Technical Specification

ISO/IEC TS 9922

Programming Languages — Technical specification for C++ extensions for concurrency 2

Langages de programmation — Spécification technique pour les extensions C++ de concurrency 2

First edition
2024-11

Reference number
ISO/IEC TS 9922:2024(en)

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 o f t echnical a c tivity. I SO a nd IEC technical committees collaborate in fields o f m utual i n terest. O ther i nternational o rganizations, 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 a p proval c r iteria n e eded f o r t he 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 or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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

For an explanation of the voluntary nature of standards, the meaning of ISO specific t erms a nd 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. In the IEC, see www.iec.ch/understanding-standards.

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 and www.iec.ch/national-committees.

1 Scope [scope]

This document builds upon ISO/IEC 14882 by describing requirements for implementations of an interface that computer programs written in the C++ programming language could use to invoke algorithms with concurrent execution. The algorithms described by this document are realizable across a broad class of computer architectures. This document is written as a set of differences from the base standard.

Some of the functionality described by this document might be considered for standardization in a future version of C++, but it is not currently part of ISO/IEC 14882:2020. Some of the functionality in this document might never be standardized, and other functionality might be standardized in a substantially different form.

The goal of this document is to build widespread existing practice for concurrency in the ISO/IEC 14882:2020 algorithms library. It gives advice on extensions to those vendors who wish to provide them.

2 Normative references [refs]

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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.

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

3 Terms and definitions [defs]

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

— IEC Electropedia: available at www.electropedia.org

— ISO Online browsing platform: available at www.iso.org/obp

4 General [general]

4.1 Implementation compliance [general.compliance]

Conformance requirements for this document are those defined in 4.1, as applied to a merged document consisting of ISO/IEC 14882:2020 amended by this document.

NOTE Conformance is defined in terms of the behaviour of programs.

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

Since the extensions described in this document are experimental and not part of the ISO/IEC 14882:2020 library, they are not declared directly within namespace `std`. Unless otherwise specified, all components described in this document either:

— modify an existing interface in the ISO/IEC 14882:2020 library in-place,

— are declared in a namespace whose name appends `::experimental::concurrency_v2` to a namespace defined in the ISO/IEC 14882:2020 library, such as `std`, or

— 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`.

Whenever an unqualified name is used in the specification of a declaration `D`, its meaning is established in accordance with 4.1.2 by performing unqualified name lookup in the context of `D`.

NOTE 1 Argument-dependent lookup is not performed.

Similarly, the meaning of a qualified-id is established in accordance with performing qualified name lookup in the context of `D`.

NOTE 2 Operators in expressions are not so constrained.

Table 1 shows the headers described in this document

Table 1 — C++ library headers

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

4.3 Feature-testing recommendations [general.feature.test]

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 Table 2 and Table 3.

Table 2 — Feature-test macros name

Title	Subclause	Macro name
Synchronized Value	5	`__cpp_lib_experimental_synchronized_value`
Hazard pointers	6.2	`__cpp_lib_experimental_hazard_pointer`
Read-copy update(RCU)	6.3	`__cpp_lib_experimental_rcu`
Bytewise atomic memcpy	7	`__cpp_lib_experimental_bytewise_atomic_memcpy`
Asymmetric Fence	8,33	`__cpp_lib_experimental_asymmetric_fence`

Table 3 — Feature-test macros header

Title	Value	Header
`Synchronized Value`	202406	`<experimental/synchronized_value>`
`Hazard pointers`	202406	`<experimental/hazard_pointer>`
`Read-copy update(RCU)`	202406	`<experimental/rcu>`
`Bytewise atomic memcpy`	202406	`<experimental/bytewise_atomic_memcpy>`
`Asymmetric Fence`	202406	`<experimental/asymmetric_fence>`

5 Synchronized Value [synchronizedvalue]

5.1 General [synchronizedvalue.general]

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

5.2 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);
}
```

5.3 Class template synchronized_value [synchronizedvalue.class]

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

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);
```

Constraints:

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

— `is_constructible_v<T,Args...>` is `true`

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

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

5.4 apply function [synchronizedvalue.fn]

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

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

Effects: Equivalent to:

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

NOTE 1 It is not possible to pass a single instance of `synchronized_value` more than once to the same invocation of `apply`.

EXAMPLE

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

NOTE 2 The invocation of `f` cannot call `apply` directly or indirectly passing any of `values...`.

6 Safe reclamation [saferecl]

6.1 General [saferecl.general]

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

6.2 Hazard pointers [saferecl.hp]

6.2.1 General [saferecl.hp.general]

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.

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.

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.

A hazard pointer belongs to exactly one domain.

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.

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.

A retired object `x` is reclaimed by invoking its deleter with a pointer to `x`.

A hazard-protectable object `x` is definitely reclaimable in a domain dom with respect to an evaluation A if:

(a) `x` is not reclaimed, and

(b) `x` is retired to dom in an evaluation that happens before A, and

(c) for all hazard pointers h that belong to dom, the end of any protection epoch where h is associated with `x` happens before A.

A hazard-protectable object `x` is possibly reclaimable in domain dom with respect to an evaluation A if:

(d) `x` is not reclaimed; and

(e) `x` is retired to dom in an evaluation R and A does not happen before R; and

(f) 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

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

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

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

(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.

NOTE 3 Condition d(2) and d(3) 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`.

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

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

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

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

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

```
namespace std::experimental::inline concurrency_v2 {
  // 6.2.3, class hazard_pointer_domain
  class hazard_pointer_domain;

  // 6.2.4, Default hazard_pointer_domain
  hazard_pointer_domain& hazard_pointer_default_domain() noexcept;

  // 6.2.5, Clean up
  void hazard_pointer_clean_up(hazard_pointer_domain& domain = hazard_pointer_default_domain())
    noexcept;

  // 6.2.6, class template hazard_pointer_obj_base
  template <typename T, typename D = default_delete<T>> class hazard_pointer_obj_base;

  // 6.2.7, class hazard_pointer
  class hazard_pointer;

  // 6.2.8, Construct non-empty hazard_pointer
  hazard_pointer make_hazard_pointer(
    hazard_pointer_domain& domain = hazard_pointer_default_domain());

  // 6.2.9, Hazard pointer swap
  void swap(hazard_pointer&, hazard_pointer&) noexcept;
}
```

6.2.3 Class `hazard_pointer_domain` [saferecl.hp.domain]

6.2.3.1 General [saferecl.hp.domain.general]

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

NOTE The bound can be independent of other domains and can be a function of the number of hazard pointers belonging to the domain, the number of threads that retire objects to the domain, and the number of threads that use hazard pointers belonging to the domain.

Concurrent access to a domain does not incur a data race.

```
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();
};
```

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

`hazard_pointer_domain() noexcept;`

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

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

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

`~hazard_pointer_domain();`

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

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

6.2.4 Default `hazard_pointer_domain` [saferecl.hp.domain.default]

`hazard_pointer_domain& hazard_pointer_default_domain() noexcept;`

Returns: A reference to the default `hazard_pointer_domain`.

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.

6.2.5 Clean up [saferecl.hp.cleanup]

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

Effects: May reclaim possibly-reclaimable objects retired to `domain`.

Postconditions: All definitely-reclaimable objects retired to `domain` have been reclaimed.

Synchronization: The completion of the deleter for each reclaimed object synchronizes with the return from this function call.

6.2.6 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
};
```

A client-supplied template argument `D` shall be a function object type 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.

The behaviour of a program that adds specializations for `hazard_pointer_obj_base` is undefined.

D shall meet the requirements for Cpp17DefaultConstructible and Cpp17MoveAssignable.

T may be an incomplete type.

```
void retire(D d = D(), hazard_pointer_domain& domain = hazard_pointer_default_domain()) 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 from d does not throw an exception. The expression d(addressof(x)) has well-defined behaviour and does not throw an exception.

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

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

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

Effects: Equivalent to retire(D(), domain).

6.2.7 Class hazard_pointer [saferecl.hp.holder]

6.2.7.1 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;
  };
```

6.2.7.2 Constructors [saferecl.hp.holder.ctor]

```
hazard_pointer() noexcept;
```

Postconditions: *this is empty.

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

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

6.2.7.3 Destructor [saferecl.hp.holder.dtor]

```
~hazard_pointer();
```

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

6.2.7.4 Assignment [saferecl.hp.holder.assign]

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

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.

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.

Returns: *this.

6.2.7.5 Member functions [saferecl.hp.holder.mem]

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

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

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

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;
```

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

Preconditions: `*this` is not empty.

Effects:

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

(b) Evaluates the function call `reset_protection(old)`.

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

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

Returns: `old == ptr`.

NOTE 1 It is possible for `try_protect` to return `true` when `ptr` is a null pointer.

Complexity: Constant.

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

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

Preconditions: `*this` is not empty.

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;
```

Preconditions: `*this` is not empty.

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

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

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.

Complexity: Constant.

6.2.8 `make_hazard_pointer` [saferecl.hp.make]

```
hazard_pointer make_hazard_pointer(
  hazard_pointer_domain& domain = hazard_pointer_default_domain());
```

Effects: Constructs a hazard pointer belonging to `domain`.

Returns: A `hazard_pointer` object that owns the newly-constructed hazard pointer.

Throws: Any exception thrown by the allocator of `domain`.

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

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

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

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

6.3.1 General [saferecl.rcu.general]

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.

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.

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

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

— 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 This associates corresponding locks and unlocks within a nested set on a given domain in each thread.

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

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.

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

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

```
namespace std::experimental::inline concurrency_v2 {
  // 6.3.3, class template rcu_obj_base
  template<class T, class D = default_delete<T>>
    class rcu_obj_base;

  // 6.3.4, class rcu_domain
  class rcu_domain;

  // 6.3.5, rcu_default_domain
  rcu_domain& rcu_default_domain() noexcept;

  // 6.3.6, rcu_synchronize
  void rcu_synchronize(rcu_domain& dom = rcu_default_domain()) noexcept;

  // 6.3.7, rcu_barrier
  void rcu_barrier(rcu_domain& dom = rcu_default_domain()) noexcept;

  // 6.3.8, rcu_retire
  template<class T, class D = default_delete<T>>
    void rcu_retire(T* p, D d = D(), rcu_domain& dom = rcu_default_domain());
}
```

6.3.3 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
};
```

A client-supplied template argument `D` shall be a function object type in accordance with ISO/IEC 14882:2020, 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.

The behaviour of a program that adds specializations for `rcu_obj_base` is undefined.

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

`T` may be an incomplete type.

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;
```

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

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 behaviour and does not throw an exception.

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

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

NOTE If such evaluations acquire resources held across any invocation of retire on `dom`, deadlock can occur.

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

6.3.4.1 General [saferecl.rcu.domain.general]

This class meets the requirements of Cpp17BasicLockable in accordance with ISO/IEC 14882:2020, 32.2.5.2 and provides regions of RCU protection.

```
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.

EXAMPLE

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

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

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

Effects: Opens a region of RCU protection.

Remarks: Calls to the function lock do not introduce a data race (in accordance with ISO/IEC 14882:2020, 6.9.2.1) involving `*this`.

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

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

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

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

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.

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.

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

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

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.

6.3.6 `rcu_synchronize` [saferecl.rcu.synchronize]

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

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.

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

6.3.7 `rcu_barrier` [saferecl.rcu.barrier]

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

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.

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

6.3.8 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());
```

Mandates: `is_move_constructible_v<D>` is true.

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.

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.

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

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.