Document number: P3427R8
Date: 2026-08-12
Project: Programming Language C++, LWG
Authors: Maged M. Michael, Michael Wong, Paul McKenney, Mark de Wever
Email: maged.michael@gmail.com, fraggamuffin@gmail.com, paulmck@kernel.org, koraq@xs4all.nl

Hazard Pointer Synchronous Reclamation

Table of Contents

History

The Varna 2023 plenary voted in favor of including hazard pointers in the C++26 standard library ([2023-06 LWG Motion 7] P2530R3 Hazard Pointers for C++26).

P3135R1 was presented to SG1 in Tokyo 2024, reviewing potential extensions of the P2530R3 C++26 interface, and proposing two of those for inclusion in the standard library. The proposal for extending the P2530R3 C++26 interface to support synchronous reclamation was voted on by SG1 with unanimous consent:

```
we want to continue work on hazard pointer cohorts (synchronous reclamation) for C++26,
with association to the cohort registered at the time of retire()
```

R0

P3427R0 was a follow-up on P3135R1, focusing on extending the P2530R3 C++26 hazard pointer interface to support synchronous reclamation, revised to take into account the feedback from SG1:

- Associate a hazard-protectable object with a cohort at the time of its call to retire().

R1

R1 added draft wording to R0.
D3427R1 was reviewed by SG1 in Wroclaw 2024 and SG1 voted with unanimous consent to forward (D)P3427R1 to LEWG for C++26 with feedback:

```
Forward (D)P3427R1 to LEWG for C++26 with notes:
* The word 'asynchronous' doesn't belong in the name of the free function
```

R2

R2 revises R1 to follow the feedback from SG1 and for review by LEWG. It makes the following changes from R1:

- Change **`hazard_pointer_asynchronous_reclamation`** to **`hazard_pointer_try_reclamation`**.
- Added "May reclaim possibly-reclaimable members of **`c`**." to the wording of `reclaim to cohort`.

R3

R3 revises R2 in response to feedback from LEWG in Croydon 2026, by merging Proposed Interface with Draft Wording, adding an exposition-only container to hazard_pointer_cohort, and using the exposition-only member in wording. Otherwise, R3 is identical to R2. LEWG reviewed R2 and the updated R3 wording.
LEWG voted to approve the design as in R2 (same as R3) with feedback:

```
Approve the design of "P3427R2 Hazard Pointer Synchronous Reclamation" with the
discussed changes.
ACTIONs for P3427R3:
   ● Polish the wording (preferably with a review from a wording expert from LWG)
     based on what was presented in the meeting and publish an updated R4.
   ● Bump the feature test macro.
```

R4

R4 revises R3 and applies feedback from LEWG in Croydon 2026, by polishing wording and including feature test macro bump.

R5

R5 revises R4 with minor fixes and editorial improvements. LEWG reviewed R5 in Brno 2026 and voted for:

```
POLL: Restrict the implementation of hazard_pointer_cohort to be non allocating.

POLL: Forward “P3427R5: Hazard Pointer Synchronous Reclamation” (with the action items:
non allocating hazard_pointer_cohort) to LWG for C++29.

ACTIONS: rename the expo only type according to the conventional format, remove the
“//expositional interface”, apply the requirements from the poll.
```

R6

R6 revises R5 according to LEWG feedback by:

(1) Adding to class `hazard_pointer_cohort` wording:

> `hazard_pointer_cohort` operations do not allocate memory.

(2) Changing __hp_obj_base to *`hp-obj-base`* and removing "*// Expositional inheritance*"

R6 also makes a correction to R5 by removing "a pointer to" from "Each element of `members_` is **a pointer to** an *`hp-obj-base`* subobject of a retired object." to be "Each element of `members_` is an *`hp-obj-base`* subobject of a retired object."

R7

R7 revises R6 as follows:

- Update the baseline value of the feature test macro.
- Add missing hazard_pointer_try_reclamation declaration to synopsis.
- Add missing *Mandates*: `T` is a hazard-protectable type to `retire_to_cohort`.

LWG reviewed R7 on 2026-07-24 and provided feedback, applied in R8.

R8

R8 revises R7 based on LWG feedback as follows:

- Make Proposed Wording a top-level section.
- Typeset large additions with green highlight.
- Remove the note that cohort members may have been reclaimed before cohort destruction.
- Add to [saferecl.hp.general] p5: Any given retired object `x` shall be reclaimed at most once.
- Add remark that concurrent invocations of retire_to_cohort with the same hazard_pointer_cohort object do not introduce data races.
- Remove the exposition-only items. Define member of cohort. Remove paragraphs related to the removed items. Rephrase references to members_ to use the added definition.
- Add a precondition to ~hazard_pointer_cohort() that all members of the cohort are possibly-reclaimable, and remove the note stating otherwise.

Introduction

This paper proposes extending the C++26 hazard pointer interface to support synchronous reclamation.

This revision, P3427R8, revises R7 by following LWG 2026-07-24 feedback. Ready for LWG review.

Background: C++26 Hazard Pointers

```
template <class T, class D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(D d = D()) noexcept;
protected:
  hazard_pointer_obj_base() = default;
  hazard_pointer_obj_base(const hazard_pointer_obj_base&) = default;
  hazard_pointer_obj_base(hazard_pointer_obj_base&&) = default;
  hazard_pointer_obj_base& operator=(const hazard_pointer_obj_base&) = default;
  hazard_pointer_obj_base& operator=(hazard_pointer_obj_base&&) = default;
  ~hazard_pointer_obj_base() = default;
private:
  D deleter ; // exposition only
};

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 <class T> T* protect(const atomic<T*>& src) noexcept;
  template <class T> bool try_protect(T*& ptr, const atomic<T*>& src) noexcept;
  template <class T> void reset_protection(const T* ptr) noexcept;
  void reset_protection(nullptr_t = nullptr) noexcept;
  void swap(hazard_pointer&) noexcept;
};

hazard_pointer make_hazard_pointer();
void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

Motivation

This paper proposes supporting object cohorts due to the importance of synchronous reclamation for general purpose usability. For example, a concurrent hash map that uses hazard pointers is more generally usable if it allows arbitrary key and value types rather than only types without dependence on resources with independent lifetimes.

Implementation and Use Experience

Object cohorts have been part of the Folly open-source library (under the name **`hazptr_obj_cohort`**) and in heavy use in production since 2018. (See CppCon 2021 *Hazard Pointer Synchronous reclamation beyond Concurrency TS2* for details about the evolution of support for synchronous reclamation in Folly).

Synchronous Reclamation

The P2530R3 C++26 hazard pointer interface supports only asynchronous reclamation which does not guarantee the timing of the reclamation of protectable objects that are no longer protected. As a result, hazard pointer users must guarantee separately that the deleters of such objects do not depend on resources that may become subsequently unavailable.

Support for synchronous reclamation allows users to synchronously induce and wait for the reclamation of unprotected objects.

Global Cleanup

A straightforward albeit inefficient solution to this problem is global cleanup, which guarantees the completion of deleters of all unprotected retired objects. This involves synchronously checking all retired objects against all hazard pointers.

The main drawback of the global cleanup approach is its high overhead that makes it impractical to use. For example, it may be useful to include global cleanup in the destructor of a generic container library. However, the prohibitive overhead of global cleanup makes that impractical for many use cases of such a container.

Another drawback of the global cleanup approach is that it adds overhead to the hazard pointer implementation even when users never use this feature (e.g., would require synchronization on thread local private buffers of retired objects that would otherwise be unnecessary).

We do not recommend the standardization of the global cleanup approach due to these drawbacks and the lack of production experience of the necessity of such approach in contrast to the more efficient approach discussed in the following section.

Object Cohorts

An alternative approach is the use of object cohorts, which are sets of protectable objects. Object cohorts support synchronous reclamation by guaranteeing that

> **all the deleters of the object cohort members are completed before the completion of the cohorts destructor**.

The main advantage of the object cohort approach is its performance. While its guarantees are weaker and less flexible than global cleanup, its efficiency enables users to use it in performance sensitive cases where the cost of global cleanup may be impractical. It strikes a better balance between practicality and performance. Therefore, we recommend object cohorts for standardization.

Proposed Wording

Edit **17.3.2 [version.syn] p2** as follows:

`__cpp_lib_hazard_pointer` ~~`202606L`~~ `202XXXXL`
[*Editor's note*: The value will be set to the date of the meeting where this paper is approved for the working draft.]

Edit **32.11.3.1 [saferecl.hp.general] p4** as follows:

An object `x` of hazard-protectable type `T` is *retired* with a deleter of type `D` when either the member function `hazard_pointer_obj_base<T, D>::retire` or the member function `hazard_pointer_obj_base<T, D>::retire_to_cohort` is invoked on `x`. Any given object `x` shall be retired at most once.

Edit **32.11.3.1 [saferecl.hp.general] p5** as follows:

A retired object `x` is *reclaimed* by invoking its deleter with a pointer to x; the behavior is undefined if that invocation exits via an exception. Any given retired object `x` shall be reclaimed at most once.

Add the following to namespace `std` in **32.11.3.2 [hazard.pointer.syn]** before `hazard_pointer_obj_base`:

```
  // [saferecl.hp.cohort], class hazard_pointer_cohort
  class hazard_pointer_cohort;
```

Add the following to namespace `std` in **32.11.3.2 [hazard.pointer.syn]**:

```
void hazard_pointer_try_reclamation() noexcept;
```

Add **32.11.3.x [saferecl.hp.cohort]**:

```
class hazard_pointer_cohort {
public:
  hazard_pointer_cohort() noexcept = default;
  hazard_pointer_cohort(const hazard_pointer_cohort&) = delete;
  hazard_pointer_cohort(hazard_pointer_cohort&&) = delete;
  hazard_pointer_cohort& operator=(const hazard_pointer_cohort&) = delete;
  hazard_pointer_cohort& operator=(hazard_pointer_cohort&&) = delete;
  ~hazard_pointer_cohort();
};
```

A *hazard-cohort* is a set of hazard-protectable objects. An object of type `hazard_pointer_cohort` is a hazard-cohort.

An object `x` of hazard-protectable type `T` with a deleter of type `D` becomes a member of a hazard-cohort `c` when the member function `hazard_pointer_obj_base<T, D>::retire_to_cohort` is invoked on `x` with `c` as the `cohort` argument. A retired object `x` that is a member of a hazard-cohort `c` ceases to be a member of `c` when `x` is reclaimed.

`hazard_pointer_cohort` operations do not allocate memory.

`~hazard_pointer_cohort();`
Preconditions: All members of `*this` are possibly-reclaimable with respect to the evaluation of this destructor.
Effects: Reclaims all members of `*this`.
Synchronization: The completion of the deleter for each object that has ever been a member of `*this` synchronizes with the return from this destructor.

Edit **32.11.3.3** **[saferecl.hp.base]** as follows:

```
template <class T, class D = default_delete<T>>
class hazard_pointer_obj_base {
public:
  void retire(D d = D()) noexcept;
  void retire_to_cohort(hazard_pointer_cohort& c, D d = D()) noexcept;
```

Add the following in **32.11.3.3** **[saferecl.hp.base]** after `retire`:

`void retire_to_cohort(hazard_pointer_cohort& c, D d = D()) noexcept;`
Mandates: `T` is a hazard-protectable type.
Preconditions: `*this` is a base class subobject of an object `x` of type `T`. `x` is not retired. Move-assigning `d` to `deleter` does not exit via an exception.
Effects: Move-assigns `d` to `deleter`, thereby setting it as the deleter of `x`, then retires `x`. May reclaim possibly-reclaimable members of `c`.
Remarks: Concurrent invocations of `retire_to_cohort` with the same `hazard_pointer_cohort` object do not introduce data races.

Add the following to **32.11.3.4.4** **[saferecl.hp.holder.nonmem]**:

`void hazard_pointer_try_reclamation() noexcept;`
Effects: May reclaim possibly-reclaimable objects.

Usage Example

The following table shows two code snippets one using the C++26 hazard pointer interface and one using object cohorts, respectively. The latter supports synchronous reclamation.

C++26 (Asynchronous Reclamation Only)

```
/// Library
template <class T> struct Container {
  struct Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };

  void insert(T data) {
    Obj* obj = new Obj(data);
    /* Insert obj in container */
  }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    executor_.add([obj] {
      obj->retire();
    });

  }
};

/// User
struct A {
  // Deleter cannot depend on resources
  // with independent lifetime.
   ~A();
};

{
  Container<A> container;
  container.insert(a);
  container.erase(a);
}
// Obj containing 'a' may be not deleted
// yet.
```

Cohort-Based Synchronous Reclamation

```
/// Library
template <class T> struct Container {
  struct Obj : hazard_pointer_obj_base<Obj>
  { T data; /* etc */ };
  hazard_pointer_cohort cohort_;
  void insert(T data) {
    Obj* obj = new Obj(data);
    /* Insert obj in container */
  }
  void erase(Args args) {
    Obj* obj = find(args);
    /* Remove obj from container */
    obj->retire_to_cohort(cohort_);
    executor_.add([] {
      hazard_pointer_try_reclamation();
    });
  }
};

/// User
struct B {
  // Deleter may depend on resources
  // with independent lifetime.
  ~B() { use_resource_XYZ(); }
};
make_resource_XYZ();
{
  Container<B> container;
  container.insert(b);
  container.erase(b);
}
// Obj containing 'b' was deleted.
destroy_resource_XYZ();
```

In the above example:

- In the code snippet on the left side, the removed object is retired by submitting the retirement code to a dedicated thread pool (details not shown) to be executed asynchronously. Doing so avoids burdening the current thread with potentially performing amortized reclamation of tens of thousands of possibly unrelated retired objects, because calling **`retire`** may trigger amortized asynchronous reclamation.
- In contrast, in the code snippet on the right side, retirement to the cohort must be executed synchronously in order for the object to be included as intended in synchronous reclamation of the associated cohort. Therefore, the call to **`retire_to_cohort`** must be synchronous. But in order not to burden worker threads with amortized asynchronous reclamation, **`retire_to_cohort`** does not try to perform general asynchronous reclamation. Instead, if desired, the current thread may submit an asynchronous task to a dedicated thread pool to try to perform asynchronous reclamation (which attempts to reclaim unprotected retired objects, both cohort-associated and not).

Separating Cohort Object Retirement from Asynchronous Reclamation

Asynchronous Reclamation of Cohort Objects

If a cohort is long-lived, large numbers (e.g., billions) of objects may be retired to it. If such objects are not included in asynchronous reclamation, they would remain not reclaimed. Therefore, cohort objects need to be included in asynchronous reclamation.

Cohort Object Retirement Must Be Synchronous

In order for an object to be included in synchronous reclamation in association with a cohort, the object must be retired to the cohort. Therefore, cohort object retirement needs to be synchronous.

Why Doesn't `retire_to_cohort` Try to Invoke Asynchronous Reclamation Implicitly?

As indicated above cohort-object retirement must be synchronous. However, it is often desirable to invoke asynchronous reclamation asynchronously in order to avoid burdening the thread retiring the object (typically a worker thread) with reclaiming a large number (e.g., tens of thousands) of (possibly unrelated) retired objects.

In contrast, the retirement of non-cohort objects (using `retire`) by convention may implicitly invoke asynchronous reclamation. If desired, a user may retire a non-cohort object `obj` asynchronously to avoid inline asynchronous reclamation as follows (assuming an executor associated with a separate execution resource):

```
executor_.add([obj] { obj->retire(); });
```

However, the retirement of a cohort object needs to be synchronous, as mentioned above. Therefore, a free function hazard_pointer_try_reclamation() is added, so that users can write the cohort equivalent to the above non-cohort code snippet as follows:

```
obj->retire_to_cohort(cohort_);
executor_.add([] { hazard_pointer_try_reclamation(); });
```

Acknowledgments

The authors thank Olivier Giroux, Mark Hoemmen, Tomasz Kaminski, Jens Maurer, Tim Song, Jonathan Wakely, Andreas Weis, and other members of SG1, LEWG, and LWG for useful discussions and suggestions that helped improve components of this proposal.

References

- P2530R3: Hazard Pointers for C++26 (2023-03-02).
- P3135R1: Hazard Pointer Extensions (2024-04-12).
- Folly: Facebook Open-source Library.
- CppCon 2021: *Hazard Pointer Synchronous reclamation beyond Concurrency TS2*