Document Number: N4501
Date: 2015-05-08
Editor: Artur Laksberg
Microsoft Corp.
arturl@microsoft.com

Working Draft, Technical Specification for C++ Extensions for Concurrency

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

Contents

1 General [general]

1.1 Namespaces, 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 should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:

> **Editor's note:** This section reflects the consensus between the LWG and LEWG at the Chicago 2013 and Issaquah 2014 meetings.

- modify an existing interface in the C++ Standard Library in-place,
- are declared in a namespace whose name appends `::experimental::concurrency_v1` to a namespace defined in the C++ Standard 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`.

2 Each header described in this technical specification shall import the contents of `std::experimental::concurrency_v1` into `std::experimental` as if by

```
namespace std {
  namespace experimental {
    inline namespace concurrency_v1 {}
  }
}
```

3 Unless otherwise specified, references to other entities described in this technical specification are assumed to be qualified with `std::experimental::concurrency_v1::`, and references to entities described in the standard are assumed to be qualified with `std::`.

4 Extensions that are expected to eventually be added to an existing header `<meow>` are provided inside the `<experimental/meow>` header, which shall include the standard contents of `<meow>` as if by

```
#include <meow>
```

5 New headers are also provided in the `<experimental/>` directory, but without such an `#include`.

Table 1 — C++ library headers

`<experimental/future>`	`<experimental/barrier>`
`<experimental/latch>`	`<experimental/atomic>`

1.2 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 year, containing the library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::concurrency_v2`, `std::experimental::concurrency_v3`, 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_vN` segment of its namespace and by removing the `experimental/` prefix from its header's path.

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

1 For the sake of improved portability between partial implementations of various C++ standards, WG21 (the ISO technical committee for the C++ programming language) recommends that implementers and programmers follow the guidelines in this section concerning feature-test macros. [*Note:* WG21's SD-6 makes similar recommendations for the C++ Standard itself. — *end note*]

2 Implementers who provide a new standard feature should define a macro with the recommended name, in the same circumstances under which the feature is available (for example, taking into account relevant command-line options), to indicate the presence of support for that feature. Implementers should define that macro with the value specified in the most recent version of this technical specification that they have implemented. The recommended macro name is "`__cpp_lib_experimental_`" followed by the string in the "Macro Name Suffix" column.

3 Programmers who wish to determine whether a feature is available in an implementation should base that determination on the presence of the header (determined with `__has_include(<header/name>)`) and the state of the macro with the recommended name. (The absence of a tested feature may result in a program with decreased functionality, or the relevant functionality may be provided in a different way. A program that strictly depends on support for a feature can just try to use the feature unconditionally; presumably, on an implementation lacking necessary support, translation will fail.)

Table 2 — Significant features in this technical specification

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
N4399	Improvements to std::future<T> and Related APIs	2	`future_continuations`	201505	`<experimental/future>`
N4204	C++ Latches and Barriers	3	`latch`	201505	`<experimental/latch>`
N4204	C++ Latches and Barriers	3	`barrier`	201505	`<experimental/barrier>`
N4260	Atomic Smart Pointers	4	`atomic_smart_pointers`	201505	`<experimental/atomics>`

2 Improvements to `std::future<T>` and Related APIs [futures]

2.1 General [futures.general]

1 The extensions proposed here are an evolution of the functionality of `std::future` and `std::shared_future`. The extensions enable wait-free composition of asynchronous operations. Class templates `std::promise` and `std::packaged_task` are also updated to be compatible with the updated `std::future`.

2.2 Header <experimental/future> synopsis [header.future.synop]

> **Editor's note:** An additional editorial fix as in Fundamental v1 TS is applied in the declaration of `swap` for `packaged_task`

```
#include <future>

namespace std {
  namespace experimental {
  inline namespace concurrency_v1 {

    template <class R> class promise;
    template <class R> class promise<R&>;
    template <> class promise<void>;

    template <class R>
      void swap(promise<R>& x, promise<R>& y) noexcept;

    template <class R> class future;
    template <class R> class future<R&>;
    template <> class future<void>;
    template <class R> class shared_future;
    template <class R> class shared_future<R&>;
    template <> class shared_future<void>;

    template <class> class packaged_task; // undefined
    template <class R, class... ArgTypes>
      class packaged_task<R(ArgTypes...)>;

    template <class R, class... ArgTypes>
      void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

    template <class T>
      see below make_ready_future(T&& value);
    future<void> make_ready_future();

    template <class T>
      future<T> make_exceptional_future(exception_ptr ex);
    template <class T, class E>
      future<T> make_exceptional_future(E ex);

    template <class InputIterator>
      see below when_all(InputIterator first, InputIterator last);
    template <class... Futures>
```



```
      see below when_all(Futures&&... futures);

    template <class Sequence>
    struct when_any_result;

    template <class InputIterator>
      see below when_any(InputIterator first, InputIterator last);
    template <class... Futures>
      see below when_any(Futures&&... futures);

  } // namespace concurrency_v1
  } // namespace experimental

  template <class R, class Alloc>
    struct uses_allocator<experimental::promise<R>, Alloc>;

  template <class R, class Alloc>
    struct uses_allocator<experimental::packaged_task<R>, Alloc>;

} // namespace std
```

2.3 Class template `future` [futures.unique_future]

1 The specifications of all declarations within this subclause 2.3 and its subclauses are the same as the corresponding declarations, as specified in C++14 §30.6.6, unless explicitly specified otherwise.

```
namespace std {
  namespace experimental {
  inline namespace concurrency_v1 {

    template <class R>
    class future {
    public:
      future() noexcept;
      future(future &&) noexcept;
      future(const future&) = delete;
      future(future<future<R>>&&) noexcept;
      ~future();
      future& operator=(const future&) = delete;
      future& operator=(future&&) noexcept;
      shared_future<R> share();

      // retrieving the value
      see below get();

      // functions to check state
      bool valid() const noexcept;
      bool is_ready() const;

      void wait() const;
      template <class Rep, class Period>
        future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```



```
    template <class Clock, class Duration>
      future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;

    // continuations
    template <class F>
      see below then(F&& func);

  };

} // namespace concurrency_v1
} // namespace experimental
} // namespace std
```

2 `future(future<future<R>>&& rhs) noexcept;`

3 *Effects:* Constructs a `future` object from the shared state referred to by `rhs`. The `future` becomes ready when one of the following occurs:

- Both the `rhs` and `rhs.get()` are ready. The value or the exception from `rhs.get()` is stored in the `future`'s shared state.
- `rhs` is ready but `rhs.get()` is invalid. An exception of type `std::future_error`, with an error condition of `std::future_errc::broken_promise` is stored in the `future`'s shared state.

4 *Postconditions:*

- `valid() == true`.
- `rhs.valid() == false`.

5 The member function template `then` provides a mechanism for attaching a *continuation* to a `future` object, which will be executed as specified below.

6
```
template <class F>
  see below then(F&& func);
```

7 *Requires:* `INVOKE(DECAY_COPY (std::forward<F>(func)), std::move(*this))` shall be a valid expression.

8 *Effects:* The function creates a shared state that is associated with the returned `future` object. Additionally,

- — When the object's shared state is ready, the continuation `INVOKE(DECAY_COPY(std::forward<F>(func)), std::move(*this))` is called on an unspecified thread of execution with the call to `DECAY_COPY()` being evaluated in the thread that called `then`.
- — Any value returned from the continuation is stored as the result in the shared state of the resulting `future`. Any exception propagated from the execution of the continuation is stored as the exceptional result in the shared state of the resulting `future`.

9 *Returns:* When `result_of_t<decay_t<F>(future<R>)>` is `future<R2>`, for some type `R2`, the function returns `future<R2>`. Otherwise, the function returns `future<result_of_t<decay_t<F>(future<R>)>>`. [*Note:* The rule above is referred to as *implicit unwrapping*. Without this rule, the return type of `then` taking a callable returning a `future<R>` would have been `future<future<R>>`. This rule avoids such nested `future` objects. The type of `f2` below is `future<int>` and not `future<future<int>>`:
[*Example:*

```
future<int> f1 = g();
future<int> f2 = f1.then([](future<int> f) {
                    future<int> f3 = h();
                    return f3;
                 });
```

— *end example*]
— *end note*]

10 *Postconditions:* `valid() == false` on the original `future`. `valid() == true` on the `future` returned from `then`. [*Note:* In case of implicit unwrapping, the validity of the `future` returned from `then` cannot be established until after the completion of the continuation. If it is not valid, the resulting `future` becomes ready with an exception of type `std::future_error`, with an error condition of `std::future_errc::broken_promise`. — *end note*]

11
```
bool is_ready() const;
```

12 *Returns:* `true` if the shared state is ready, otherwise `false`.

2.4 Class template `shared_future` [futures.shared_future]

1 The specifications of all declarations within this subclause 2.4 and its subclauses are the same as the corresponding declarations, as specified in C++14 §30.6.7, unless explicitly specified otherwise.

```
namespace std {
namespace experimental {
inline namespace concurrency_v1 {

  template <class R>
  class shared_future {
  public:
    shared_future() noexcept;
    shared_future(const shared_future&) noexcept;
    shared_future(future<R>&&) noexcept;
    shared_future(future<shared_future<R>>&& rhs) noexcept;
    ~shared_future();
```

```
    shared_future& operator=(const shared_future&);
    shared_future& operator=(shared_future&&) noexcept;

    // retrieving the value
    see below get();

    // functions to check state
    bool valid() const noexcept;
    bool is_ready() const;

    void wait() const;
    template <class Rep, class Period>
      future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
      future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) cons

    // continuations
    template <class F>
      see below then(F&& func) const;
  };

} // namespace concurrency_v1
} // namespace experimental
} // namespace std
```

2 `shared_future(future<shared_future<R>>&& rhs) noexcept;`

3 *Effects:* Constructs a `shared_future` object from the shared state referred to by `rhs`. The `shared_future` becomes ready when one of the following occurs:

- Both the `rhs` and `rhs.get()` are ready. The value or the exception from `rhs.get()` is stored in the `shared_future`'s shared state.
- `rhs` is ready but `rhs.get()` is invalid. The `shared_future` stores an exception of type `std::future_error`, with an error condition of `std::future_errc::broken_promise`.

4 *Postconditions:*

- `valid() == true`.
- `rhs.valid() == false`.

5 The member function template `then` provides a mechanism for attaching a *continuation* to a `shared_future` object, which will be executed as specified below.

6 `template <class F>`
`see below then(F&& func) const;`

7 *Requires:* `INVOKE(DECAY_COPY (std::forward<F>(func)), *this)` shall be a valid expression.

8 *Effects:* The function creates a shared state that is associated with the returned `future` object. Additionally,

- When the object's shared state is ready, the continuation `INVOKE(DECAY_COPY(std::forward<F>(func)), *this)` is called on an unspecified thread of execution with the call to `DECAY_COPY()` being evaluated in the thread that called `then`.
- Any value returned from the continuation is stored as the result in the shared state of the resulting `future`. Any exception propagated from the execution of the continuation is stored as the exceptional result in the shared state of the resulting `future`.

9 *Returns:* When `result_of_t<decay_t<F>(const shared_future&)>` is `future<R2>`, for some type `R2`, the function returns `future<R2>`. Otherwise, the function returns `future<result_of_t<decay_t<F>(const shared_future&)>>`. [*Note:* This analogous to `future`. See the notes on the return type of `future::then` in 2.3. — *end note*]

10 *Postconditions:* `valid() == true` on the original `shared_future` object. `valid() == true` on the `future` returned from `then`. [*Note:* In case of implicit unwrapping, the validity of the `future` returned from `then` cannot be established until after the completion of the continuation. In such case, the resulting `future` becomes ready with an exception of type `std::future_error`, with an error condition of `std::future_errc::broken_promise`. — *end note*]

11 `bool is_ready() const;`

12 *Returns:* `true` if the shared state is ready, otherwise `false`.

2.5 Class template `promise` [futures.promise]

1 The specifications of all declarations within this subclause 2.5 and its subclauses are the same as the corresponding declarations, as specified in C++14 §30.6.5, unless explicitly specified otherwise.

2 The `future` returned by the function `get_future` is the one defined in the `experimental` namespace (2.3).

2.6 Class template `packaged_task` [futures.task]

1 The specifications of all declarations within this subclause 2.6 and its subclauses are the same as the corresponding declarations, as specified in C++14 §30.6.9, unless explicitly specified otherwise.

2 The `future` returned by the function `get_future` is the one defined in the `experimental` namespace (2.3).

2.7 Function template `when_all` [futures.when_all]

1 The function template `when_all` creates a `future` object that becomes ready when all elements in a set of `future` and `shared_future` objects become ready.

2
```
template <class InputIterator>
  future<vector<typename iterator_traits<InputIterator>::value_type>>
  when_all(InputIterator first, InputIterator last);

  template <class... Futures>
  future<tuple<decay_t<Futures>...>> when_all(Futures&&... futures);
```

3 *Requires:* All `future`s and `shared_future`s passed into `when_all` must be in a valid state (i.e. `valid() == true`).

4 *Remarks:*

- The first overload shall not participate in overload resolution unless `iterator_traits<InputIterator>::value_type` is `future<R>` or `shared_future<R>` for some type `R`.
- For the second overload, let D_i be `decay_t<F_i>`, and let U_i be `remove_reference_t<F_i>` for each `F_i` in `Futures`. This function shall not participate in overload resolution unless for each *i* either D_i is a `shared_future<`R_i`>` or U_i is a `future<`R_i`>`.

5 *Effects:*

- A new shared state containing a `Sequence` is created, where `Sequence` is either `vector` or `tuple` based on the overload, as specified above. A new `future` object that refers to that shared state is created and returned from `when_all`.
- If the first overload is called with `first == last`, `when_all` returns a `future` with an empty `vector` that is immediately ready.
- If the second overload is called with no arguments, `when_all` returns a `future<tuple<>>` that is immediately ready.
- Otherwise, any `future`s are moved, and any `shared_future`s are copied into, correspondingly, `future`s or `shared_future`s of `Sequence` in the shared state.
- The order of the objects in the shared state matches the order of the arguments supplied to `when_all`.
- Once all the `future`s and `shared_future`s supplied to the call to `when_all` are ready, the resulting `future`, as well as the `future`s and `shared_future`s of the `Sequence`, are ready.
-
- The shared state of the `future` returned by `when_all` will not store an exception, but the shared states of `future`s and `shared_future`s held in the shared state may.

6 *Postconditions:*

- For the returned `future`, `valid() == true`.
- For all input `future`s, `valid() == false`.
- For all input `shared_future`s, `valid() == true`.

7 *Returns:* A `future` object that becomes ready when all of the input `future`sand `shared_future`s are ready.

2.8 Class template `when_any_result` [futures.when_any_result]

1 The library provides a template for storing the result of `when_any`.

```
template<class Sequence>
struct when_any_result {
    size_t index;
    Sequence futures;
};
```

2.9 Function template `when_any` [futures.when_any]

1 The function template `when_any` creates a `future` object that becomes ready when at least one element in a set of `future` and `shared_future` objects becomes ready.

2
```
template <class InputIterator>
    future<when_any_result<vector<typename iterator_traits<InputIterator>::value_type>>>
    when_any(InputIterator first, InputIterator last);

    template <class... Futures>
    future<when_any_result<tuple<decay_t<Futures>...>>> when_any(Futures&&... futures);
```

3 *Requires:* All `future`s and `shared_future`s passed into `when_all` must be in a valid state (i.e. `valid() == true`).

4 *Remarks:*

- — The first overload shall not participate in overload resolution unless `iterator_traits<InputIterator>::value_type` is `future<R>` or `shared_future<R>` for some type `R`.
- — For the second overload, let D_i be `decay_t<`F_i`>`, and let U_i be `remove_reference_t<`F_i`>` for each F_i in `Futures`. This function shall not participate in overload resolution unless for each *i* either D_i is a `shared_future<`R_i`>` or U_i is a `future<`R_i`>`.

5 *Effects:*

- — A new shared state containing `when_any_result<Sequence>` is created, where `Sequence` is a `vector` for the first overload and a `tuple` for the second overload. A new `future` object that refers to that shared state is created and returned from `when_any`.
- — If the first overload is called with `first == last`, `when_any` returns a `future` that is immediately ready. The value of the `index` field of the `when_any_result` is `static_cast<size_t>(-1)`. The `futures` field is an empty `vector`.
- — If the second overload of is called with no arguments, `when_any` returns a `future` that is immediately ready. The value of the `index` field of the `when_any_result` is `static_cast<size_t>(-1)`. The `futures` field is `tuple<>`.
- — Otherwise, any `future`s are moved, and any `shared_future`s are copied into, correspondingly, `future`s or `shared_future`s of the `futures` member of `when_any_result<Sequence>` in the shared state.
- — The order of the objects in the `futures` shared state matches the order of the arguments supplied to `when_any`.
- — Once at least one of the `future`s or `shared_future`s supplied to the call to `when_any` is ready, the resulting `future` is ready. Given the result future `f`, `f.get().index` is the position of the ready `future` or `shared_future` in the `futures` member of `when_any_result<Sequence>` in the shared state.
- — The shared state of the `future` returned by `when_all` will not store an exception, but the shared states of `future`s and `shared_future`s held in the shared state may.

6 *Postconditions:*

- — For the returned `future`, `valid() == true`.
- — For all input `future`s, `valid() == false`.
- — For all input `shared_future`s, `valid() == true`.

7 *Returns:*

- — A `future` object that becomes ready when any of the input `future`s and `shared_future`s are ready.

2.10 Function template `make_ready_future` [futures.make_ready_future]

1
```
template <class T>
  future<V> make_ready_future(T&& value);

  future<void> make_ready_future();
```

2 Let `U` be `decay_t<T>`. Then `V` is `X&` if `U` equals `reference_wrapper<X>`, otherwise `V` is `U`.

3 *Effects:* The function creates a shared state that is immediately ready and returns a `future` associated with that shared state. For the first overload, the type of the shared state is `V` and the result is constructed from `std::forward<T>(value)`. For the second overload, the type of the shared state is `void`.

4 *Postconditions:* For the returned `future`, `valid() == true` and `is_ready() == true`.

2.11 Function template `make_exceptional_future` [futures.make_exceptional_future]

1
```
template <class T>
  future<T> make_exceptional_future(exception_ptr ex);
```

2 *Effects:* Equivalent to

```
promise<T> p;
p.set_exception(ex);
return p.get_future();
```

3
```
template <class T, class E>
future<T> make_exceptional_future(E ex);
```

4 *Effects:* Equivalent to

```
promise<T> p;
p.set_exception(make_exception_ptr(ex));
return p.get_future();
```

3 Latches and Barriers [coordination]

3.1 General [coordination.general]

1 This section describes various concepts related to thread coordination, and defines the `latch`, `barrier` and `flex_barrier` classes.

3.2 Terminology [thread.coordination.terminology]

Editor's note: This section uses the term 'thread' throughout. Where relevant, it should be updated to refer to execution agents when these are adopted in the standard. See N4231 and N4156.

1 In this subclause, a *synchronization point* represents a point at which a thread may block until a given condition has been reached.

3.3 Latches [thread.coordination.latch]

1 Latches are a thread coordination mechanism that allow one or more threads to block until an operation is completed. An individual latch is a single-use object; once the operation has been completed, the latch cannot be reused.

3.4 Header <experimental/latch> synopsis [thread.coordination.latch.synopsis]

```
namespace std {
namespace experimental {
inline namespace concurrency_v1 {
  class latch {
   public:
    explicit latch(ptrdiff_t count);
    latch(const latch&) = delete;

    latch& operator=(const latch&) = delete;
    ~latch();


    void count_down_and_wait();
    void count_down(ptrdiff_t n);

    bool is_ready() const noexcept;
    void wait() const;

   private:
    ptrdiff_t counter_; // exposition only
  };
} // namespace concurrency_v1
} // namespace experimental
} // namespace std
```


3.5 Class `latch` [coordination.latch.class]

1 A latch maintains an internal `counter_` that is initialized when the latch is created. Threads may block at a synchronization point waiting for `counter_` to be decremented to `0`. When `counter_` reaches `0`, all such blocked threads are released.

2 Calls to `count_down_and_wait()`, `count_down()`, `wait()`, and `is_ready()` behave as atomic operations.

3 `explicit latch(ptrdiff_t count);`

4 *Requires:* `count >= 0`.

5 *Synchronization:* None.

6 *Postconditions:* `counter_ == count`.

7 `~latch();`

8 *Requires:* No threads are blocked at the synchronization point.

9 *Remarks:* May be called even if some threads have not yet returned from `wait()` or `count_down_and_wait()` provided that `counter_` is `0`. [*Note:* The destructor might not return until all threads have exited `wait()` or `count_down_and_wait()`. — *end note*]

10 `void count_down_and_wait();`

11 *Requires:* `counter_ > 0`.

12 *Effects:* Decrements `counter_` by `1`. Blocks at the synchronization point until `counter_` reaches `0`.

13 *Synchronization:* Synchronizes with all calls that block on this latch and with all `is_ready` calls on this latch that return true.

14 *Throws:* Nothing.

15 `void count_down(ptrdiff_t n);`

16 *Requires:* `counter_ >= n` and `n >= 0`.

17 *Effects:* Decrements `counter_` by `n`. Does not block.

18 *Synchronization:* Synchronizes with all calls that block on this latch and with all `is_ready` calls on this latch that return true.

19 *Throws:* Nothing.

20 `void wait() const;`

21 *Effects:* If `counter_` is `0`, returns immediately. Otherwise, blocks the calling thread at the synchronization point until `counter_` reaches `0`.

22 *Throws:* Nothing.

23 `is_ready() const noexcept;`

24 *Returns:* `counter_ == 0`. Does not block.

Editor's note: SG1 seems to have a convention that blocking functions are never marked noexcept (e.g. future::wait) even if they never throw. LWG requests that SG1 check whether this pattern is intended, and update the noexcept clauses here accordingly.

3.6 Barrier types [thread.coordination.barrier]

1 Barriers are a thread coordination mechanism that allow a *set of participating threads* to block until an operation is completed. Unlike a latch, a barrier is reusable: once the participating threads are released from a barrier's synchronization point, they can re-use the same barrier. It is thus useful for managing repeated tasks, or phases of a larger task, that are handled by multiple threads.

2 The *barrier types* are the standard library types `barrier` and `flex_barrier`. They shall meet the requirements set out in this subclause. In this description, `b` denotes an object of a barrier type.

3 Each barrier type defines a *completion phase* as a (possibly empty) set of effects. When the member functions defined in this subclause *arrive at the barrier's synchronization point*, they have the following effects:

1. The function blocks.
2. When all threads in the barrier's set of participating threads are blocked at its synchronization point, one participating thread is unblocked and executes the barrier type's completion phase.
3. When the completion phase is completed, all other participating threads are unblocked. The end of the completion phase synchronizes with the returns from all calls unblocked by its completion.

4 The expression `b.arrive_and_wait()` shall be well-formed and have the following semantics:

5 `void arrive_and_wait();`

6 *Requires:* The current thread is a member of the set of participating threads.

7 *Effects:* Arrives at the barrier's synchronization point. [*Note:* It is safe for a thread to call `arrive_and_wait()` or `arrive_and_drop()` again immediately. It is not necessary to ensure that all blocked threads have exited `arrive_and_wait()` before one thread calls it again. — *end note*]

8 *Synchronization:* The call to `arrive_and_wait()` synchronizes with the start of the completion phase.

9 *Throws:* Nothing.

10 The expression `b.arrive_and_drop()` shall be well-formed and have the following semantics:

11 `void arrive_and_drop();`

12 *Requires:* The current thread is a member of the set of participating threads.

13 *Effects:* Either arrives at the barrier's synchronization point and then removes the current thread from the set of participating threads, or just removes the current thread from the set of participating threads. [*Note:* Removing the current thread from the set of participating threads can cause the completion phase to start. — *end note*]

14 *Synchronization:* The call to `arrive_and_drop()` synchronizes with the start of the completion phase.

15 *Throws:* Nothing.

16 *Notes:* If all participating threads call `arrive_and_drop()`, any further operations on the barrier are undefined, apart from calling the destructor. If a thread that has called `arrive_and_drop()` calls another method on the same barrier, other than the destructor, the results are undefined.

17 Calls to `arrive_and_wait()` and `arrive_and_drop()` never introduce data races with themselves or each other.

3.7 Header <experimental/barrier> synopsis [thread.coordination.barrier.synopsis]

```
namespace std {
namespace experimental {
inline namespace concurrency_v1 {
  class barrier;
  class flex_barrier;
} // namespace concurrency_v1
} // namespace experimental
} // namespace std
```

3.8 Class `barrier` [coordination.barrier.class]

1 `barrier` is a barrier type whose completion phase has no effects. Its constructor takes a parameter representing the initial size of its set of participating threads.

```
class barrier {
 public:
  explicit barrier(ptrdiff_t num_threads);
  barrier(const barrier&) = delete;

  barrier& operator=(const barrier&) = delete;
  ~barrier();



  void arrive_and_wait();
  void arrive_and_drop();
};
```

2 `explicit barrier(ptrdiff_t num_threads);`

3 *Requires:* `num_threads >= 0`. [*Note:* If `num_threads` is zero, the barrier may only be destroyed. — *end note*]

4 *Effects:* Initializes the barrier for `num_threads` participating threads. [*Note:* The set of participating threads is the first `num_threads` threads to arrive at the synchronization point. — *end note*]

5 `~barrier();`

6 *Requires:* No threads are blocked at the synchronization point.

7 *Effects:* Destroys the barrier.

3.9 Class `flex_barrier` [coordination.flexbarrier.class]

1 `flex_barrier` is a barrier type whose completion phase can be controlled by a function object.

```
class flex_barrier {
 public:
  template <class F>
    flex_barrier(ptrdiff_t num_threads, F completion);
  explicit flex_barrier(ptrdiff_t num_threads);
  flex_barrier(const flex_barrier&) = delete;
  flex_barrier& operator=(const flex_barrier&) = delete;

  ~flex_barrier();

  void arrive_and_wait();
  void arrive_and_drop();

 private:
  function<ptrdiff_t()> completion_;  // exposition only
};
```

2 The completion phase calls `completion_()`. If this returns `-1`, then the set of participating threads is unchanged. Otherwise, the set of participating threads becomes a new set with a size equal to the returned value. [*Note:* If `completion_()` returns `0` then the set of participating threads becomes empty, and this object may only be destroyed. — *end note*]

3 `template <class F>`
    `flex_barrier(ptrdiff_t num_threads, F completion);`

4 *Requires:*

- `num_threads >= 0`.
- `F` shall be `CopyConstructible`.
- `completion` shall be Callable (C++14 §[func.wrap.func]) with no arguments and return type `ptrdiff_t`.
- Invoking `completion` shall return a value greater than or equal to `-1` and shall not exit via an exception.

5 *Effects:* Initializes the `flex_barrier` for `num_threads` participating threads, and initializes `completion_` with `std::move(completion)`. [*Note:* The set of participating threads consists of the first `num_threads` threads to arrive at the synchronization point. — *end note*] [*Note:* If `num_threads` is `0` the set of participating threads is empty, and this object may only be destroyed. — *end note*]

6 `explicit flex_barrier(ptrdiff_t num_threads);`

7 *Requires:* `num_threads >= 0`.

8 *Effects:* Has the same effect as creating a `flex_barrier` with `num_threads` and with a callable object whose invocation returns `-1` and has no side effects.

9 `~flex_barrier();`

10 *Requires:* No threads are blocked at the synchronization point.

11 *Effects:* Destroys the barrier.

4 Atomic Smart Pointers [atomic]

4.1 General [atomic.smartptr.general]

1 This section provides alternatives to raw pointers for thread-safe atomic pointer operations, and defines the `atomic_shared_ptr` and `atomic_weak_ptr` class templates.

2 The class templates `atomic_shared_ptr<T>` and `atomic_weak_ptr<T>` have the corresponding non-atomic types `shared_ptr<T>` and `weak_ptr<T>`. The template parameter `T` of these class templates may be an incomplete type.

3 The behavior of all operations is as specified in C++14 §29.6.5, unless stated otherwise.

4.2 Header <experimental/atomic> synopsis [atomic.smartptr.synop]

```
#include <atomic>


namespace std {
namespace experimental {
inline namespace concurrency_v1 {

  template <class T> struct atomic_shared_ptr;
  template <class T> struct atomic_weak_ptr;

} // namespace concurrency_v1
} // namespace experimental
} // namespace st
```

4.3 Class template `atomic_shared_ptr` [atomic.shared_ptr]

```
namespace std {
  namespace experimental {
  inline namespace concurrency_v1 {

  template <class T> struct atomic_shared_ptr {
    bool is_lock_free() const noexcept;
    void store(shared_ptr<T>, memory_order = memory_order_seq_cst) noexcept;
    shared_ptr<T> load(memory_order = memory_order_seq_cst) const noexcept;
    operator shared_ptr<T>() const noexcept;

    shared_ptr<T> exchange(shared_ptr<T>,
      memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_weak(shared_ptr<T>&, const shared_ptr<T>&,
      memory_order, memory_order) noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, shared_ptr<T>&&,
      memory_order,  memory_order) noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, const shared_ptr<T>&,
```



```
      memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, shared_ptr<T>&&,
      memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(shared_ptr<T>&, const shared_ptr<T>&,
      memory_order, memory_order) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, shared_ptr<T>&&,
      memory_order, memory_order) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, const shared_ptr<T>&,
      memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, shared_ptr<T>&&,
      memory_order = memory_order_seq_cst) noexcept;

    constexpr atomic_shared_ptr() noexcept = default;
    atomic_shared_ptr(shared_ptr<T>) noexcept;
    atomic_shared_ptr(const atomic_shared_ptr&) = delete;
    atomic_shared_ptr& operator=(const atomic_shared_ptr&) = delete;
    atomic_shared_ptr& operator=(shared_ptr<T>) noexcept;
  };
  } // namespace concurrency_v1
  } // namespace experimental
} // namespace std
```

1 `constexpr atomic_shared_ptr() noexcept = default;`

2 *Effects:* Initializes the atomic object to an empty value.

4.4 Class template `atomic_weak_ptr` [atomic.weak_ptr]

```
namespace std {
  namespace experimental {
  inline namespace concurrency_v1 {

  template <class T> struct atomic_weak_ptr {
    bool is_lock_free() const noexcept;
    void store(weak_ptr<T>, memory_order = memory_order_seq_cst) noexcept;
    weak_ptr<T> load(memory_order = memory_order_seq_cst) const noexcept;
    operator weak_ptr<T>() const noexcept;

    weak_ptr<T> exchange(weak_ptr<T>,
      memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_weak(weak_ptr<T>&, const weak_ptr<T>&,
      memory_order, memory_order) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, weak_ptr<T>&&,
      memory_order, memory_order) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, const weak_ptr<T>&,
      memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, weak_ptr<T>&&,
      memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(weak_ptr<T>&, const weak_ptr<T>&,
```



```
      memory_order, memory_order) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, weak_ptr<T>&&,
      memory_order, memory_order) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, const weak_ptr<T>&,
      memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, weak_ptr<T>&&,
      memory_order = memory_order_seq_cst) noexcept;

    constexpr atomic_weak_ptr() noexcept = default;
    atomic_weak_ptr(weak_ptr<T>) noexcept;
    atomic_weak_ptr(const atomic_weak_ptr&) = delete;
    atomic_weak_ptr& operator=(const atomic_weak_ptr&) = delete;
    atomic_weak_ptr& operator=(weak_ptr<T>) noexcept;
  };
  } // namespace concurrency_v1
  } // namespace experimental
} // namespace std
```

1 `constexpr atomic_weak_ptr() noexcept = default;`

2 *Effects:* Initializes the atomic object to an empty value.

3 When any operation on an `atomic_shared_ptr` or `atomic_weak_ptr` causes an object to be destroyed or memory to be deallocated, that destruction or deallocation shall be sequenced after the changes to the atomic object's state.

4 [*Note:* This prevents potential deadlock if the atomic smart pointer operation is not lock-free, such as by including a spinlock as part of the atomic object's state, and the destruction or the deallocation may attempt to acquire a lock. — *end note*]

5 [*Note:* These types replace all known uses of the functions in C++14 §20.8.2.6. — *end note*]

