A Cooperatively Interruptible Joining Thread, Rev 4

New in R4

— Removed interruptible CV waiting members that don’t take a predicate.
— Removed adding a new `cv_status` value `interrupted`.
— Added CV members for interruptible timed waits.
— Renamed CV members that wait interruptible.
— Several minor fixes (e.g. on `noexcept`) and full proposed wording.

Purpose

This is the proposed wording for a cooperatively interruptible joining thread.

For a full discussion of the motivation, see [www.wg21.link/p0660r0](www.wg21.link/p0660r0) and [www.wg21.link/p0660r1](www.wg21.link/p0660r1).

A default implementation exists at: [github.com/josuttis/jthread](http://github.com/josuttis/jthread). Note that the proposed functionality can be fully implemented on top of the existing C++ standard library.

Basis examples:

```cpp
void testJThread()
{
    std::jthread t([]
    {
        while (...) {
            std::this_thread::throw_if_interrupted();
            ...
            // temporarily disable interrupts:
            auto origToken = std::this_thread::exchange_interrupt_token(std::interrupt_token{});
            ...
            std::this_thread::exchange_interrupt_token(origToken);
            ...
        }
    });

    ...
} // jthread destructor signals interrupt and therefore silently ends the started thread
```

```cpp
void testInterruptibleCVWait()
{
    bool ready = false;
    std::mutex readyMutex;
    std::condition_variable readyCV;
    std::jthread t([&ready, &readyMutex, &readyCV] {
        while (...) {
            {
                std::unique_lock lg(readyMutex);
                readyCV.iwait_for(lg,
                    std::chrono::seconds(2),
                    [&ready] {
                        return ready;
                    });
            }
            ...
        }
    });

    ...
} // jthread destructor signals interrupt and therefore unblocks the CV wait and silently ends the started thread
```
Feature Test Macro

This is a new feature so that it shall have the following feature macro:

```
__cpp_lib_jthread
```

Acknowledgements

Thanks to all who incredibly helped me to prepare this paper, such as all people in the C++ concurrency and library working group. Especially, we want to thank: Hans Boehm, Olivier Giroux, Pablo Halpern, Howard Hinnant, Alisdair Meredith, Gor Nishanov, Ville Voutilainen, and Jonathan Wakely.

Proposed Wording

All against N4659.

[Editorial note: This proposal uses the LaTeX macros of the draft standard. To adopt it please ask for the LaTeX source code of the proposed wording.]
30  Thread support library

30.1  General

The following subclauses describe components to create and manage threads, perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 1.

Table 1 — Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2  Requirements</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>30.3  Threads</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>30.4  Interrupt Tokens</td>
<td>&lt;interrupt_token&gt;</td>
</tr>
<tr>
<td>30.5  Joining Threads</td>
<td>&lt;jthread&gt;</td>
</tr>
<tr>
<td>??  Mutual exclusion</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td>??  Futures</td>
<td>&lt;future&gt;</td>
</tr>
</tbody>
</table>

30.2  Requirements

30.3  Threads

30.3.2.2  thread constructors

thread() noexcept;

Effects: Constructs a thread object that does not represent a thread of execution.

Ensures: get_id() == id().

template<class F, class... Args> explicit thread(F&& f, Args&&... args);

Requires: F and each T< in Args shall satisfy the Cpp17MoveConstructible requirements. INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...) (??) shall be a valid expression.

Remarks: This constructor shall not participate in overload resolution if remove_cvref_t<F> is the same type as std::thread.

Effects: Constructs an object of type thread. The new thread of execution executes INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...) with the calls to DECAY_COPY being evaluated in the constructing thread. Any return value from this invocation is ignored. [Note: This implies that any exceptions not thrown from the invocation of the copy of f will be thrown in the constructing thread, not the new thread. — end note] If the invocation of INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...) terminates with an uncaught exception, terminate shall be called.

An uncaught interrupted exception in the started thread of execution will silently be ignored. [Note: Thus, an uncaught exception thrown by this_thread::throw_if_interrupted() will cause the started thread to end silently. — end note]

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.

Ensures: get_id() != id(). *this represents the newly started thread.

Throws: system_error if unable to start the new thread.

Error conditions:
— resource_unavailable_try_again — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.
30.4 Interrupt Tokens

30.4 describes components that can be used to asynchronously signal an interrupt. The interrupt can only be signaled once.

30.4.1 Header <interrupt_token> synopsis

```cpp
namespace std {
    // 30.4.2 class interrupted
class interrupted;

    // 30.4.3 class interrupt_token
class interrupt_token;
}
```

30.4.2 Class interrupted

```cpp
namespace std {
    class interrupted {
        public:
            explicit interrupted() noexcept;
            explicit interrupted(const interrupted&) noexcept;
            interrupted& operator=(const interrupted&) noexcept;
            const char* what() const noexcept;
    };
}
```

1 The class `interrupted` defines the type of objects thrown as exceptions by C++ standard library components, and certain expressions, to report a signaled interrupt. [Note: This class is not derived from class `exception`. —end note]

```cpp
interrupted() noexcept;
```

2 Effects: Constructs an object of class `interrupted`.

```cpp
interrupted(const interrupted& rhs) noexcept;
interrupted& operator=(const interrupted& rhs) noexcept;
```

3 Effects: Copies an object of class `interrupted`.

```cpp
const char* what() const noexcept;
```

4 Ensures: If `*this` and `rhs` both have dynamic type `interrupted` then the value of the expression `strcmp(what(), rhs.what())` shall equal 0.

5 Returns: An implementation-defined ntbs.

6 Remarks: The message may be a null-terminated multibyte string (??), suitable for conversion and display as a wstring (??, ??). The return value remains valid until the exception object from which it is obtained is destroyed or a non-const member function of the exception object is called.

30.4.3 Class interrupt_token

```cpp
namespace std {
    class interrupt_token {
        public:
            // 30.4.3.1 create, copy, destroy:
            explicit interrupt_token() noexcept;
            explicit interrupt_token(bool initial_state);

            interrupt_token(const interrupt_token&) noexcept;
            interrupt_token(interrupt_token&&) noexcept;
            interrupt_token& operator=(const interrupt_token&) noexcept;
            interrupt_token& operator=(interrupt_token&&) noexcept;
            ~interrupt_token();
    };
}
```

1 The class `interrupt_token` implements semantics of shared ownership of an interrupt state (an atomic token to signal an interrupt). An interrupt can only be signaled once. All owners can signal an interrupt, provided the token is valid. All owners can check whether an interrupt was signaled. The last remaining owner of the interrupt state is responsible for releasing the resources associated with the interrupt state.
void swap(interrupt_token&) noexcept;

// 30.4.3.5 interrupt handling:
bool valid() const noexcept;
bool is_interrupted() const noexcept;
bool interrupt();
void throw_if_interrupted() const;
}
}
bool operator==(const interrupt_token& lhs, const interrupt_token& rhs);
bool operator!=(const interrupt_token& lhs, const interrupt_token& rhs);

Calls to interrupt(), is_interrupted(), and throw_if_interrupted() are atomic operations(6.8.2.1p3 ??) on an atomic object contained in the interrupt_token. Hence concurrent calls to these functions do not introduce data races. A call to interrupt() synchronizes with any call to interrupt(), is_interrupted(), or throw_if_interrupted() that observes the interrupt.

30.4.3.1 interrupt_token constructors

interrupt_token() noexcept;
1 Effects: Constructs a new interrupt_token object that can’t be used to signal interrupts. [Note: Therefore, no resources have to be associated for the state. — end note]
2 Ensures: valid() == false.

interrupt_token(bool initial_state) noexcept;
3 Effects: Constructs a new interrupt_token object that can signal interrupts via an atomic associated interrupt state.
4 Ensures: valid() == true and is_interrupted() == initial_state.

interrupt_token(const interrupt_token& rhs) noexcept;
5 Effects: If rhs is not valid, constructs an interrupt_token object that is not valid; otherwise, constructs an interrupt_token that shares the ownership of the interrupt state with rhs.
6 Ensures: valid() == rhs.valid() and is_interrupted() == rhs.is_interrupted() and *this == rhs.

interrupt_token(interrupt_token&& rhs) noexcept;
7 Effects: Move constructs an object of type interrupt_token from rhs.
8 Ensures: *this shall contain the old value of rhs and rhs.valid() == false.

30.4.3.2 interrupt_token destructor

~interrupt_token();
1 Effects: If valid() and *this is the last owner of the interrupt state, releases the associated with the interrupt state.

30.4.3.3 interrupt_token assignment

interrupt_token& operator=(const interrupt_token& rhs) noexcept;
1 Effects: Equivalent to: interrupt_token(rhs).swap(*this);
2 Returns: *this.

interrupt_token& operator=(interrupt_token&& rhs) noexcept;
3 Effects: Equivalent to: interrupt_token(std::move(rhs)).swap(*this);
4 Returns: *this.

30.4.3.4 interrupt_token swap

void swap(interrupt_token& rhs) noexcept;
1 Effects: Swaps the state of *this and rhs.
30.4.3.5 interrupt_token members

bool valid() const noexcept;

1 Returns: true if the interrupt token can be used to signal interrupts.

bool is_interrupted() const noexcept;

2 Returns: true if initialized with true or initialized with false and interrupt() was called by one of the owners.

bool interrupt();

3 Effects: If !valid() or is_interrupted() the call has no effect. Otherwise, signals an interrupt so that is_interrupted() == true. [Note: Signaling an interrupt includes notifying all condition_variables temporarily registered via a an interruptable wait (30.6.1.2) — end note]

Ensures: !valid() || is_interrupted()

4 Returns: The value of is_interrupted() prior to the call.

void throw_if_interrupted() const;

5 Effects: Equivalent to:

   if (is_interrupted())
   throw interrupted();

30.4.3.6 interrupt_token comparisons

bool operator== (const interrupt_token& lhs, const interrupt_token& rhs);

1 Returns: !lhs.valid() && !rhs.valid() or whether lhs and rhs refer to the same interrupt state (copied or moved from the same initial interrupt_token object).

bool operator!= (const interrupt_token& lhs, const interrupt_token& rhs);

2 Returns: !(lhs==rhs).
30.5 Joining Threads

30.5 describes components that can be used to create and manage threads with the ability to signal interrupts to cooperatively cancel the running thread.

30.5.1 Header <jthread> synopsis

#include <interrupt_token>

namespace std {
    // 30.5.2 class jthread
    class jthread;

    void swap(jthread& x, jthread& y) noexcept;

    // 30.5.3 this_thread interrupt token handling
    namespace this_thread {
        static bool is_interrupted() noexcept;
        static void throw_if_interrupted();
        static interrupt_token get_interrupt_token() noexcept;
        static interrupt_token exchange_interrupt_token(const interrupt_token&) noexcept;
    }
}

[Editorial note: The this_thread extensions listed here might instead be added to ’33.3.3 Namespace this_thread.]  

30.5.2 Class jthread

The class jthread provides a mechanism to create a new thread of execution. The functionality is identical to class thread with the additional ability to signal an interrupt and to automatically join() the started thread.

[Editorial note: This color signals differences to class std::thread.]

namespace std {
    class jthread {
        public:
            // types
            using id = thread::id;
            using native_handle_type = thread::native_handle_type;

            // construct/copy/destroy
            jthread() noexcept;
            template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
            ~jthread();
            jthread(const jthread&) = delete;
            jthread(jthread&&) noexcept;
            jthread& operator=(const jthread&) = delete;
            jthread& operator=(jthread&&) noexcept;

            // members
            void swap(jthread&) noexcept;
            bool joinable() const noexcept;
            void join();
            void detach();
            id get_id() const noexcept;
            native_handle_type native_handle();  // see ??

            // interrupt token handling
            interrupt_token get_original_interrupt_token() const noexcept;
            bool interrupt() noexcept;

            // static members
            static unsigned int hardware_concurrency() noexcept;

        private:
            interrupt_token itoken;  // exposition only
        };
    }
}
30.5.2.1 jthread constructors

jthread() noexcept;

**Effects:** Constructs a jthread object that does not represent a thread of execution.

**Ensures:** get_id() == id() and itoken.valid() == false.

```cpp
template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
```

**Requires:** F and each T_i in Args shall satisfy the Cpp17MoveConstructible requirements. `INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...)` shall be a valid expression.

**Remarks:** This constructor shall not participate in overload resolution if `remove_cvref_t<F>` is the same type as std::jthread.

**Effects:** Constructs an object of type jthread. The new thread of execution executes `INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...)` with the calls to `DECAY_COPY` being evaluated in the constructing thread. Any return value from this invocation is ignored. [Note: This implies that any exceptions not thrown from the invocation of the copy of f will be thrown in the constructing thread, not the new thread. —end note] If the invocation of `INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...)` terminates with an uncaught exception, `terminate` shall be called.

An uncaught interrupted exception in the started thread of execution will silently be ignored. [Note: Thus, an uncaught exception thrown by `this_thread::throw_if_interrupted()` will cause the started thread to end silently. —end note]

**Synchronization:** The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.

**Ensures:** get_id() != id(). itoken.valid() == true. *this represents the newly started thread.

In the started thread of execution `this_thread::thread_itoken` is an interrupt_token equal to itoken. [Note: Note that the calling thread can signal an interrupt only once, because it can’t replace this interrupt token. —end note]

**Throws:** system_error if unable to start the new thread.

**Error conditions:**

9

(9.1) — resource_unavailable_try_again — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

`jthread(jthread&& x) noexcept;`

**Effects:** Constructs an object of type jthread from x, and sets x to a default constructed state.

**Ensures:** x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction. itoken yields the value of x.itoken prior to the start of construction and x.itoken.valid() == false.

30.5.2.2 jthread destructor

`~jthread();`

**If joinable(), calls interrupt() and join(). Otherwise, has no effects. [Note: Operations on *this are not synchronized. —end note]**

30.5.2.3 jthread assignment

`jthread& operator=(jthread&& x) noexcept;`

**Effects:** If joinable(), calls interrupt() and join(). Assigns the state of x to *this and sets x to a default constructed state.

**Ensures:** x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment. itoken yields the value of x.itoken prior to the assignment and x.itoken.valid() == false.

**Returns:** *this.
30.5.2.4 jthread interrupt members

interrupt_token get_original_interrupt_token() const noexcept

1    Effects: Equivalent to: return itoken;

bool interrupt() noexcept;

2    Effects: Equivalent to: return itoken.interrupt();

30.5.3 Namespace this_thread Interrupt Handling

[Editorial note: The this_thread extensions listed here might instead be added to 33.3.3 Namespace this_thread.]

To be able to deal with signaled interrupt this_thread provides the following access to an interrupt_token (30.4.3):

namespace std::this_thread {
    interrupt_token thread_itoken;       // exposition only
    static interrupt_token get_interrupt_token() noexcept;
    static bool is_interrupted() noexcept;
    static void throw_if_interrupted();
    static interrupt_token exchange_interrupt_token(const interrupt_token&) noexcept;
}

For any thread of execution, thread_itoken is default initialized unless the thread was started with a constructor of class jthread (30.5.2).

interrupt_token get_interrupt_token() noexcept;

1    Returns: this_thread::thread_itoken.

bool is_interrupted() noexcept;

2    Returns: this_thread::get_interrupt_token().is_interrupted().

void throw_if_interrupted();

3    Effects: Equivalent to: this_thread::get_interrupt_token().throw_if_interrupted();

interrupt_token exchange_interrupt_token(const interrupt_token& it) noexcept;

4    Effects: Equivalent to: this_thread::itoken = it;

5    Returns: this_thread::thread_itoken prior to the exchange.

[Note: By calling this function for the first time on an object this_thread::get_interrupt_token() will no longer signal interrupts from the calling thread unless the returned token is restored. — end note]
30.6 Condition variables

30.6.1 Class condition_variable

namespace std {
    class condition_variable {
        public:
            condition_variable();
            ~condition_variable();

            condition_variable(const condition_variable&) = delete;
            condition_variable& operator=(const condition_variable&) = delete;

        void notify_one() noexcept;
        void notify_all() noexcept;

        // 30.6.1.1 noninterruptable waits:
        void wait(unique_lock<mutex>& lock);
        template<class Predicate>
        void wait(unique_lock<mutex>& lock, Predicate pred);

        template<class Clock, class Duration>
        cv_status wait_until(unique_lock<mutex>& lock,
            const chrono::time_point<Clock, Duration>& abs_time);
        template<class Clock, class Duration, class Predicate>
        bool wait_until(unique_lock<mutex>& lock,
            const chrono::time_point<Clock, Duration>& abs_time,
            Predicate pred);

        template<class Rep, class Period>
        cv_status wait_for(unique_lock<mutex>& lock,
            const chrono::duration<Rep, Period>& rel_time);
        template<class Rep, class Period, class Predicate>
        bool wait_for(unique_lock<mutex>& lock,
            const chrono::duration<Rep, Period>& rel_time,
            Predicate pred);

        // 30.6.1.2 interruptable waits:
        template<class Predicate>
        void iwait(unique_lock<mutex>& lock, Predicate pred);

        template<class Clock, class Duration, class Predicate>
        bool iwait_until(unique_lock<mutex>& lock,
            const chrono::time_point<Clock, Duration>& abs_time,
            Predicate pred);

        template<class Rep, class Period, class Predicate>
        bool iwait_for(unique_lock<mutex>& lock,
            const chrono::duration<Rep, Period>& rel_time,
            Predicate pred);

        // 30.6.1.3 interrupt_token waits:
        template <class Predicate>
        bool wait_until(unique_lock<mutex>& lock,
            Predicate pred,
            interrupt_token itoken);

        template <class Clock, class Duration, class Predicate>
        bool wait_until(unique_lock<mutex>& lock,
            const chrono::time_point<Clock, Duration>& abs_time
            Predicate pred,
            interrupt_token itoken);

        template <class Rep, class Period, class Predicate>
        bool wait_for(unique_lock<mutex>& lock,
            const chrono::duration<Rep, Period>& rel_time,
            Predicate pred,
            interrupt_token itoken);
    }
}
The class `condition_variable` shall be a standard-layout class.

```cpp
condition_variable();
```

**Effects:** Constructs an object of type `condition_variable`.  
**Throws:** `system_error` when an exception is required.  
**Error conditions:**
- `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

```cpp
~condition_variable();
```

**Requires:** There shall be no thread blocked on `*this`.  
[Note: That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. The user should take care to ensure that no threads wait on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. —end note]

**Effects:** Destroys the object.

```cpp
void notify_one() noexcept;
```

**Effects:** If any threads are blocked waiting for `*this`, unblocks one of those threads.

```cpp
void notify_all() noexcept;
```

### 30.6.1.1 Noninterruptable waits

**Requires:** `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either
- no other thread is waiting on this `condition_variable` object or
- `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until`, `iwait`, `iwait_for`, or `iwait_until`) threads.

**Effects:**
- Atomically calls `lock.unlock()` and blocks on `*this`.  
- When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.  
- The function will unblock when signaled by a call to `notify_one()` or a call to `notify_all()`, or spuriously.

**Remarks:** If the function fails to meet the postcondition, `terminate()` shall be called.  
[Note: This can happen if the re-locking of the mutex throws an exception. —end note]

**Ensures:** `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.  
**Throws:** Nothing.

```cpp
template<class Predicate>
void wait(unique_lock<mutex>& lock, Predicate pred);
```

**Requires:** `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either
- no other thread is waiting on this `condition_variable` object or
- `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until`, `iwait`, `iwait_for`, or `iwait_until`) threads.
Effects: Equivalent to:
   
   ```
   while (!pred())
     wait(lock);
   ```

Remarks: If the function fails to meet the postcondition, `terminate()` shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

Throws: Any exception thrown by `pred`.

```template<class Clock, class Duration>
cv_status wait_until(unique_lock<mutex>& lock, 
                    const chrono::time_point<Clock, Duration>& abs_time);
```

Requires: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either
   - no other thread is waiting on this `condition_variable` object or
   - `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until`, `iwait`, `iwait_for`, or `iwait_until`) threads.

Effects:
   - Atomically calls `lock.unlock()` and blocks on *this.
   - When unblocked, calls `lock.lock()` (possibly blocking on the lock), then returns.
   - The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (??) specified by `abs_time`, or spuriously.
   - If the function exits via an exception, `lock.lock()` shall be called prior to exiting the function.

Remarks: If the function fails to meet the postcondition, `terminate()` shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

Returns: `cv_status::timeout` if the absolute timeout (??) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

Throws: Timeout-related exceptions (??).

```template<class Rep, class Period>
cv_status wait_for(unique_lock<mutex>& lock, 
                    const chrono::duration<Rep, Period>& rel_time);
```

Requires: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either
   - no other thread is waiting on this `condition_variable` object or
   - `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until`, `iwait`, `iwait_for`, or `iwait_until`) threads.

Effects: Equivalent to:
   ```
   return wait_until(lock, chrono::steady_clock::now() + rel_time);
   ```

Returns: `cv_status::timeout` if the relative timeout (??) specified by `rel_time` expired, otherwise `cv_status::no_timeout`.

Remarks: If the function fails to meet the postcondition, `terminate()` shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

Throws: Timeout-related exceptions (??).

```template<class Clock, class Duration, class Predicate>
bool wait_until(unique_lock<mutex>& lock, 
                const chrono::time_point<Clock, Duration>& abs_time, 
                Predicate pred);
```

Requires: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread, and either
   - no other thread is waiting on this `condition_variable` object or
   - `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting (via `wait`, `wait_for`, `wait_until`, `iwait`, `iwait_for`, or `iwait_until`) threads.

Effects: Equivalent to:
   ```
   return wait_until(lock, abs_time + pred());
   ```

Returns: `false` if the absolute timeout (??) specified by `abs_time` expired, otherwise `true`.

Remarks: If the function fails to meet the postcondition, `terminate()` shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: `lock.owns_lock()` is `true` and `lock.mutex()` is locked by the calling thread.

Throws: Timeout-related exceptions (??).

§ 30.6.1.1
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

**Effects:** Equivalent to:

```
while (!pred())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
    return true;
```

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Ensures:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered. — end note]

**Throws:** Timeout-related exceptions (??) or any exception thrown by pred.

```c++
template<class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
              const chrono::duration<Rep, Period>& rel_time,
              Predicate pred);
```

**Requires:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

(1.1) no other thread is waiting on this condition_variable object or

(1.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

**Effects:** Equivalent to:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Ensures:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered. — end note]

**Throws:** Timeout-related exceptions (??) or any exception thrown by pred.

### 30.6.1.2 Interruptable waits

[thread.condition.iwait]

The following functions ensure to get notified if an interrupt is signaled for this_thread::get_interrupt_token(). In that case they throw interrupted as if this_thread::throw_if_interrupted() was called.

[Editorial note: Because all signatures here in the effects clause call is_interrupted() or throw_if_interrupted(), we don’t need wording that the calls synchronize with interrupt().]

[Editorial note: This color signals differences to the corresponding wait...() functions.]

```c++
template<class Predicate>
void iwait(unique_lock<mutex>& lock, Predicate pred);
```

**Requires:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

(1.1) no other thread is waiting on this condition_variable object or

(1.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

**Effects:** Registers *this to get notified when an interrupt is signaled on this_thread::get_interrupt_token() during this call and then equivalent to:

```
while(!pred()) {
    this_thread::throw_if_interrupted();
    cv.wait(lock, [&pred] {
        return pred() || this_thread::is_interrupted();
    });
```

§ 30.6.1.2
```cpp
template<class Clock, class Duration, class Predicate>
bool iwait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time,
    Predicate pred);

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(6.1) no other thread is waiting on this condition_variable object or
(6.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently
waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

Effects: Registers *this to get notified when an interrupt is signaled on this_thread::get_interrupt_token() during this call and then equivalent to:
while(!pred() && Clock::now() < abs_time) {
    this_thread::throw_if_interrupted();
    cv.wait_until(lock, abs_time,
        [&pred] {
            return pred() || this_thread::is_interrupted();
        });
    return pred();
}

Remarks: If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Throws: Any exception thrown by pred or exception interrupted if this_thread::is_interrupted().
```

```cpp
template<class Rep, class Period, class Predicate>
bool iwait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time,
    Predicate pred);

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(12.1) no other thread is waiting on this condition_variable object or
(12.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently
waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

Effects: Equivalent to:
this_thread::throw_if_interrupted();
return iwait_until(lock,
    std::chrono::steady_clock::now() + rel_time,
    std::move(pred));

Remarks: If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered or an interrupt was signaled. — end note]

Throws: Timeout-related exceptions (??) or any exception thrown by pred or exception interrupted if this_thread::is_interrupted().
```
30.6.1.3 interrupt_token waits

The following functions ensure to get notified if an interrupt is signaled for the passed interrupt_token. In that case they return false (if the predicate evaluates to false).

[Editorial note: Because all signatures here in the effects clause call is_interrupted() or throw_if_interrupted(), we don’t need wording that the calls synchronize with interrupt().]

```cpp
template <class Predicate>
bool wait_until(unique_lock<mutex>& lock,
    Predicate pred,
    interrupt_token itoken);
```

[Editorial note: This color signals differences to the corresponding wait() function without the interrupt token parameter.]

1 Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(1.1) no other thread is waiting on this condition_variable object or
(1.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

2 Effects: Registers *this to get notified when an interrupt is signaled on itoken during this call and then equivalent to:

```cpp
while(!pred() && !itoken.is_interrupted()) {
    cv.wait(lock, [pred, &itoken] {
        return pred() || itoken.is_interrupted();
    });
}
return pred();
```

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered or an interrupt was signaled. — end note]

3 Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

4 Remarks: If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

5 Throws: Any exception thrown by pred.

```cpp
template <class Clock, class Duration, class Predicate>
bool wait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time
    Predicate pred,
    interrupt_token itoken);
```

[Editorial note: This color signals differences to the corresponding wait_until() function without the interrupt token parameter.]

6 Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(7.1) no other thread is waiting on this condition_variable object or
(7.2) lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

7 Effects: Registers *this to get notified when an interrupt is signaled on itoken during this call and then equivalent to:

```cpp
while(!pred() && !itoken.is_interrupted() && Clock::now() < abs_time) {
    cv.wait_until(lock,
        abs_time,
        [pred, &itoken] {
            return pred() || itoken.is_interrupted();
        });
}
return pred();
```

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered or an interrupt was signaled. — end note]
Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Remarks: If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Throws: Timeout-related exceptions (??) or any exception thrown by pred.

template <class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
               const chrono::duration<Rep, Period>& rel_time,
               Predicate pred,
               interrupt_token itoken);

[Editorial note: This color signals differences to the corresponding wait_for() function without the interrupt token parameter. ]

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
— no other thread is waiting on this condition_variable object or
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, wait_until, iwait, iwait_for, or iwait_until) threads.

Effects: Equivalent to:
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred),
                 std::move(itoken));

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the timeout was triggered or an interrupt was signaled. — end note]

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Remarks: If the function fails to meet the postcondition, terminate() shall be called (??). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Throws: Timeout-related exceptions (??) or any exception thrown by pred.