Document Number: N4953
Date: 2023-05-15
Revises: N4895
Reply to: Michael Wong
Codeplay
fraggamuffin@gmail.com

Working Draft, Extensions to C++for Concurrency Version 2

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

Foreword [foreword]

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents) or the IEC list of patent declarations received (see http://patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

1 Scope [scope]

1 This document describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with concurrent execution. The algorithms described by this document are realizable across a broad class of computer architectures.

2 ISO/IEC 14882:2020 provide important context and specification for this document. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from ISO/IEC 14882:2020 use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

3 This document is non-normative. Some of the functionality described by this document may be considered for standardization in a future version of C++, but it is not currently part of any C++ standard. Some of the functionality in this document may never be standardized, and other functionality may be standardized in a substantially changed form.

4 The goal of this document is to build widespread existing practice for concurrency in the C++ standard algorithms library. It gives advice on extensions to those vendors who wish to provide them.

2 Normative references [refs]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2020, *Programming Languages — C++*

2 ISO/IEC 14882:2020 is herein called the C++ Standard. References to clauses within the C++ Standard are written as "C++20 §3.2". The library described in C++20 §16-32 is herein called the C++ Standard Library.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++20 §16) is included into this Technical Specification by reference.

3 Terms and definitions [defs]

1 No terms and definitions are listed in this document. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

(1.1) — IEC Electropedia: available at https://www.electropedia.org/

(1.2) — ISO Online browsing platform: available at https://www.iso.org/obp

4 General [general]

4.1 Implementation compliance [general.compliance]

1 Conformance requirements for this document are those defined in C++20 §4.1, as applied to a merged document consisting of C++20 amended by this document.

[*Note 1*: Conformance is defined in terms of the behavior of programs. — *end note*]

4.2 Namespaces and headers and modifications to standard classes [general.namespaces]

1 Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they are not declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:

(1.1) — modify an existing interface in the C++ Standard Library in-place,

(1.2) — are declared in a namespace whose name appends `::experimental::concurrency_v2` to a namespace defined in the C++ Standard Library, such as `std`, or

(1.3) — are declared in a subnamespace of a namespace described in the previous bullet, whose name is not the same as an existing subnamespace of namespace `std`.

2 Whenever an unqualified name is used in the specification of a declaration `D`, its meaning is established as-if by performing unqualified name lookup in the context of `D`.

[*Note 1*: Argument-dependent lookup is not performed. — *end note*]

Similarly, the meaning of a *qualified-id* is established as-if by performing qualified name lookup in the context of `D`.

[*Note 2*: Operators in expressions are not so constrained. — *end note*]

3 These are the headers described in this document (see Table 1)

Table 1: C++ library headers

`<experimental/rcu>`
`<experimental/hazard_pointer>`
`<experimental/bytewise_atomic_memcpy>`
`<experimental/asymmetric_fence>`
`<experimental/synchronized_value>`

4.3 Feature-testing recommendations (Informative) [general.feature.test]

1 An implementation that provides support for this document should define each feature test macro defined in Table 2 and Table 3 if no associated headers are indicated for that macro, and if associated headers are indicated for a macro, that macro is defined after inclusion of one of the corresponding headers specified in the table.

Table 2: Feature-test macros name

Title	Subclause	Macro name
`Hazard pointers`	5.2	`__cpp_lib_experimental_hazard_pointer`
`Read-copy update(RCU)`	5.3	`__cpp_lib_experimental_rcu`
`bytewise atomic memcpy`	6	`__cpp_lib_experimental_bytewise_atomic_memcpy`
`Asymmetric Fence`	7	`__cpp_lib_experimental_asymmetric_fence`
`Synchronized Value`	8	`__cpp_lib_experimental_synchronized_value`

Table 3: Feature-test macros header

Title	Value	Header
`Hazard pointers`	202108	`<experimental/hazard_pointer>`
`Read-copy update(RCU)`	202108	`<experimental/rcu>`
`bytewise atomic memcpy`	202108	`<experimental/bytewise_atomic_memcpy>`
`Asymmetric Fence`	202108	`<experimental/asymmetric_fence>`
`Synchronized Value`	202108	`<experimental/synchronized_value>`

4.4 Future plans (Informative) [general.plans]

1 This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.

2 The C++ committee intends to release a new version of this technical specification approximately every few years, containing the concurrency extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::concurrency_v3`, `std::experimental::concurrency_v4`, etc., with the most recent implemented version inlined into `std::experimental`.

3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::concurrency_v`*N* segment of its namespace and by removing the `experimental/` prefix from its header's path.

4.5 Acknowledgments [general.ack]

This work is the result of a collaboration of researchers in industry and academia. We wish to thank the original authors of this document, Michael Wong, Paul McKenney, and Maged Michael, and the editing review team of Jonathan Wakely, Daniel Krügler, and Bryan St. Amour. We also wish to thank people who made valuable contributions within and outside these groups, including Jens Maurer, and many others not named here who contributed to the discussion.

5 Safe reclamation [saferecl]

5.1 General [saferecl.general]

This clause adds safe-reclamation techniques, which are most frequently used to straightforwardly resolve access-deletion races.

5.2 Hazard pointers [saferecl.hp]

5.2.1 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*]

5.2.2 Header <experimental/hazard_pointer> synopsis [saferecl.hp.syn]

```
namespace std::experimental::inline concurrency_v2 {
  // 5.2.3, class hazard_pointer_domain
  class hazard_pointer_domain;

  // 5.2.4, Default hazard_pointer_domain
  hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

  // 5.2.5, Clean up
  void hazard_pointer_clean_up(hazard_pointer_domain& domain = hazard_pointer_default_domain())
    noexcept;

  // 5.2.6, class template hazard_pointer_obj_base
  template <typename T, typename D = default_delete<T>> class hazard_pointer_obj_base;

  // 5.2.7, class hazard_pointer
  class hazard_pointer;

  // 5.2.8, Construct non-empty hazard_pointer
  hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

  // 5.2.9, Hazard pointer swap
  void swap(hazard_pointer&, hazard_pointer&) noexcept;
}
```

5.2.3 Class hazard_pointer_domain [saferecl.hp.domain]

5.2.3.1 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();
};
```

5.2.3.2 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.

5.2.4 Default `hazard_pointer_domain` [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.

5.2.5 Clean up [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.

5.2.6 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()) 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 (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()) 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;
```

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

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

5.2.7.1 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;
};
```

5.2.7.2 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.

5.2.7.3 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.

5.2.7.4 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`.

5.2.7.5 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.

5.2.8 `make_hazard_pointer` [saferecl.hp.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`.

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

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

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

5.3 Read-copy update (RCU) [saferecl.rcu]

5.3.1 General [saferecl.rcu.general]

1 RCU is a synchronization mechanism that can be used for linked data structures that are frequently read, but seldom updated. RCU does not provide mutual exclusion, but instead allows the user to schedule specified actions such as deletion at some later time.

2 A class type `T` is *rcu-protectable* if it has exactly one public base class of type `rcu_obj_base<T,D>` for some `D` and no base classes of type `rcu_obj_base<X,Y>` for any other combination `X`, `Y`. An object is rcu-protectable if it is of rcu-protectable type.

3 An invocation of `unlock` U on an `rcu_domain dom` corresponds to an invocation of `lock` L on `dom` if L is sequenced before U and either

(3.1) — no other invocation of `lock` on `dom` is sequenced after L and before U or

(3.2) — every invocation of `unlock` U' on `dom` such that L is sequenced before U' and U' is sequenced before U corresponds to an invocation of `lock` L' on `dom` such that L is sequenced before L' and L' is sequenced before U'.

[*Note 1*: This pairs nested locks and unlocks on a given domain in each thread. — *end note*]

4 A *region of RCU protection* on a domain `dom` starts with a `lock` L on `dom` and ends with its corresponding `unlock` U.

5 Given a region of RCU protection R on a domain `dom` and given an evaluation E that scheduled another evaluation F in `dom`, if E does not strongly happen before the start of R, the end of R strongly happens before evaluating F.

6 The evaluation of a scheduled evaluation is potentially concurrent with any other such evaluation. Each scheduled evaluation is evaluated at most once.

5.3.2 Header `<experimental/rcu>` synopsis [saferecl.rcu.syn]

```
namespace std::experimental::inline concurrency_v2 {
  // 5.3.3, class template rcu_obj_base
  template<class T, class D = default_delete<T>>
    class rcu_obj_base;

  // 5.3.4, class rcu_domain
  class rcu_domain;

  // 5.3.5, rcu_default_domain
  rcu_domain& rcu_default_domain() noexcept;

  // 5.3.6, rcu_synchronize
  void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;

  // 5.3.7, rcu_barrier
  void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;

  // 5.3.8, rcu_retire
  template<class T, class D = default_delete<T>>
    void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
}
```

5.3.3 Class `rcu_obj_base` [saferecl.rcu.base]

Objects of type `T` to be protected by RCU inherit from a specialization of `rcu_obj_base<T,D>`.

```
template<class T, class D = default_delete<T>>
class rcu_obj_base {
public:
  void retire(D d = D(), rcu_domain& dom = rcu_default_domain()) noexcept;
protected:
  rcu_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 `rcu_obj_base` is undefined.

3 `D` shall meet the requirements for *Cpp17DefaultConstructible* and *Cpp17MoveAssignable*.

4 `T` may be an incomplete type.

5 If `D` is trivially copyable, all specializations of `rcu_obj_base<T,D>` are trivially copyable.

```
void retire(D d = D(), rcu_domain& dom = rcu_default_domain()) noexcept;
```

6 *Mandates*: `T` is an rcu-protectable type.

7 *Preconditions*: `*this` is a base class subobject of an object `x` of type `T`. The member function `rcu_-obj_base<T,D>::retire` was not invoked on `x` before. The assignment to *`deleter`* does not throw an exception. The expression *`deleter`*`(addressof(x))` has well-defined behavior and does not throw an exception.

8 *Effects*: Evaluates *`deleter`* `= std::move(d)` and schedules the evaluation of the expression `delet-er(addressof(x))` in the domain `dom`.

9 *Remarks*: It is implementation-defined whether or not scheduled evaluations in `dom` can be invoked by the `retire` function.

[*Note 1*: If such evaluations acquire resources held across any invocation of retire on `dom`, deadlock can occur. — *end note*]

5.3.4 Class `rcu_domain` [saferecl.rcu.domain]

This class meets the requirements of *Cpp17BasicLockable* C++20 §32.2.5.2 and provides regions of RCU protection.

[*Example 1*:

```
std::scoped_lock<rcu_domain> rlock(rcu_default_domain());
```

— *end example*]

```
class rcu_domain {
public:
  rcu_domain(const rcu_domain&) = delete;
  rcu_domain& operator=(const rcu_domain&) = delete;

  void lock() noexcept;
  void unlock() noexcept;
};
```

The functions `lock` and `unlock` establish (possibly nested) regions of RCU protection.

5.3.4.1 `rcu_domain::lock` [saferecl.rcu.domain.lock]

```
void lock() noexcept;
```

1 *Effects*: Opens a region of RCU protection.

2 *Remarks*: Calls to the function lock do not introduce a data race (C++20 §6.9.2.1) involving `*this`.

5.3.4.2 `rcu_domain::unlock` [saferecl.rcu.domain.unlock]

```
void unlock() noexcept;
```

1 *Preconditions*: A call to the function `lock` that opened an unclosed region of RCU protection is sequenced before the call to `unlock`.

2 *Effects*: Closes the unclosed region of RCU protection that was most recently opened.

3 *Remarks*: It is implementation-defined whether or not scheduled evaluations in `*this` can be invoked by the `unlock` function.

[*Note 1*: If such evaluations acquire resources held across any invocation of `unlock` on `*this`, deadlock can occur. — *end note*]

Calls to the function `unlock` do not introduce a data race involving `*this`.

[*Note 2*: Evaluation of scheduled evaluations can still cause a data race. — *end note*]

5.3.5 `rcu_default_domain` [saferecl.rcu.default.domain]

```
rcu_domain& rcu_default_domain() noexcept;
```

1 *Returns*: A reference to the default object of type `rcu_domain`. A reference to the same object is returned every time this function is called.

5.3.6 `rcu_synchronize` [saferecl.rcu.synchronize]

```
void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;
```

1 *Effects*: If the call to `rcu_synchronize` does not strongly happen before the lock opening an RCU protection region `R` on `dom`, blocks until the `unlock` closing `R` happens.

2 *Synchronization*: The `unlock` closing `R` strongly happens before the return from `rcu_synchronize`.

5.3.7 `rcu_barrier` [saferecl.rcu.barrier]

```
void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;
```

1 *Effects*: May evaluate any scheduled evaluations in `dom`. For any evaluation that happens before the call to `rcu_barrier` and that schedules an evaluation E in `dom`, blocks until E has been evaluated.

2 *Synchronization*: The evaluation of any such E strongly happens before the return from `rcu_barrier`.

5.3.8 Template `rcu_retire` [saferecl.rcu.retire]

```
template<class T, class D = default_delete<T>>
void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
```

1 *Mandates*: `is_move_constructible_v<D>` is true.

2 *Preconditions*: `D` meets the *Cpp17MoveConstructible* and *Cpp17Destructible* requirements. The expression `d1(p)`, where `d1` is defined below, is well-formed and its evaluation does not exit via an exception.

3 *Effects*: May allocate memory. It is unspecified whether the memory allocation is performed by invoking `operator new`. Initializes an object `d1` of type `D` from `std::move(d)`. Schedules the evaluation of `d1(p)` in the domain `dom`.

[*Note 1*: If `rcu_retire` exits via an exception, no evaluation is scheduled. — *end note*]

4 *Throws*: Any exception that would be caught by a handler of type `bad_alloc`. Any exception thrown by the initialization of `d1`.

5 *Remarks*: It is implementation-defined whether or not scheduled evaluations in dom can be invoked by the `rcu_retire` function.

[*Note 2*: If such evaluations acquire resources held across any invocation of `rcu_retire` on `dom`, deadlock can occur. — *end note*]

6 Bytewise Atomic Memcpy [byteatomicmemcpy]

6.1 General [byteatomicmemcpy.general]

This clause describes bytewise atomic memcpy access.

6.2 Header <bytewiseatomicmemcpy> synopsis [byteeatomicmemcpy.syn]

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

  void* atomic_load_per_byte_memcpy(void* dest, const void* source, size_t count, memory_order order);

  void* atomic_store_per_byte_memcpy(void* dest, const void* source, size_t count, memory_order order);


}
```

1 The `atomic_load_per_byte_memcpy()` and `atomic_store_per_byte_memcpy()` functions support concurrent programming idioms in which values may be read while being written, but the value is trusted only when it can be determined after the fact that a race did not occur.

[*Note 1*: So-called "seqlocks" are the canonical example of such an idiom. — *end note*]

6.3 atomic_load_per_byte_memcpy [byteatomicmemcpy.load]

1 The `atomic_load_per_byte_memcpy` / `atomic_store_per_byte_memcpy` functions behave as if the `source` and `dest` bytes respectively were individual atomic objects.

`void* atomic_load_per_byte_memcpy(void* dest, const void* source, size_t count, memory_order order);`

2 *Preconditions*:

`order` is `memory_order::acquire` or `memory_order::relaxed`. `(char*)dest + [0, count)` and `(const char*)source + [0, count)` are valid ranges that do not overlap.

3 *Effects*: Copies `count` consecutive bytes pointed to by `source` into consecutive bytes pointed to by `dest`. Each individual load operation from a source byte is atomic with memory order `order`. These individual loads are unsequenced with respect to each other. The function implicitly creates objects ([intro.object]) in the destination region of storage immediately prior to copying the sequence of bytes to the destination.

[*Note 1*: There is no requirement that the individual bytes be copied in order, or that the implementation must operate on individual bytes. — *end note*]

4 *Returns*: `dest`.

6.4 atomic_store_per_byte_memcpy [byteatomicmemcpy.store]

`void* atomic_store_per_byte_memcpy(void* dest, const void* source, size_t count, memory_order order);`

1 *Preconditions*: `order` is `memory_order::release` or `memory_order::relaxed`. `(char*)dest + [0, count)` and `(const char*)source + [0, count)` are valid ranges that do not overlap.

2 *Effects*: Copies `count` consecutive bytes pointed to by `source` into consecutive bytes pointed to by `dest`. Each individual store operation to a destination byte is atomic with memory order `order`. These individual stores are unsequenced with respect to each other. The function implicitly creates objects ([intro.object]) in the destination region of storage immediately prior to copying the sequence of bytes to the destination.

3 *Returns*:

`dest`.

[*Note 1*: If any of the atomic byte loads performed by an `atomic_load_per_byte_memcpy()` call A with `memory_order::acquire` argument take their value from an atomic byte store performed by `atomic_store_per_byte_memcpy()` call B with `memory_order::release` argument, then the start of B strongly happens before the completion of A. — *end note*]

7 Asymmetric Fence [asymmetric]

7.1 General [asymmetric.general]

This clause describes Asymmetric Fence access.

7.2 Header <experimental/asymmetric_fence> synopsis [asymmetric.syn]

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


  // 7.3 asymmetric_thread_fence_heavy
  void asymmetric_thread_fence_heavy(memory_order order) noexcept;
  // 7.4 asymmetric_thread_fence_light
  void asymmetric_thread_fence_light(memory_order order) noexcept;


}
```

1 This subclause introduces synchronization primitives called *heavyweight-fences* and *lightweight-fences*. Like fences, heavyweight-fences and lightweight-fences can have acquire semantics, release semantics, or both, and can be sequentially consistent (in which case they are included in the total order `S` on `memory_order::seq_cst` operations). A heavyweight-fence with acquire semantics is called an acquire heavyweight-fence. A heavyweight-fence has all the synchronization effects of a fence (C++20 §33.5.11 [atomics.fences]).

[*Note 1*: Heavyweight-fences and lightweight-fences are distinct from fences. — *end note*]

2 A heavyweight-fence with acquire semantics is called an *acquire heavyweight-fence*. A heavyweight-fence with release semantics is called a *release heavyweight-fence*. A lightweight-fence with acquire semantics is called an *acquire lightweight-fence*. A lightweight-fence with release semantics is called a *release lightweight-fence*.

3 If there are evaluations A and B, and atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation, and one of the following hold:

(3.1) — A is a release lightweight-fence and B is an acquire heavyweight-fence; or

(3.2) — A is a release heavyweight-fence and B is an acquire lightweight-fence

then any evaluation sequenced before A strongly happens before any evaluation that B is sequenced before.

7.3 asymmetric_thread_fence_heavy [asymmetric.heavy]

```
void asymmetric_thread_fence_heavy(memory_order order) noexcept;
```

1 *Effects*: Depending on the value of `order`, this operation:

(1.1) — has no effects, if `order == memory_order::relaxed`;

(1.2) — is an acquire heavyweight-fence, if `order == memory_order::acquire` or `order == memory_order::consume`;

(1.3) — is a release heavyweight-fence, if `order == memory_order::release`;

(1.4) — is both an acquire heavyweight-fence and a release heavyweight-fence, if `order == memory_order::acq_rel`;

(1.5) — is a sequentially consistent acquire and release heavyweight-fence, if `order == memory_order::seq_cst`.

7.4 asymmetric_thread_fence_light [asymmetric.light]

```
void asymmetric_thread_fence_light(memory_order order) noexcept;
```

1 *Effects*: Depending on the value of `order`, this operation:

(1.1) — has no effects, if `order == memory_order::relaxed`;

(1.2) — is an acquire lightweight-fence, if `order == memory_order::acquire` or `order == memory_order::consume`;

(1.3) — is a release lightweight-fence, if `order == memory_order::release`;

(1.4) — is both an acquire lightweight-fence and a release lightweight-fence, if `order == memory_order::acq_rel`;

(1.5) — is a sequentially consistent acquire and release lightweight-fence, if `order == memory_order::seq_cst`.

[*Note 1*: : Delegating both heavyweight-fence and lightweight-fence functions to an `atomic_thread_fence(order)` call is a valid implementation. Implementations can adopt techniques in which calls to `asymmetric_thread_fence_light` execute more quickly than calls to `atomic_thread_fence` with the same `memory_order`, at the cost of `asymmetric_thread_fence_heavy` executing more slowly than calls to `atomic_thread_fence` with the same `memory_order` — *end note*]

8 Synchronized Value [synchronizedvalue]

8.1 General [synchronizedvalue.general]

This section describes a class template to provide locked access to a value in order to facilitate the construction of race-free programs.

8.2 Header <experimental/synchronized_value> synopsis [synchronizedvalue.syn]

```
namespace std::experimental::inline concurrency_v2 {
    template<class T>
    class synchronized_value;

    template<class F,class ... ValueTypes>
    invoke_result_t<F, ValueTypes&...> apply(
        F&& f,synchronized_value<ValueTypes>&... values);
}
```

8.3 Class template `synchronized_value` [synchronizedvalue.class]

```
namespace std::experimental::inline concurrency_v2 {
  template<class T>
    class synchronized_value
    {
    public:
        synchronized_value(synchronized_value const&) = delete;
        synchronized_value& operator=(synchronized_value const&) = delete;

        template<class ... Args>
        synchronized_value(Args&& ... args);

    private:
        T value; // exposition only
        mutex mut; // exposition only
    };

template<class T>
synchronized_value(T)
-> synchronized_value<T>;
}
```

1 An object of type `synchronized_value<T>` wraps an object of type `T`. The wrapped object can be accessed by passing a callable object or function to `apply`. All such accesses are done with a lock held to ensure that only one thread may be accessing the wrapped object for a given `synchronized_value` at a time.

```
template<class ... Args>
synchronized_value(Args&& ... args);
```

2 *Constraints*:

(2.1) — `(sizeof...(Args) != 1)` is `true` or `(!same_as<synchronized_value,remove_cvref_t<Args>> &&...)` is `true`

(2.2) — `is_constructible_v<T,Args...>` is `true`

3 *Effects*: Direct-non-list-initializes *`value`* with `std::forward<Args>(args)...`.

4 *Throws*: Any exceptions emitted by the initialization of *`value`*.
`system_error` if any necessary resources cannot be acquired.

8.4 apply function [synchronizedvalue.fn]

```
template<class F,class ... ValueTypes>
invoke_result_t<F, ValueTypes&...> apply(
    F&& f,synchronized_value<ValueTypes>&... values);
```

1 *Constraints*: `sizeof...(values) != 0` is `true`.

2 *Effects*: Equivalent to:

```
scoped_lock lock(values.mut...);
return invoke(std::forward<F>(f),values.value...);
```

[*Note 1*: A single instance of `synchronized_value` can not be passed more than once to the same invocation of `apply`.

[*Example 1*:

```
synchronized_value<int> sv;
void f(int,int);
apply(f,sv,sv); // undefined behaviour, sv passed more than once to same call
```

— *end example*]

— *end note*]

[*Note 2*: The invocation of `f` can not call `apply` directly or indirectly passing any of `values...`. — *end note*]

33 Concurrency support library [thread]

33.5 Atomic operations [atomics]

33.5.4 Order and consistency [atomics.order]

Change in C++20 §33.5.4 paragraph 4 as indicated:

4 There is a single total order S on all `memory_order::seq_cst` operations, including fences, that satisfies the following constraints. First, if A and B are `memory_order::seq_cst` operations and A strongly happens before B, then A precedes B in S. Second, for every pair of atomic operations A and B on an object M, where A is coherence-ordered before B, the following ~~four~~ conditions are required to be satisfied by S:

Add the following two bullets to the list:

(4.1) — ...

(4.2) — if a `memory_order::seq_cst` lightweight-fence X happens before A and B happens before a `memory_order::seq_cst` heavyweight-fence Y, then X precedes Y in S; and

(4.3) — if a `memory_order::seq_cst` heavyweight-fence X happens before A and B happens before a `memory_order::seq_cst` lightweight-fence Y, then X precedes Y in S.