloads of hazptr_obj_base retire().
- Revision of the wording in P0566 for more precise specification of memory ordering.
- Consideration of adding a non-intrusive interface.
- Ensuring that the hazard pointer interface satisfies the requirements for a performant cell<T> implementation (P0561 describes the cell<T> deferred reclamation proposal).

2.   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 (typically one or two for linked data structures) 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.
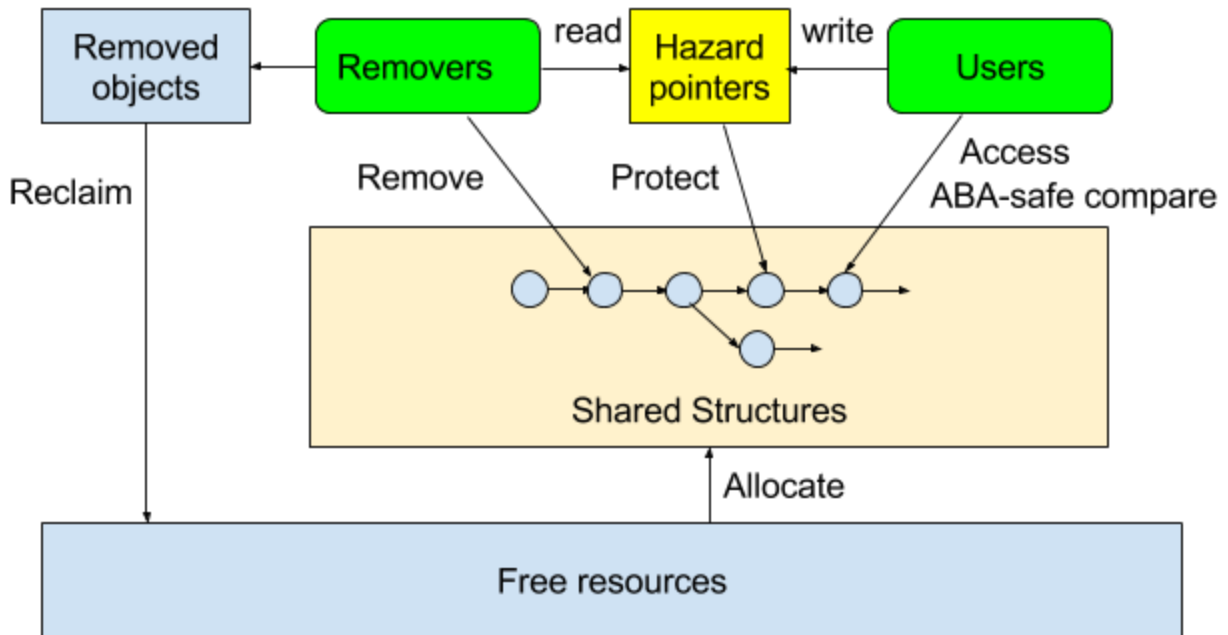
Hazard pointers are owned and written by threads that act as users (i.e., may use removable objects) and are read by thread that act as removers (i.e., may remove objects). The set of user and remover threads may overlap, so the same thread may write to its own hazard pointers when using objects and read the hazard pointers including those of other threads when reclaiming removed objects.

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

In addition to the primary use cases for hazard pointers for memory reclamation, objects protected by hazard pointers could represent other reclaimable resources such as files, ports, and devices. Also, the method can be used by signal handlers and among processes as well as among language-level threads.

2.1. Hazard Pointer 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.

2.2. Main Structures and Operations

The main structures of the hazard pointers method are:
- **Hazard pointers:** pointer-sized variables.
- **Removed 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.
- Request the **deferred reclamation** of a removed object.
- **Read** the value of a hazard pointer.

Design details are discussed in following sections.

2.3. 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 time for) reclamation
5. Its operations are lock-free[2] (mostly wait-free), and therefore it is suitable for use in non-blocking operations that are required to be async signal-safe or immune to asynchronous process termination.

A potential disadvantage of the hazard pointers method on systems without support for `sys_membarrier()` is that each traversal incurs a store-load memory order fence, when using the method's basic form (without blocking).

3.   Design Considerations

3.1. Progress Guarantees

Some use cases of hazard pointers require that all operations be non-blocking from end to end. An operation is non-blocking if it is guaranteed to complete in a finite number of its own steps

---

[2] Provided that atomic pointer and integer types are lock-free.

when it runs without interference from other operations, regardless of where other threads are blocked. Lock-free progress is a stronger form of non-blocking progress; it further guarantees collective forward progress even in the presence of interference among threads. Wait-free progress (even stronger) guarantees that an operation will complete in a finite number of its own steps. The hazard pointers method can have end-to-end lock-free implementations.

Non-blocking progress is an essential requirement for operations to be async signal safe. It is also essential for guaranteeing availability of resources in cases where processes may be killed asynchronously while sharing such resources.

The main requirements for guaranteeing lock-free progress are:
- Not using thread-local storage (unless TLS is guaranteed to be non-blocking). This implies the need to implement non-blocking container structures for removed objects.
- Not using the default memory allocator, as it is unlikely to be completely non-blocking. This implies the need to design the library interface in a way that allows the specification of custom allocation and deallocation functions, as well as avoidance of memory allocation when possible.

3.2. Thread Types

Some use cases are by thread types other than typical language-level threads — in particular, signal handlers and processes. Support for signal handlers requires implementation options that avoid thread-local storage and that allow the use of non-blocking allocators.[3] Support for processes require allowing custom allocation and deallocation functions that can operate on shared memory (and other shared system resources protected by hazard pointers).

3.3. Memory Allocation and Deallocation

There are several cases (as mentioned above) that require the use of custom allocators:
- The deferred reclamation of objects that are not allocated using malloc (e.g., new).
- End-to-end non-blocking progress is required.
- Sharing resources among processes.

Accordingly, the implementation must provide the capability to specify custom allocation and deallocation functions in various parts of the library interface.

3.4. Reclamation Frequency

There is a trade-off between:
- The upper bound on the number of removed objects that are not yet reclaimed.
- The time complexity of reclamation per object

---

[3] c.f. p0270r0 and minutes from Jacksonville: http://wiki.edg.com/bin/view/Wg21jacksonville/P0270

- Using thread-local storage.

For the purposes of this discussion, let N be the maximum number of hazard pointers (in a domain), and let M be the number of remover threads.

Using thread-local storage (assuming wait-free TLS), the M removers can perform bulk reclamation after accumulating a number of removed objects that is at least $N+\Theta(N)$ (e.g., $2*N$). In such case the upper bound on the number of unreclaimed removed objects is $O(M*N)$ and the amortized expected time per reclaimed object is constant. The progress is wait-free and contention-free.

Without using thread-local storage, removed objects are inserted in shared lock-free structures. The worst-case unreclaimed removed objects can be bounded by $O(N)$, but contention becomes possible and progress becomes lock-free instead of wait-free.

3.5. Number of Hazard Pointers and Thread Caching

Using a fixed number of hazard pointers simplifies the implementation, but it restricts use and can be inconsistent with non-blocking progress if a larger number of hazard pointers is needed. For the sake of flexibility, the implementation must allow the dynamic allocation of hazard pointers.

Caching released hazard pointers between operations can minimize contention related to acquiring hazard pointers. Caching can be done transparently in the library implementation using TLS, however TLS is not always guaranteed to be non-blocking. Of course the programmer can cache hazard pointers explicitly at the cost of some inconvenience and taking responsibility for explicitly releasing hazard pointers instead of depending on their automatic release by the library.

3.6. Thread-Local Storage

As discussed above the use of thread-local storage has pros and cons. It reduces or eliminates contention in acquiring hazard pointers and allows wait-free progress (if TLS is wait-free). On the other hand, it is incompatible with async signal safety, and TLS implementations are not guaranteed to be non-blocking.

Due to the performance advantages of using TLS, the library implementation should allow the programmer to choose implementation paths that benefit from TLS when suitable, and avoid TLS when incompatible with the use case.

3.7. Exceptions

The sources of exceptions in implementations of the hazard pointers method are related to memory allocation, in particular the allocation of hazard pointers. All other operations can avoid memory allocation exceptions at some performance cost in the worst case when allocation is impossible.

Programmers concerned about such exceptions (for example, in real-time code) can guarantee that hazard pointer operations will not throw if they meet certain conditions. Implementations can guarantee that the total number of hazard pointers never shrinks throughout the lifetime of the associated domain. Therefore, programmers can pre-allocate the needed number of hazard pointers and then release them, knowing that all these hazard pointers will remain available for reallocation throughout the lifetime of the associated domain, provided that care is taken in managing thread caching of hazard pointers. Alternatively, programmers can avoid creating hazard pointers ahead of time by creating a simple wait-free allocator that manages sufficient memory to allocate a large number of hazard pointers (and therefore is guaranteed not to throw) and provide this allocator as an argument to the hazard pointer constructor.

3.8. Primitives and Dependencies

The hazard pointers method requires the use of atomic primitives on pointers and `size_t` variables and memory ordering primitives. The method has no direct dependencies on any system calls.

The method in its purely non-blocking form incurs a store-load fence. This fence becomes unnecessary if the method is used in more restricted cases such as inside critical sections protected by a lock, or by using interrupts to enforce ordering only when a remover thread is about to inspect the hazard pointers. Optionally, the method can use the sys_membarrier() system call when available to reduce the user-side memory ordering cost in exchange for increasing the less frequent remover-side memory ordering cost.

4. Design Overview

Based on the above considerations and with a goal of maximizing usability, we believe that hazard pointer implementations should have the following features or policies:
- Use TLS for performance but provide a path that is TLS-free.
- Provide an end-to-end lock-free path.
- Allow custom allocation and deallocation function objects.
- Support an end-to-end async signal safe path.
- Support multi-process sharing.
- Support multiple hazard pointer domains.

- Support dynamic hazard pointer allocation.
- Do not throw exceptions except in hazard pointer allocation, and provide use conditions that guarantee that hazard pointer constructors will not throw exceptions.
- Support hazard pointer caching.
- Support automatic release of hazard pointers.
- Support an interface that can avoid the store-load fence when not needed.
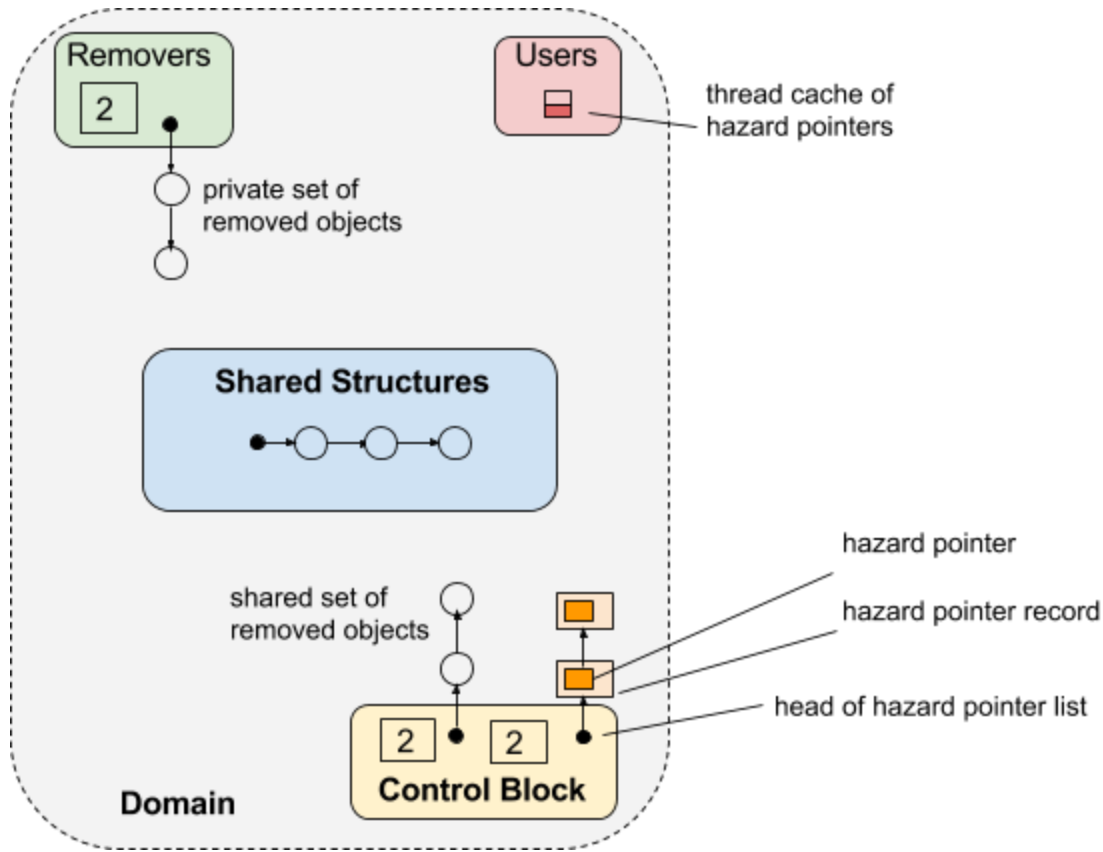
The above diagram shows the main components of the hazard pointer method's design:
- **Hazard pointer domains**: Multiple domains may be present concurrently. Threads may participate in multiple domains in different roles as users, removers, or both. There is one default domain per process.
- The **hazard pointer control block** is the defining component of a domain. It manages the **set of all hazard pointer records** in the domain, and the **set of retired objects** that are protected by these hazard pointers at some point.
- A **hazard pointer record** contains a **hazard pointer** and an indicator of whether the hazard pointer is free or owned by a user thread. Hazard pointers may point to **removed objects** or **reachable objects in shared structures**.
- **User threads** (optionally) manage a small **thread cache** for hazard pointer records.
- **Remover threads** (optionally) manage a **private set of removed objects**.

5.   Impact on the Standard

Hazard pointers will be a pure library addition (with no core language elements), likely in Clause 30 "Thread support library" [thread], or else located in a new clause titled "Concurrency Support Library". It does require Clause 29 "Atomic operations library" [atomics] for atomic operations and memory ordering. Draft standard wording is in P0566R0, *Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU)* [4].

6.   Existing Implementations and Target Workloads

The hazard pointer method is used in several proprietary products that require high-availability and non-blocking progress for safe resource management. Other uses are in supporting lock-free access in key-value stores and applications with soft real-time requirements. The method is used in the MongoDB/WiredTiger open-source NoSQL database [5].

There are several open source implementations, such as Concurrency Kit [6], Concurrency Building Blocks [7], libcds [8], and Parallelism Shift [9]. These implementations provide different interfaces that have their pros and cons. We aim to maximize flexibility, and use variations of the flexible features of these interfaces and avoid restrictive features, such as supporting regular threads only, or requiring the numbers of hazard pointers to be fixed beforehand.

7.   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 progress	Either blocking or lock-free with limited reclamation	**Lock-free**	**Wait-free**	**Lock-free.**
Reclamation progress	Either blocking or lock-free with limited reclamation	**Lock-free**	Blocking	**Lock-free**

Traversal speed	Atomic updates	Atomic updates	**No or low overhead**	Low overhead when using sys_membarrier
Reference acquisition	**Unconditional**	**Unconditional**	**Unconditional**	Conditional
Automatic reclamation	**Yes**	**Yes**	No	No

Advantages

8.   Proposal for Adding a Hazard Pointer Library

8.1. `hazptr_domain` class

This class is the root of all shared hazard pointer data structures in a domain. There is exactly one instance of this class in each domain. It is included in the library header in order to allow the programmer to create and control hazard pointer domains.

```cpp
class hazptr_domain {
 public:
  explicit hazptr_domain(
      pmr::polymorphic_allocator<byte>* = {}) noexcept;
  ~hazptr_domain();

  hazptr_domain(const hazptr_domain&) = delete;
  hazptr_domain(hazptr_domain&&) = delete;
  hazptr_domain& operator=(const hazptr_domain&) = delete;
  hazptr_domain& operator=(hazptr_domain&&) = delete;
};

hazptr_domain& default_hazptr_domain() noexcept;

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

The **constructor** takes a pointer to pmr::polymorphic_allocator<byte> as a parameter to use for allocating hazard pointer structures.

The **destructor** destroys all shared hazard pointer structures and reclaims all retired objects that are managed by this domain.

This class does not allow copy and move constructors and assignment operators.

The free function `default_hazptr_domain()` returns a reference to the default hazptr_domain.

The free function **hazptr_barrier()** returns only after all object retired to the specified domain (or the default domain otherwise) have been reclaimed. This function can be called only when there is exactly one thread remaining to have ever retired any objects to the domain.

8.2. `hazptr_obj_base` class template

This is the base class template for objects protected by hazard pointers.

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

The **retire()** function passes the responsibility for reclaiming this object to the hazptr library. The function takes up to two optional arguments:
- A reference to a hazptr_domain
- A deleter to be used to reclaim this object when it is safe to do so according to the hazard pointer system.

The deleter type must be nothrow move-constructible; i.e., `std::is_nothrow_move_constructible<D>::value` must be true.

Usage example:
```
class Node : public hazptr_obj_base<Node, MyReclaimer<Node>> { ...
```

8.3. `hazptr_holder` class

This template manages all operations on individual hazard pointers (allocation, acquisition, setting, clearing, and release).

```
class hazptr_holder {
 public:
  explicit hazptr_holder(hazptr_domain& domain = default_hazptr_domain());
```

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

    ~hazptr_holder();

    hazptr_holder(const hazptr_holder&) = delete;
    hazptr_holder& operator=(const hazptr_holder&) = delete;

    hazptr_holder& operator=(hazptr_holder&&) noexcept;

    bool empty() const noexcept;

    template <typename T>
    T* get_protected(const atomic<T*>& src) noexcept;
    template <typename T>
    bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
    template <typename T>
    void reset(const T* ptr) noexcept;
    void reset(nullptr_t = nullptr) noexcept;
    void swap(hazptr_holder&) noexcept;
};

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

The **default constructor** takes a reference to a hazptr_domain (by default the result of default_hazptr_domain()). It automatically allocates a hazard pointer that belongs to that domain. Depending on the memory resource used by the domain, the constructor may throw. The **move constructor** takes a reference to another hazptr_holder `other` as an an argument. It constructs the hazptr_holder with the contents of `other` using move semantics, `other` becomes empty.

The static function `make_empty()` constructs an empty hazptr_holder.

The **destructor** automatically clears and releases the owned hazard pointer, if any.

Copy constructor and assignment operator are disallowed.

The **move assignment operator** transfers ownership of the pointer originally owned by other, and other becomes empty if *this != other. Otherwise no effect.

The member function `empty()` returns true if and only if the hazptr_holder is empty. Note that an empty hazptr_holder is different from a nonempty hazptr_holder that owns a hazard pointer with null value. An empty hazptr_holder does not own any hazard pointers.

The member function template `get_protected()` takes one argument `src`, an atomic pointer to the template parameter `T`. The function returns a pointer value read from `src` that is guaranteed to be protected by the owned hazard pointer. A protected pointer value is safe to dereference and comparisons with it are ABA-safe until the hazard pointer is modified or cleared provided that removers use only the member function `retire()` of `hazptr_obj_base` to request the reclamation of `*ptr`.

The member function template `try_protect()` takes two arguments, a pointer value `ptr` and an atomic pointer `src`. The function returns true only if it can guarantee that the owned hazard pointer is protecting the pointer `ptr`. Otherwise, this function returns false. Note that the function may return true even when `ptr` has the value null.

The member function template `reset()` takes one argument, a pointer value `ptr`. The function sets the owned hazard pointer to the value `ptr`. An overload of `reset()` that takes an optional nullptr argument sets the owned hazard pointer to nullptr.

Calls to get_protected(), try_protect(), and reset() must be by a nonempty hazptr_holder.

The member function `swap()` takes one argument, a reference to another `hazptr_holder`. It swaps ownership of hazard pointers between this and the other hazptr_holder. The owned hazard pointers remain unmodified during the swap and continue to protect the respective objects that they were protecting before the swap, if any.

A free function `swap()` swaps two `hazptr_holder` objects with the same effect as calling the `swap()` member function of the first with the second as an argument, or vice versa.

9. Sample Interface and Implementation

A C++ Standard Library sample interface code is in Appendix A. An implementation of the interface with use examples is available at (https://github.com/facebook/folly/blob/master/folly/experimental/hazptr/).
Multiple implementation options are supported: with and without thread caching of hazard pointers, and with and without use of asymmetric memory barriers. The most reader performant implementation uses thread caching and asymmetric memory barriers.

10. Appendix A: Draft Library Interface Header

```
#include <atomic>
#include <memory>
#include <memory_resource>

class hazptr_domain {
 public:
  explicit hazptr_domain(
      polymorphic_allocator<byte>* = {}) noexcept;
  ~hazptr_domain();

  hazptr_domain(const hazptr_domain&) = delete;
  hazptr_domain(hazptr_domain&&) = delete;
  hazptr_domain& operator=(const hazptr_domain&) = delete;
  hazptr_domain& operator=(hazptr_domain&&) = delete;
};

hazptr_domain& default_hazptr_domain();

void hazptr_barrier(hazptr_domain& = default_hazptr_domain());

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

class hazptr_holder {
 public:
  explicit hazptr_holder(
      hazptr_domain& domain = default_hazptr_domain());
  hazptr_holder(hazptr_holder&& other) noexcept;
  static hazptr_holder make_empty() noexcept;

  ~hazptr_holder();
```

```
    hazptr_holder(const hazptr_holder&) = delete;
    hazptr_holder& operator=(const hazptr_holder&) = delete;

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

    bool empty() const noexcept;

    template <typename T>
    T* get_protected(const atomic<T*>& src) noexcept;
    template <typename T>
    bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
    template <typename T>
    void reset(const T* ptr) noexcept;
    void reset(nullptr_t = nullptr) noexcept;
    void swap(hazptr_holder&) noexcept;
};

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

11. Acknowledgement

Thanks SG1 and SG14 members for reviewing earlier versions of the proposal. We especially thank JF Bastien, Jeffrey Yasskin, Pablo Halpern, Lee Howes, David Goldblatt, Dave Watson, Xiao Shi for their comments and suggestions for the interface.

12. References

[1] Paul E McKenney. "Structured deferral: synchronization via procrastination." *Communications of the ACM* 56.7 (2013): 40-49.
[2] Maged M Michael. "Hazard pointers: Safe memory reclamation for lock-free objects." *Parallel and Distributed Systems, IEEE Transactions on* 15.6 (2004): 491-504.
[3] P0232R0,, P. McKenney, M. Wong, M. Michael, A Concurrency ToolKit for Structured Deferral or Optimistic Speculation, Feb. 2016.
[4] P0566R1, Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU), June 2017.
[5] https://github.com/wiredtiger/wiredtiger/blob/master/src/support/hazard.c.
[6] http://concurrencykit.org/
[7] http://amino-cbbs.sourceforge.net/
[8] http://libcds.sourceforge.net/
[9] http://www.johantorp.com/