Programming Languages — Technical Specification for C++ Extensions for Concurrency

Warning
This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.
Copyright notice

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

ISO copyright officer
Case postale 56, CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction may be subject to royalty payments or a licensing agreement.
Violators may be prosecuted.
## Contents

1 **General** ................................................................. 4  
1.1 Namespaces, headers, and modifications to standard classes ........................................... 4  
1.2 Future plans (Informative) ..................................................................................................... 4  
1.3 Feature-testing recommendations (Informative) ................................................................. 5  
2 **Improvements to std::future<T> and Related APIs** ....................................................... 6  
2.1 General ................................................................................................................................. 6  
2.2 Header <experimental/future> synopsis .................................................................................. 6  
2.3 Class template future ............................................................................................................. 7  
2.4 Class template shared_future ................................................................................................ 9  
2.5 Class template promise ......................................................................................................... 11  
2.6 Class template packaged_task .............................................................................................. 11  
2.7 Function template when_all .................................................................................................. 11  
2.8 Class template when_any_result .......................................................................................... 12  
2.9 Function template when_any ................................................................................................ 12  
2.10 Function template make_ready_future ............................................................................... 14  
2.11 Function template make_exceptional_future ..................................................................... 14  
3 **Latches and Barriers** ........................................................................................................... 15  
3.1 General .................................................................................................................................. 15  
3.2 Terminology ........................................................................................................................ 15  
3.3 Latches .................................................................................................................................. 15  
3.4 Header <experimental/latch> synopsis ................................................................................ 15  
3.5 Class latch ........................................................................................................................... 16  
3.6 Barrier types ........................................................................................................................ 17  
3.7 Header <experimental/barrier> synopsis ............................................................................. 18  
3.8 Class barrier ......................................................................................................................... 18  
3.9 Class flex_barrier .................................................................................................................. 18  
4 **Atomic Smart Pointers** ....................................................................................................... 20  
4.1 General ................................................................................................................................ 20  
4.2 Header <experimental/atomic> synopsis ............................................................................ 20  
4.3 Class template atomic_shared_ptr ....................................................................................... 20  
4.4 Class template atomic_weak_ptr ........................................................................................... 21
1 General

1.1 Namespaces, headers, and modifications to standard classes

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:

- modify an existing interface in the C++ Standard Library in-place,
- are declared in a namespace whose name appends std::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.

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

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

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::.

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

```cpp
#include <meow>
```

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

<table>
<thead>
<tr>
<th>Table 1 — C++ library headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;experimental/future&gt;</td>
</tr>
<tr>
<td>&lt;experimental/latch&gt;</td>
</tr>
<tr>
<td>&lt;experimental/barrier&gt;</td>
</tr>
<tr>
<td>&lt;experimental/atomic&gt;</td>
</tr>
</tbody>
</table>

1.2 Future plans (Informative)

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

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.

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)

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

<table>
<thead>
<tr>
<th>Doc. No.</th>
<th>Title</th>
<th>Primary Section</th>
<th>Macro Name Suffix</th>
<th>Value</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>N4399</td>
<td>Improvements to std::future&lt;T&gt; and Related APIs</td>
<td>2</td>
<td>future_continuations</td>
<td>201505</td>
<td>&lt;experimental/future&gt;</td>
</tr>
<tr>
<td>N4204</td>
<td>C++ Latches and Barriers</td>
<td>3</td>
<td>latch</td>
<td>201505</td>
<td>&lt;experimental/latch&gt;</td>
</tr>
<tr>
<td>N4204</td>
<td>C++ Latches and Barriers</td>
<td>3</td>
<td>barrier</td>
<td>201505</td>
<td>&lt;experimental/barrier&gt;</td>
</tr>
<tr>
<td>N4260</td>
<td>Atomic Smart Pointers</td>
<td>4</td>
<td>atomic_smart_pointers</td>
<td>201505</td>
<td>&lt;experimental/atomics&gt;</td>
</tr>
</tbody>
</table>
2 Improvements to std::future<T> and Related APIs

2.1 General

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

#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>
2.3 Class template `future`

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();
                futures operator=(const future&) = delete;
                futures 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;

            } // class future

        } // namespace concurrency_v1
    } // namespace experimental
}

} // namespace std

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

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

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.

Postconditions:
  — valid() == true.
  — rhs.valid() == false.

The member function template then provides a mechanism for attaching a continuation to a future object, which will be executed as specified below.
template <class F>
see below then(F&& func);

Requires: \(\text{INVOKE}(\text{DECAY\_COPY}(\text{std}::\text{forward}\langle F\rangle(\text{func})), \text{std}::\text{move}(\ast \text{this}))\) shall be a valid expression.

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
  \(\text{INVOKE}(\text{DECAY\_COPY}(\text{std}::\text{forward}\langle F\rangle(\text{func})), \text{std}::\text{move}(\ast \text{this}))\) is called on an unspecified thread of execution with the call to \(\text{DECAY\_COPY()}\) 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.

Returns: When result of \(\text{t<decay\_t<future\langle R\rangle>}(\text{future\langle R\rangle})\) is future\langle R2\rangle, for some type R2, the function returns future\langle R2\rangle. Otherwise, the function returns future\langle result of \(\text{t<decay\_t<future\langle R\rangle>}(\text{future\langle R\rangle})\)\rangle. [Note: The rule above is referred to as implicit unwrapping. Without this rule, the return type of then taking a callable returning a future\langle R\rangle would have been future\langle future\langle R\rangle\rangle. This rule avoids such nested future objects. The type of f2 below is future\langle int\rangle and not future\langle future\langle int\rangle\rangle:]

\[
\text{Example:}
\begin{align*}
  &\text{future}\langle \text{int}\rangle \ f1 = g(); \\
  &\text{future}\langle \text{int}\rangle \ f2 = f1.\text{then()}(\ [](\text{future}\langle \text{int}\rangle \ f) \ \{ \\
  &\quad \text{future}\langle \text{int}\rangle \ f3 = h(); \\
  &\quad \text{return} \ f3; \\
  &\});
\end{align*}
\]

- end example ]
- end note ]

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]

bool is_ready() const;

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

2.4 Class template \texttt{shared\_future} [futures.shared\_future]

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.
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) const;

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

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

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

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.

Postconditions:
— valid() == true.
— rhs.valid() == false.

The member function template then provides a mechanism for attaching a continuation to a shared_future object, which will be executed as specified below.
template <class F>
  see below then(F&& func) const;

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

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.

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]

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]

bool is_ready() const;

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

2.5 Class template promise

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.

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

2.6 Class template packaged_task

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.

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

2.7 Function template when_all

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.

§ 2.7
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 futures and shared_futures 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 \( \text{decay\_t}<F_i> \), and let \( U_i \) be \( \text{remove\_reference\_t}<F_i> \) 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> or \( U_i \) is a future<R>.

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 \( \text{first} == \text{last} \), 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 futures are moved, and any shared_futures are copied into, correspondingly, futures or
  shared_futures 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 futures and shared_futures supplied to the call to when_all are ready, the resulting future,
  as well as the futures and shared_futures 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
  futures and shared_futures held in the shared state may.

6 Postconditions:
— For the returned future, valid() == true.
— For all input futures, valid() == false.
— For all input shared_futures, valid() == true.

7 Returns: A future object that becomes ready when all of the input futures and shared_futures are ready.

2.8 Class template when_any_result

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

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.
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);

Requires: All futures and shared futures passed into when_all must be in a valid state (i.e. valid() == true).

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>.

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 futures are moved, and any shared futures are copied into, correspondingly, futures or
  shared futures 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 futures or shared futures 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
  futures and shared futures held in the shared state may.

Postconditions:
— For the returned future, valid() == true.
— For all input futures, valid() == false.
— For all input shared futures, valid() == true.

Returns:
— A future object that becomes ready when any of the input futures and shared futures are ready.
2.10 Function template make_ready_future

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

   future<void> make_ready_future();

2 Let $\text{U}$ be $\text{decay}\_t\text{<T>}. Then $V$ is $X&$ if $U$ equals $\text{reference}\_\text{wrapper}\langle X\rangle$, 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 $\text{void}$.

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

2.11 Function template 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

3.1 General

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

3.2 Terminology

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

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

```cpp
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:
                    pptrdiff_t counter_; // exposition only
                } // namespace concurrency_v1
            } // namespace experimental
        } // namespace std
    } // namespace experimental
} // namespace std
```

© ISO/IEC N4538 § 3.4
3.5 Class latch

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.

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

```cpp
explicit latch(ptrdiff_t count);

Requires: count >= 0.

Synchronization: None.

Postconditions: counter_ == count.
```

~latch();

`Requires`: No threads are blocked at the synchronization point.

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]

```cpp
void count_down_and_wait();

Requires: counter_ > 0.

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

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

Throws: Nothing.
```

```cpp
void count_down(ptrdiff_t n);

Requires: counter_ >= n and n >= 0.

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

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

Throws: Nothing.
```

```cpp
void wait() const;

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

Throws: Nothing.
```

```cpp
is_ready() const noexcept;

Returns: `counter_` == 0. Does not block.
```
3.6 Barrier types

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.

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.

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.

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

```c
void arrive_and_wait();
```

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

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 ]

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

Throws: Nothing.

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

```c
void arrive_and_drop();
```

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

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 ]

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

Throws: Nothing.

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

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

3.8 Class barrier

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();

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

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

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
};

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 pptrdiff_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

4.1 General

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.

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.

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

4.2 Header <experimental/atomic> synopsis

```cpp
#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 std
```

4.3 Class template atomic_shared_ptr

```cpp
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;

            }

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

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;
        }
    }
} // namespace experimental
} // namespace std

§ 4.4

© ISO/IEC N4538
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 ]