Document Number: P2530R0
Date: 2022-02-14
Project WG21, SG1
Reply to: Maged M. Michael
Meta (Facebook)
maged.michael@gmail.com

Why Hazard Pointers Should Be in C++26

Authors:
Maged M. Michael, Michael Wong, Paul McKenney, Andrew Hunter, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Mathias Stearn

Email:
`maged.michael@gmail.com`, `michael@codeplay.com`, `paulmck@kernel.org`, `andrewhhunter@gmail.com`, `dhollman@google.com`, `cxx@jfbastien.com`, `hboehm@google.com`, `davidtgoldblatt@gmail.com`, `frank.birbacher@gmail.com`, `redbeard0531+isocpp@gmail.com`

Contents

1 Introduction

We propose the hazard pointer safe deferred reclamation technique [1] for inclusion into C++26. This paper contains proposed interface and wording for hazard pointers and rationale for its inclusion in C++26. The wording is based on N4700 ("Working Draft, Standard for Programming Language C++"). An implementation is in the Facebook Folly open source library [2] and has been in heavy use in production since 2017.

The proposal is a subset of the hazard pointer interface and wording in N4895 ("Working Draft, Extensions to C++ for Concurrency Version 2") (henceforth referred to as TS2), which is based on P1121R3 ("Hazard Pointers: Proposed Interface and Wording for Concurrency TS 2"). The proposed interface and wording are in Section 5.

2 Rationale for inclusion in C++26

This proposal is based on the experience gained from hazard pointer implementation in the Facebook Folly open-source library (since 2016) and its heavy use in production (since 2017). The Folly implementation includes the TS2 interface (which was functionally frozen in late 2017) and significant subsequent functional extensions. The implementation of the selected subset of TS2 proposed for C++26 has been stable and in heavy continuous use in production since 2017.

What are the reasons for selecting the proposed subset of the TS2 interface for C++26?

- The subset supports a large part of hazard pointer usage in production.
- The subset interface and implementation have been stable for years under heavy use in production.
- The changes to the existing TS2 wording are straightforward (as can be seen in Section 5 of this paper) and are therefore expected to require little of the committee's time.

Which parts of the TS2 interface are omitted from this proposal for C++26 and why?

- This proposal omits **custom domains**:
  - The Folly implementation experience has been that we were able to improve and extend the default domain, without needing custom domains. This does not preclude the need for custom domains for some use cases (e.g., async-signal-safety).
  - That is, we did not include custom domain for lack of evidence they are not necessary for common usage.
  - This proposal does not preclude adding custom domains in the future, as evident in TS2.
- This proposal also omits **global cleanup**:
  - The TS2 interface includes a global cleanup function (hazard_pointer_clean_up) which guarantees that all reclaimable retired objects are reclaimed before the function returns. We refer to this mode of reclamation as synchronous reclamation in contrast to the default asynchronous reclamation, where retired objects are only guaranteed to be reclaimed by the end of the program.
  - In 2018 we introduced to Folly cohort-based synchronous reclamation. Cohort-based reclamation is faster and more scalable than global cleanup and can be used conservatively. It has been in heavy use in production since 2018.
  - We do not include cohort-based synchronous reclamation in this proposal for the following reasons:
    - The wording is expected to be complex and would take a nontrivial amount of committee time.
    - Use cases that do not require synchronous reclamation are important. There is no need to delay inclusion into the IS to support such cases.
    - This proposal is amenable to adding future support for synchronous reclamation.

Why the rush to get it into C++26 when the Concurrency TS 2 is not published as yet (as of early 2022)?

- We were able to use the TS2 interface (functionally unchanged since 2017) and gain feedback for the Facebook Folly library implementation under heavy use in production.
- This production experience has enabled us to reduce the TS2 interface already to a stable subset that is currently heavily used in production.
- The plan was always to include a basic interface into C++ IS, and we have achieved that with this experience.
- With hazard pointers in C++26, it enables us to evolve it further and we have a growing list of features and improvements which would be hard to add without being in the IS.
- We can continue to incorporate feedback experience from other usage and evolve it.
- The straightforward changes to the interface surface area (all subtractions) means that the wording review effort is small, and can take little time to be reviewed for C++26 IS inclusion.
- The subset proposed for C++26 is amenable to potential future extensions based on feedback to TS2.
- Note that this selection does not constitute a rejection of the components of TS2 not selected for the C++26 proposal. Rather, we cannot recommend the excluded components with as much confidence for standardization as the selected subset. Excluded components remain in the Folly implementation. There is still value in receiving feedback about the excluded components of TS2.

3 Hazard pointer overview

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 acquires ownership of hazard pointer(s) 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 accessors/protectors (i.e., protect removable objects from unsafe reclamation in order to access 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.

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.**

The core basic mechanism of the hazard pointer technique is the interaction between protection and deferred reclamation.

Protection:

- By setting a hazard pointer H 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 H to the address of A, and then you retire A, then don't reclaim A as long as H 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 of all hazard pointers and keep a private set of such 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.

The main user operations on hazard pointers are:

- Acquiring ownership of a hazard pointer.
- Setting the value of a hazard pointer to protect an object.
- Clearing the value of a hazard pointer.
- Releasing ownership of a hazard pointer.

The main user operation on objects protectable by hazard pointers in addition to regular operations such as reading, writing and removal is:

- Retiring a removed object.

The main internal library operations are:

- Allocation of hazard pointers.
- Deferred reclamation.

4 Use examples

4.1 Copy-on-write

```
struct Block : hazard_pointer_obj_base<Block> { /* members */ };

atomic<Block*> block_;

template <typename Func>
Result reader_op(Func fn) {
  hazard_pointer h = make_hazard_pointer();
  Block* p = h.protect(block_);
  return fn(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
}
```

4.2 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 github.com/facebook/folly/blob/master/folly/synchronization/example/HazptrSWMRSet.h

5 Proposed Wording [saferecl]

Omissions from TS2 are marked in two ways, either by being in red ~~strikethrough~~ or in red and labelled *[Omitted]*.

5.1 Hazard pointers [saferecl.hp]

5.1.1 General [saferecl.hp.general]

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 1*: 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:~~

(8.1) — ~~`x` is not reclaimed, and~~

(8.2) — ~~`x` is retired to *dom* in an evaluation that happens before A, and~~

(8.3) — ~~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:

(9.1) — `x` is not reclaimed; and

(9.2) — `x` is retired ~~to *dom*~~ in an evaluation R and A does not happen before R; and

(9.3) — 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

(9.4) — for all hazard pointers h ~~belonging to *dom*~~ and for every protection epoch E of h during which h is associated with `x`:

(9.4.1) — A does not happen before the end of E, and

(9.4.2) — if the beginning of E happens before `x` is retired, the end of E strongly happens before A, and

(9.4.3) — 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 2*: In typical use, a store to `src` sequenced before retiring `x` will be such an atomic operation B. — *end note*]

[*Note 3*: 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 1*: 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*]

5.1.2 Header `<hazard_pointer>` synopsis [saferecl.hp.syn]

```
namespace std::experimental::inline concurrency_v2 /* [Omitted] */ {
  // 5.1.3, class hazard_pointer_domain [Omitted]
  class hazard_pointer_domain;

  // 5.1.4, Default hazard_pointer_domain [Omitted]
  hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

  // 5.1.5, Clean up [Omitted]
  void hazard_pointer_clean_up(hazard_pointer_domain& domain = hazard_pointer_default_domain())
    noexcept;
  // 5.1.6, class template hazard_pointer_obj_base
  template <typename T, typename D = default_delete<T>> class hazard_pointer_obj_base;

  // 5.1.7, class hazard_pointer
  class hazard_pointer;

  // 5.1.8, Construct non-empty hazard_pointer
  hazard_pointer make_hazard_pointer(
      hazard_pointer_domain& domain = hazard_pointer_default_domain() // [Omitted]
      );
```

```
    // 5.1.9, Hazard pointer swap
    void swap(hazard_pointer&, hazard_pointer&) noexcept;
  }
```

5.1.3 Class `hazard_pointer_domain` *[Omitted]* [saferecl.hp.domain]

5.1.3.1 General [saferecl.hp.domain.general]

1 The number of unreclaimed possibly-reclaimable objects retired to a domain is bounded. The bound is implementation-defined.

[*Note 1*: 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 (C++20 §6.9.2.1).

```
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();
};
```

5.1.3.2 Member functions [saferecl.hp.domain.mem]

```
hazard_pointer_domain() noexcept;
```

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

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

2 *Remarks*: All allocation and deallocation related to hazard pointers belonging to this domain use a copy of `poly_alloc`.

```
~hazard_pointer_domain();
```

3 *Preconditions*: All hazard pointers belonging to `*this` have been destroyed.

4 *Effects*: Reclaims all objects retired to this domain that have not yet been reclaimed.

5.1.4 Default `hazard_pointer_domain` *[Omitted]* [saferecl.hp.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.

5.1.5 Clean up *[Omitted]* [saferecl.hp.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.

5.1.6 Class template `hazard_pointer_obj_base` [saferecl.hp.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() /* [Omitted] */) noexcept;
  void retire(hazard_pointer_domain& domain) noexcept; // [Omitted]
protected:
  hazard_pointer_obj_base() = default;
private:
  D deleter; // exposition only
};
```

1 A client-supplied template argument `D` shall be a function object type (C++20 §20.14) 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() /* [Omitted] */) noexcept;
```

5 *Mandates*: `T` is a hazard-protectable type.

6 *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.

7 *Effects*: Move-assigns `d` to *`deleter`*, thereby setting it as the deleter of `x`, then retires `x` ~~`to domain`~~.

8 Invoking the retire function may reclaim possibly-reclaimable objects retired to `domain`.

```
void retire(hazard_pointer_domain& domain) noexcept; // [Omitted]
```

9 ~~*Effects*: Equivalent to `retire(D(), domain)`.~~

5.1.7 Class `hazard_pointer` [saferecl.hp.holder]

5.1.7.1 Synopsis [saferecl.hp.holder.syn]

```
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;
};
```

5.1.7.2 Constructors [saferecl.hp.holder.ctor]

```
hazard_pointer() noexcept;
```

1 *Postconditions*: `*this` is empty.

`hazard_pointer(hazard_pointer&& other) noexcept;`

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

5.1.7.3 Destructor [saferecl.hp.holder.dtor]

`~hazard_pointer();`

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

5.1.7.4 Assignment [saferecl.hp.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`.

5.1.7.5 Member functions [saferecl.hp.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;`

2 *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;`

3 *Mandates*: `T` is a hazard-protectable type.

4 *Preconditions*: `*this` is not empty.

5 *Effects*:

(5.1) — Initializes a variable `old` of type `T*` with the value of `ptr`.

(5.2) — Evaluates the function call `reset_protection(old)`.

(5.3) — Assigns the value of `src.load(std::memory_order_acquire)` to `ptr`.

(5.4) — If `old == ptr` is false, evaluates the function call `reset_protection()`.

6 *Returns*: `old == ptr`.

[*Note 1*: It is possible for `try_protect` to return `true` when `ptr` is a null pointer. — *end note*]

7 *Complexity*: Constant.

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

8 *Mandates*: `T` is a hazard-protectable type.

9 *Preconditions*: `*this` is not empty.

10 *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;`

11 *Preconditions*: `*this` is not empty.

12 *Postconditions*: The hazard pointer owned by `*this` is unassociated.

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

13 *Effects*: Swaps the hazard pointer ownership of this object with that of other.

[*Note 2*: 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*]

14 *Complexity*: Constant.

5.1.8 `make_hazard_pointer` [saferecl.hp.make]

```
hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain() /* [Omitted] */ );
```

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~~ May throw `bad_alloc` if memory for the hazard pointer could not be allocated.

5.1.9 `hazard_pointer` specialized algorithms [saferecl.hp.special]

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

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

Acknowledgments

The authors thank Keith Bostic, Olivier Giroux, Pablo Halpern, Davis Herring, Lee Howes, Bronek Kozicki, Jens Maurer, Nathan Myers, Arthur O'Dwyer, Geoffrey Romer, 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 components of this proposal.

References

[1] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[2] Facebook Folly library: Hazard pointer implementation (`folly/synchronization/Hazptr*`).