Mandating the Standard Library: Clause 32 - Thread support library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

- Change 'participate in overload resolution' wording into "Constraints' elements
- Change 'Requires' elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.
- Drive-by fixes (hopefully very few)

This paper covers Clause 32 (Thread Support Library)

The entire clause is reproduced here, but the changes are confined to a few sections:

- thread.req.paramname 32.2.1
- thread.req.exception 32.2.2
- thread.req.lockable.general 32.2.5.1
- thread.req.lockable.basic 32.2.5.2
- thread.thread.class 32.4.2
- thread.thread.constr 32.4.2.2
- thread.thread.member 32.4.2.5
- thread.jthread.cons 32.4.3.1
- thread.jthread.mem 32.4.3.2
- thread.thread.this 32.4.4
- thread.mutex.requirements.mutex 32.5.3.2
- thread.mutex.class 32.5.3.2.1
- thread.mutex.recursive 32.5.3.2.2
- thread.timedmutex.requirements 32.5.3.3
- thread.timedmutex.class 32.5.3.3.1
- thread.timedmutex.recursive 32.5.3.3.2
- thread.sharedmutex.requirements 32.5.3.4
- thread.sharedmutex.class 32.5.3.4.1
- thread.sharedtimedmutex.requirements 32.5.3.5
- thread.sharedtimedmutex.class 32.5.3.5.1
- thread.lock 32.5.4
- thread.lock.guard 32.5.4.1
- thread.lock.scoped 32.5.4.2
- thread.lock.unique 32.5.4.3
- thread.lock.unique.cons 32.5.4.3.1
- thread.lock.unique.locking 32.5.4.3.2
- thread.lock.shared 32.5.4.4
- thread.lock.shared.cons 32.5.4.4.1
- thread.lock.algorithm 32.5.5
- thread.once.onceflag 32.5.6.1
- thread.once.callonce 32.5.6.2
- thread.condition 32.6
- thread.condition.nonmember 32.6.2
- thread.condition.condvar 32.6.3
- thread.condition.condvarany 32.6.4
- futures.errors 32.9.3
- futures.future_error 32.9.4
- futures.state 32.9.5
- futures.promise 32.9.6
- futures.unique_future 32.9.7
- futures.shared_future 32.9.8
- futures.async 32.9.9
- futures.task.members 32.9.10.1
Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:dsunder/draft.git dsunder-draft
cd dsunder-draft
git diff master..P1622 -- source/threads.tex
```

_Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525._
32 Thread support library [thread]

32.1 General [thread.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 145.

Table 145: Thread support library summary [tab:thread.summary]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>32.3</td>
<td>Stop tokens &lt;stop_token&gt;</td>
</tr>
<tr>
<td>32.4</td>
<td>Threads &lt;thread&gt;</td>
</tr>
<tr>
<td>32.5</td>
<td>Mutual exclusion &lt;mutex&gt;, &lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>32.6</td>
<td>Condition variables &lt;condition_variable&gt;</td>
</tr>
<tr>
<td>32.7</td>
<td>Semaphores &lt;semaphore&gt;</td>
</tr>
<tr>
<td>32.8</td>
<td>Coordination types &lt;latch&gt; &lt;barrier&gt;</td>
</tr>
<tr>
<td>32.9</td>
<td>Futures &lt;future&gt;</td>
</tr>
</tbody>
</table>

32.2 Requirements [thread.req]

32.2.1 Template parameter names [thread.req.paramname]

Throughout this Clause, the names of template parameters are used to express type requirements. If a template parameter is named `Predicate`, `operator()` applied to the template argument shall return a value that is convertible to `bool`. If a template parameter is named `Clock`, the corresponding template argument shall be a type `C` for which `is_clock_v<C>` is true; otherwise the program is ill-formed.

32.2.2 Exceptions [thread.req.exception]

Some functions described in this Clause are specified to throw exceptions of type `system_error` (??). Such exceptions shall be thrown if any of the function’s error conditions is detected or a call to an operating system or other underlying API results in an error that prevents the library function from meeting its specifications. Failure to allocate storage shall be reported as described in ??.

[Example: Consider a function in this clause that is specified to throw exceptions of type `system_error` and specifies error conditions that include `operation_not_permitted` for a thread that does not have the privilege to perform the operation. Assume that, during the execution of this function, an `errno` of `EPERM` is reported by a POSIX API call used by the implementation. Since POSIX specifies an `errno` of `EPERM` when “the caller does not have the privilege to perform the operation”, the implementation maps `EPERM` to an `error_condition` of `operation_not_permitted` (??) and an exception of type `system_error` is thrown. — end example]

The `error_code` reported by such an exception’s `code()` member function shall compare equal to one of the conditions specified in the function’s error condition element.

32.2.3 Native handles [thread.req.native]

Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation-defined. [Note: These members allow implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — end note]

32.2.4 Timing specifications [thread.req.timing]

Several functions described in this Clause take an argument to specify a timeout. These timeouts are specified as either a `duration` or a `time_point` type as specified in ??.

Implementations necessarily have some delay in returning from a timeout. Any overhead in interrupt response, function return, and scheduling induces a “quality of implementation” delay, expressed as duration $D_i$. Ideally,
this delay would be zero. Further, any contention for processor and memory resources induces a “quality of management” delay, expressed as duration $D_m$. The delay durations may vary from timeout to timeout, but in all cases shorter is better.

3 The functions whose names end in _for take an argument that specifies a duration. These functions produce relative timeouts. Implementations should use a steady clock to measure time for these functions.\footnote{327} Given a duration argument $D_t$, the real-time duration of the timeout is $D_t + D_i + D_m$.

4 The functions whose names end in _until take an argument that specifies a time point. These functions produce absolute timeouts. Implementations should use the clock specified in the time point to measure time for these functions. Given a clock time point argument $C_t$, the clock time point of the return from timeout should be $C_t + D_i + D_m$ when the clock is not adjusted during the timeout. If the clock is adjusted to the time $C_a$ during the timeout, the behavior should be as follows:

\begin{enumerate}
\item if $C_a > C_t$, the waiting function should wake as soon as possible, i.e., $C_a + D_i + D_m$, since the timeout is already satisfied. [Note: This specification \textbf{can} result in the total duration of the wait decreasing when measured against a steady clock. \textit{end note}]
\item if $C_a \leq C_t$, the waiting function should not time out until \texttt{Clock::now()} returns a time $C_n \geq C_t$, i.e., waking at $C_t + D_i + D_m$. [Note: When the clock is adjusted backwards, this specification \textbf{can} result in the total duration of the wait increasing when measured against a steady clock. When the clock is adjusted forwards, this specification \textbf{can} result in the total duration of the wait decreasing when measured against a steady clock. \textit{end note}]
\end{enumerate}

An implementation \textbf{shall} return from such a timeout at any point from the time specified above to the time it would return from a steady-clock relative timeout on the difference between $C_t$ and the time point of the call to the _until function. [Note: Implementations should decrease the duration of the wait when the clock is adjusted forwards. \textit{end note}]

5 [Note: If the clock is not synchronized with a steady clock, e.g., a CPU time clock, these timeouts might not provide useful functionality. \textit{end note}]

6 The resolution of timing provided by an implementation depends on both operating system and hardware. The finest resolution provided by an implementation is called the \textit{native resolution}.

7 Implementation-provided clocks that are used for these functions \textbf{shall} meet the Cpp17TrivialClock requirements (??).

8 A function that takes an argument which specifies a timeout will throw if, during its execution, a clock, time point, or time duration throws an exception. Such exceptions are referred to as \textit{timeout-related exceptions}. [Note: Instantiations of clock, time point and duration types supplied by the implementation as specified in ?? \textbf{do} not throw exceptions. \textit{end note}]

32.2.5 Requirements for Cpp17Lockable types [thread.req.lockable]

32.2.5.1 In general [thread.req.lockable.general]

An execution \textit{agent} is an entity such as a thread that \textbf{can} perform work in parallel with other execution agents. [Note: Implementations or users \textbf{can} introduce other kinds of agents such as processes or thread-pool tasks. \textit{end note}] The calling \textit{agent} is determined by context, e.g., the calling thread that contains the call, and so on.

2 [Note: Some lockable objects are “agent oblivious” in that they work for any execution agent model because they do not determine or store the agent’s ID (e.g., an ordinary spin lock). \textit{end note}]

The standard library templates unique_lock (32.5.4.3), shared_lock (32.5.4.4), scoped_lock (32.5.4.2), lock_guard (32.5.4.1), lock, try_lock (32.5.5), and condition_variable_any (32.6.4) all operate on user-supplied lockable objects. The Cpp17BasicLockable requirements, the Cpp17Lockable requirements, and the Cpp17TimedLockable requirements list the requirements imposed by these library types in order to acquire or release ownership of a lock by a given execution agent. [Note: The nature of any lock ownership and any synchronization it \textbf{entails} are not part of these requirements. \textit{end note}]

32.2.5.2 Cpp17BasicLockable requirements [thread.req.lockable.basic]

1 A \textit{type L} meets the Cpp17BasicLockable requirements if the following expressions are well-formed and have the specified semantics (m denotes a value of type L).

\footnote{327} All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.
m.lock()

Effects: Blocks until a lock can be acquired for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

m.unlock()

Requires—Expects: The current execution agent shall hold a lock on m.

Effects: Releases a lock on m held by the current execution agent.

Throws: Nothing.

32.2.5.3 Cpp17Lockable requirements

A type \( L \) meets the Cpp17Lockable requirements if it meets the Cpp17BasicLockable requirements and the following expressions are well-formed and have the specified semantics (\( m \) denotes a value of type \( L \)).

m.try_lock()

Effects: Attempts to acquire a lock for the current execution agent without blocking. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

Return type: bool.

Returns: true if the lock was acquired, false otherwise.

32.2.5.4 Cpp17TimedLockable requirements

A type \( L \) meets the Cpp17TimedLockable requirements if it meets the Cpp17Lockable requirements and the following expressions are well-formed and have the specified semantics (\( m \) denotes a value of type \( L \), \( rel\_time \) denotes a value of an instantiation of \( \text{duration} \) (??), and \( abs\_time \) denotes a value of an instantiation of \( \text{time\_point} \) (??)).

m.try_lock_for(rel_time)

Effects: Attempts to acquire a lock for the current execution agent within the relative timeout (32.2.4) specified by \( rel\_time \). The function shall not return within the timeout specified by \( rel\_time \) unless it has obtained a lock on m for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

Return type: bool.

Returns: true if the lock was acquired, false otherwise.

m.try_lock_until(abs_time)

Effects: Attempts to acquire a lock for the current execution agent before the absolute timeout (32.2.4) specified by \( abs\_time \). The function shall not return before the timeout specified by \( abs\_time \) unless it has obtained a lock on m for the current execution agent. If an exception is thrown then a lock shall not have been acquired for the current execution agent.

Return type: bool.

Returns: true if the lock was acquired, false otherwise.

32.3 Stop tokens

32.3.1 Introduction

This clause describes components that can be used to asynchronously request that an operation stops execution in a timely manner, typically because the result is no longer required. Such a request is called a stop request.

Any \( \text{stop\_source} \), \( \text{stop\_token} \), and \( \text{stop\_callback} \) implement semantics of shared ownership of a stop state. Any \( \text{stop\_source} \), \( \text{stop\_token} \), or \( \text{stop\_callback} \) that shares ownership of the same stop state is an associated \( \text{stop\_source} \), \( \text{stop\_token} \), or \( \text{stop\_callback} \), respectively. The last remaining owner of the stop state automatically releases the resources associated with the stop state.

A stop_token can be passed to an operation which can either

\( (3.1) \) — actively poll the token to check if there has been a stop request, or

\( (3.2) \) — register a callback using the \( \text{stop\_callback} \) class template which will be called in the event that a stop request is made.
A stop request made via a \texttt{stop_source} will be visible to all associated \texttt{stop_token} and \texttt{stop_source} objects. Once a stop request has been made it cannot be withdrawn (a subsequent stop request has no effect).

4 Callbacks registered via a \texttt{stop_callback} object are called when a stop request is first made by any associated \texttt{stop_source} object.

5 Calls to the functions \texttt{request_stop}, \texttt{stop_requested}, and \texttt{stop_possible} do not introduce data races. A call to \texttt{request_stop} that returns \texttt{true} synchronizes with a call to \texttt{stop_requested} on an associated \texttt{stop_token} or \texttt{stop_source} object that returns \texttt{true}. Registration of a callback synchronizes with the invocation of that callback.

### 32.3.2 Header \texttt{<stop_token>} synopsis

```cpp
namespace std {
    // 32.3.3, class stop_token
    class stop_token;

    // 32.3.4, class stop_source
    class stop_source;

    // no-shared-stop-state indicator
    struct nostopstate_t {
        explicit nostopstate_t() = default;
    };
    inline constexpr nostopstate_t nostopstate{};

    // 32.3.5, class stop_callback
    template<class Callback>
    class stop_callback;
}
```

### 32.3.3 Class \texttt{stop_token}

The class \texttt{stop_token} provides an interface for querying whether a stop request has been made (\texttt{stop_requested}) or can ever be made (\texttt{stop_possible}) using an associated \texttt{stop_source} object (32.3.4). A \texttt{stop_token} can also be passed to a \texttt{stop_callback} (32.3.5) constructor to register a callback to be called when a stop request has been made from an associated \texttt{stop_source}.

```cpp
namespace std {
    class stop_token {
        public:
            // 32.3.3.1, constructors, copy, and assignment
            stop_token() noexcept;
            stop_token(const stop_token&) noexcept;
            stop_token(stop_token&&) noexcept;
            stop_token& operator=(const stop_token&) noexcept;
            stop_token& operator=(stop_token&&) noexcept;
            ~stop_token();
            void swap(stop_token&) noexcept;

            // 32.3.3.2, stop handling
            [[nodiscard]] bool stop_requested() const noexcept;
            [[nodiscard]] bool stop_possible() const noexcept;

            [[nodiscard]] friend bool operator==(const stop_token& lhs, const stop_token& rhs) noexcept;
            [[nodiscard]] friend bool operator!=(const stop_token& lhs, const stop_token& rhs) noexcept;
            friend void swap(stop_token& lhs, stop_token& rhs) noexcept;
    }
}
```

#### 32.3.3.1 Constructors, copy, and assignment

```cpp
stop_token() noexcept;
```

\emph{Ensures:} \texttt{stop_possible()} is \texttt{false} and \texttt{stop_requested()} is \texttt{false}. \emph{Note:} Because the created \texttt{stop_token} object can never receive a stop request, no resources are allocated for a stop state. — end
stop_token(const stop_token& rhs) noexcept;

Ensures: *this == rhs is true. [Note: *this and rhs share the ownership of the same stop state, if any. — end note]

stop_token(stop_token&& rhs) noexcept;

Ensures: *this contains the value of rhs prior to the start of construction and rhs.stop_possible() is false.

~stop_token();

Effects: Releases ownership of the stop state, if any.

stop_token& operator=(const stop_token& rhs) noexcept;

Effects: Equivalent to: stop_token(rhs).swap(*this).

Returns: *this.

stop_token& operator=(stop_token&& rhs) noexcept;

Effects: Equivalent to: stop_token(std::move(rhs)).swap(*this).

Returns: *this.

void swap(stop_token& rhs) noexcept;

Effects: Exchanges the values of *this and rhs.

32.3.3.2 Members

[[nodiscard]] bool stop_requested() const noexcept;

Returns: true if *this has ownership of a stop state that has received a stop request; otherwise, false.

[[nodiscard]] bool stop_possible() const noexcept;

Returns: false if:

(2.1) *this does not have ownership of a stop state, or

(2.2) a stop request was not made and there are no associated stop_source objects; otherwise, true.

32.3.3.3 Comparisons

[[nodiscard]] bool operator==(const stop_token& lhs, const stop_token& rhs) noexcept;

Returns: true if lhs and rhs have ownership of the same stop state or if both lhs and rhs do not have ownership of a stop state; otherwise false.

[[nodiscard]] bool operator!=(const stop_token& lhs, const stop_token& rhs) noexcept;

Returns: !(lhs==rhs).

32.3.3.4 Specialized algorithms

friend void swap(stop_token& x, stop_token& y) noexcept;

Effects: Equivalent to: x.swap(y).

32.3.4 Class stop_source

The class stop_source implements the semantics of making a stop request. A stop request made on a stop_source object is visible to all associated stop_source and stop_token (32.3.3) objects. Once a stop request has been made it cannot be withdrawn (a subsequent stop request has no effect).

namespace std {
    // no-shared-stop-state indicator
    struct nostopstate_t {
        explicit nostopstate_t() = default;
    };

    § 32.3.4
inline constexpr nostopstate_t nostopstate{};

class stop_source {
public:
   // 32.3.4.1, constructors, copy, and assignment
   stop_source();
   explicit stop_source(nostopstate_t) noexcept;
   stop_source(const stop_source&) noexcept;
   stop_source(stop_source&&) noexcept;
   stop_source& operator=(const stop_source&) noexcept;
   stop_source& operator=(stop_source&&) noexcept;
   ~stop_source();
   void swap(stop_source&) noexcept;

   // 32.3.4.2, stop handling
   [[nodiscard]] stop_token get_token() const noexcept;
   [[nodiscard]] bool stop_possible() const noexcept;
   [[nodiscard]] bool stop_requested() const noexcept;
   bool request_stop() noexcept;
   [[nodiscard]] friend bool operator==(const stop_source& lhs, const stop_source& rhs) noexcept;
   [[nodiscard]] friend bool operator!=(const stop_source& lhs, const stop_source& rhs) noexcept;
   friend void swap(stop_source& lhs, stop_source& rhs) noexcept;
};

32.3.4.1 Constructors, copy, and assignment

stop_source();
1
   Effects: Initialises *this to have ownership of a new stop state.
2
   Ensures: stop_possible() is true and stop_requested() is false.
3
   Throws: bad_alloc if memory could not be allocated for the stop state.

explicit stop_source(nostopstate_t) noexcept;
4
   Ensures: stop_possible() is false and stop_requested() is false. [Note: No resources are allocated for the state. — end note]

stop_source(const stop_source& rhs) noexcept;
5
   Ensures: *this == rhs is true. [Note: *this and rhs share the ownership of the same stop state, if any. — end note]

stop_source(stop_source&& rhs) noexcept;
6
   Ensures: *this contains the value of rhs prior to the start of construction and rhs.stop_possible() is false.

~stop_source();
7
   Effects: Releases ownership of the stop state, if any.

stop_source& operator=(const stop_source& rhs) noexcept;
8
   Effects: Equivalent to: stop_source(rhs).swap(*this).
9
   Returns: *this.

stop_source& operator=(stop_source&& rhs) noexcept;
10
   Effects: Equivalent to: stop_source(std::move(rhs)).swap(*this).
11
   Returns: *this.

§ 32.3.4.1
void swap(stop_source& rhs) noexcept;

Effects: Exchanges the values of *this and rhs.

32.3.4.2 Members

[[nodiscard]] stop_token get_token() const noexcept;

Returns: stop_token() if stop_possible() is false; otherwise a new associated stop_token object.

[[nodiscard]] bool stop_possible() const noexcept;

Returns: true if *this has ownership of a stop state; otherwise, false.

[[nodiscard]] bool stop_requested() const noexcept;

Returns: true if *this has ownership of a stop state that has received a stop request; otherwise, false.

bool request_stop() noexcept;

Effects: If *this does not have ownership of a stop state, returns false. Otherwise, atomically determines whether the owned stop state has received a stop request, and if not, makes a stop request. The determination and making of the stop request are an atomic read-modify-write operation (??). If the request was made, the callbacks registered by associated stop_callback objects are synchronously called. If an invocation of a callback exits via an exception then terminate is called (??). [Note: A stop request includes notifying all condition variables of type condition_variable_any temporarily registered during an interruptible wait (32.6.4.2). — end note] Ensures: stop_possible() is false or stop_requested() is true.

Returns: true if this call made a stop request; otherwise false.

32.3.4.3 Comparisons

[[nodiscard]] bool operator==(const stop_source& lhs, const stop_source& rhs) noexcept;

Returns: true if lhs and rhs have ownership of the same stop state or if both lhs and rhs do not have ownership of a stop state; otherwise false.

[[nodiscard]] bool operator!=(const stop_source& lhs, const stop_source& rhs) noexcept;

Returns: !(lhs==rhs).

32.3.4.4 Specialized algorithms

friend void swap(stop_source& x, stop_source& y) noexcept;

Effects: Equivalent to: x.swap(y).

32.3.5 Class template stop_callback

namespace std {
    template<class Callback>
    class stop_callback {
    public:
        using callback_type = Callback;

        // 32.3.5.1, constructors and destructor
        template<class C>
        explicit stop_callback(const stop_token& st, C&& cb)
            noexcept(is_nothrow_constructible_v<Callback, C>);
        template<class C>
        explicit stop_callback(stop_token&& st, C&& cb)
            noexcept(is_nothrow_constructible_v<Callback, C>);
        ~stop_callback();

        stop_callback(const stop_callback&) = delete;
        stop_callback(stop_callback&&) = delete;
        stop_callback& operator=(const stop_callback&) = delete;
        stop_callback& operator=(stop_callback&&) = delete;
    }
}

§ 32.3.5
private:
    Callback callback; // exposition only
};

template<class Callback>
stop_callback(stop_token, Callback) -> stop_callback<Callback>;
}

Mandates: stop_callback is instantiated with an argument for the template parameter Callback that satisfies both invocable and destructible.

Expects: stop_callback is instantiated with an argument for the template parameter Callback that models both invocable and destructible.

32.3.5.1 Constructors and destructor

template<class C>
explicit stop_callback(const stop_token& st, C&& cb)
    noexcept(is_nothrow_constructible_v<Callback, C>);

template<class C>
explicit stop_callback(stop_token&& st, C&& cb)
    noexcept(is_nothrow_constructible_v<Callback, C>);

Constraints: Callback and C satisfy constructible_from<Callback, C>.

Expects: Callback and C model constructible_from<Callback, C>.

Effects: Initializes callback with \texttt{std::forward\langle C\rangle}(cb). If \texttt{st.stop_requested()} is true, then \texttt{std::forward\langle Callback\rangle}(callback)() is evaluated in the current thread before the constructor returns. Otherwise, if \texttt{st} has ownership of a stop state, acquires shared ownership of that stop state and registers the callback with that stop state such that \texttt{std::forward\langle Callback\rangle}(callback)() is evaluated by the first call to \texttt{request\_stop()} on an associated \texttt{stop\_source}.

Remarks: If evaluating \texttt{std::forward\langle Callback\rangle}(callback)() exits via an exception, then terminate is called (??).

Throws: Any exception thrown by the initialization of callback.

~stop_callback();

Effects: Unregisters the callback from the owned stop state, if any. The destructor does not block waiting for the execution of another callback registered by an associated \texttt{stop\_callback}. If \texttt{callback} is concurrently executing on another thread, then the return from the invocation of \texttt{callback} strongly happens before (??) \texttt{callback} is destroyed. If \texttt{callback} is executing on the current thread, then the destructor does not block (??) waiting for the return from the invocation of \texttt{callback}. Releases ownership of the stop state, if any.

32.4 Threads

32.4 describes components that can be used to create and manage threads. \textit{[Note: These threads are intended to map one-to-one with operating system threads. — end note]}

32.4.1 Header <thread> synopsis

namespace std {
    class thread;

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

    // 32.4.3 class jthread
    class jthread;

    namespace this_thread {
        thread::id get_id() noexcept;

        void yield() noexcept;
        template<class Clock, class Duration>
        void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    }
}
32.4.2 Class thread

The class thread provides a mechanism to create a new thread of execution, to join with a thread (i.e., wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A thread object uniquely represents a particular thread of execution. That representation cannot be transferred to other thread objects in such a way that no two thread objects simultaneously represent the same thread of execution. A thread of execution is detached when no thread object represents that thread. Objects of class thread can be in a state that does not represent a thread of execution. [Note: A thread object does not represent a thread of execution after default construction, after being moved from, or after a successful call to detach or join. — end note]

```cpp
namespace std {
    class thread {
    public:
        // types
        class id;
        using native_handle_type = implementation-defined; // see 32.2.3

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

        // members
        void swap(thread&) noexcept;
        bool joinable() const noexcept;
        void join();
        void detach();
        id get_id() const noexcept;
        native_handle_type native_handle(); // see 32.2.3

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

32.4.2.1 Class thread::id

```cpp
namespace std {
    class thread::id {
    public:
        id() noexcept;
    
        bool operator==(thread::id x, thread::id y) noexcept;
        strong_ordering operator<=>(thread::id x, thread::id y) noexcept;

        template<class charT, class traits>
        basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& out, thread::id id);

        // hash support
        template<class T> struct hash;
        template<> struct hash<thread::id>;
    }
}
```

§ 32.4.2.1 9
An object of type `thread::id` provides a unique identifier for each thread of execution and a single distinct value for all `thread` objects that do not represent a thread of execution (32.4.2). Each thread of execution has an associated `thread::id` object that is not equal to the `thread::id` object of any other thread of execution and that is not equal to the `thread::id` object of any `thread` object that does not represent threads of execution.

`thread::id` is a trivially copyable class (??). The library may reuse the value of a `thread::id` of a terminated thread that can no longer be joined.

[Note: Relational operators allow `thread::id` objects to be used as keys in associative containers. — end note]

```cpp
id() noexcept;
```

**Effects:** Constructs an object of type `id`.

**Ensures:** The constructed object does not represent a thread of execution.

```cpp
bool operator==(thread::id x, thread::id y) noexcept;
```

**Returns:** `true` only if `x` and `y` represent the same thread of execution or neither `x` nor `y` represents a thread of execution.

```cpp
strong_ordering operator<>(thread::id x, thread::id y) noexcept;
```

**Let** `P(x,y)` be an unspecified total ordering over `thread::id` as described in ??.

**Returns:** `strong_ordering::less` if `P(x,y)` is `true`. Otherwise, `strong_ordering::greater` if `P(y,x)` is `true`. Otherwise, `strong_ordering::equal`.

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<< (basic_ostream<charT, traits>& out, thread::id id);
```

**Effects:** Inserts an unspecified text representation of `id` into `out`. For two objects of type `thread::id` `x` and `y`, if `x == y` the `thread::id` objects have the same text representation and if `x != y` the `thread::id` objects have distinct text representations.

**Returns:** `out`.

```cpp
template<> struct hash<thread::id>;
```

The specialization is enabled (??).

### 32.4.2.2 Constructors

**thread() noexcept;**

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

**Ensures:** `get_id() == id()`.

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

**Mandates:** The following are all true:

1. `is_move_constructible<decay_t<F>>`,
2. `(is_move_constructible<decay_t<Args>> && ...)`, and
3. `is_invocable_v<decay_t<F>, decay_t<Arg>...>`.

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

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

**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.

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.

thread(thread&& x) noexcept;

Effects: Constructs an object of type thread 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.

32.4.2.3 Destructor
~thread();

If joinable(), calls terminate(). Otherwise, has no effects. [Note: Either implicitly detaching or joining a joinable() thread in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is thrown. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. — end note]

32.4.2.4 Assignment
thread& operator=(thread&& x) noexcept;

Effects: If joinable(), calls terminate(). Otherwise, 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.

Returns: *this.

32.4.2.5 Members
void swap(thread& x) noexcept;

Effects: Swaps the state of *this and x.

bool joinable() const noexcept;

Returns: get_id() != id().

void join();

Effects: Blocks until the thread represented by *this has completed.

Synchronization: The completion of the thread represented by *this synchronizes with (??) the corresponding successful join() return. [Note: Operations on *this are not synchronized. — end note]

Ensures: The thread represented by *this has completed. get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:
— resource_deadlock_would_occur — if deadlock is detected or get_id() == this_thread::get_id().
— no_such_process — if the thread is not valid.
— invalid_argument — if the thread is not joinable.
```cpp
void detach();

Effects: The thread represented by *this continues execution without the calling thread blocking. When detach() returns, *this no longer represents the possibly continuing thread of execution. When the thread previously represented by *this ends execution, the implementation shall release any owned resources.

Ensures: get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:

— no_such_process — if the thread is not valid.
— invalid_argument — if the thread is not joinable.

id get_id() const noexcept;

Returns: A default constructed id object if *this does not represent a thread, otherwise this_thread::get_id() for the thread of execution represented by *this.

32.4.2.6 Static members

unsigned hardware_concurrency() noexcept;

Returns: The number of hardware thread contexts. [Note: This value should only be considered to be a hint. — end note] If this value is not computable or well-defined, an implementation should return 0.

32.4.2.7 Specialized algorithms

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

Effects: As if by x.swap(y).

32.4.3 Class jthread

The class jthread provides a mechanism to create a new thread of execution. The functionality is the same as for class thread (32.4.2) with the additional abilities to provide a stop_token (32.3) to the new thread of execution, make stop requests, and automatically join.

```
32.4.3.1 Constructors, move, and assignment

jthread() noexcept;

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

Ensures: get_id() == id() is true and ssouree.stop_possible() is false.

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

Requires: F and each T in Args meet the Cpp17MoveConstructible requirements. Either

INVoke(decay-copy(std::forward<F>(f)), get_stop_token(),
   decay-copy(std::forward<Args>(args))...) is a valid expression or

INVoke(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...) is a valid expression. Mandates: The following are all true:

(3.1) — is_move_constructible<decay_t<F>>,
(3.2) — (is_move_constructible<decay_t<T>>& ...), and
(3.3) — is_invocable_v<decay_t<F>, decay_t<T>...>.

Constraints: remove_cvref_t<F> is not the same type as jthread.

Effects: Initializes ssouree and constructs an object of type jthread. The new thread of execution executes

INVoke(decay-copy(std::forward<F>(f)), get_stop_token(),
   decay-copy(std::forward<Args>(args))...) if that expression is well-formed, otherwise

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 INVoke expression exits via an exception, terminate is called.

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() is true and ssouree.stop_possible() is true and *this represents the newly started thread. [Note: The calling thread can make a stop request only once, because it cannot replace this stop token. — end note]

Throws: system_error if unable to start the new thread.

Error conditions:

(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. ssouree has the value of x.ssouree prior to the start of construction and x.ssouree.stop_possible() is false.
~jthread();

Effects: If joinable() is true, calls request_stop() and then join(). [Note: Operations on *this are not synchronized. — end note]

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

Effects: If joinable() is true, calls request_stop() and then 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. ssoure has the value of x.ssource prior to the assignment and x.ssource.stop_possible() is false.

Returns: *this.

32.4.3.2 Members

void swap(jthread& x) noexcept;

Effects: Exchanges the values of *this and x.

[[nodiscard]] bool joinable() const noexcept;

Returns: get_id() != id().

void join();

Effects: Blocks until the thread represented by *this has completed.

Synchronization: The completion of the thread represented by *this synchronizes with (??) the corresponding successful join() return. [Note: Operations on *this are not synchronized. — end note]

Ensures: The thread represented by *this has completed. get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:

— resource_deadlock_would_occur — if deadlock is detected or get_id() == this_thread::get_id().

— no_such_process — if the thread is not valid.

— invalid_argument — if the thread is not joinable.

void detach();

Effects: The thread represented by *this continues execution without the calling thread blocking. When detach() returns, *this no longer represents the possibly continuing thread of execution. When the thread previously represented by *this ends execution, the implementation shall releases any owned resources.

Ensures: get_id() == id().

Throws: system_error when an exception is required (32.2.2).

Error conditions:

— no_such_process — if the thread is not valid.

— invalid_argument — if the thread is not joinable.

id get_id() const noexcept;

Returns: A default constructed id object if *this does not represent a thread, otherwise this_thread::get_id() for the thread of execution represented by *this.

32.4.3.3 Stop token handling

[[nodiscard]] stop_source get_stop_source() noexcept;

Effects: Equivalent to: return ssoure;
[[nodiscard]] stop_token get_stop_token() const noexcept;

Effects: Equivalent to: return ssource.get_token();

bool request_stop() noexcept;

Effects: Equivalent to: return ssource.request_stop();

32.4.3.4 Specialized algorithms

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

Effects: Equivalent to: x.swap(y).

32.4.3.5 Static members

unsigned hardware_concurrency() noexcept;

Returns: thread::hardware_concurrency().

32.4.4 Namespace this_thread

namespace std::this_thread {

thread::id get_id() noexcept;

Effects: Equivalent to: thread::hardware_concurrency().

void yield() noexcept;

Effects: Offers the implementation the opportunity to reschedule.

Synchronization: None.

template<class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

Effects: Blocks the calling thread for the absolute timeout (32.2.4) specified by abs_time.

Synchronization: None.

Throws: Timeout-related exceptions (32.2.4).

template<class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& rel_time);

Effects: Blocks the calling thread for the relative timeout (32.2.4) specified by rel_time.

Synchronization: None.

Throws: Timeout-related exceptions (32.2.4).

32.5 Mutual exclusion

This subclause provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (??).

32.5.1 Header <mutex> synopsis

namespace std {

class mutex;

class recursive_mutex;

class timed_mutex;

class recursive_timed_mutex;

§ 32.5.1
struct defer_lock_t { explicit defer_lock_t() = default; };
struct try_to_lock_t { explicit try_to_lock_t() = default; };
struct adopt_lock_t { explicit adopt_lock_t() = default; };

inline constexpr defer_lock_t defer_lock { };
inline constexpr try_to_lock_t try_to_lock { };
inline constexpr adopt_lock_t adopt_lock { };

template<class Mutex> class lock_guard;
template<class... MutexTypes> class scoped_lock;
template<class Mutex> class unique_lock;
template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

struct once_flag;
template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);

32.5.2 Header <shared_mutex> synopsis

namespace std {
    class shared_mutex;
    class shared_timed_mutex;
template<class Mutex> class shared_lock;
template<class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
}

32.5.3 Mutex requirements

32.5.3.1 In general

A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (32.2.5). An execution agent owns a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes can be either recursive or non-recursive, and can grant simultaneous ownership to one or many execution agents. Both recursive and non-recursive mutexes are supplied.

32.5.3.2 Mutex types

The mutex types are the standard library types mutex, recursive_mutex, timed_mutex, recursive_timed_mutex, shared_mutex, and shared_timed_mutex. They shall meet the requirements set out in this subclause. In this description, m denotes an object of a mutex type.

The mutex types shall meet the Cpp17Lockable requirements (32.2.5.3).

The mutex types shall be meet Cpp17DefaultConstructible and Cpp17Destructible. If initialization of an object of a mutex type fails, an exception of type system_error shall be thrown. The mutex types shall not be neither copyable nor movable.

The error conditions for error codes, if any, reported by member functions of the mutex types shall be as follows:

(4.1) — resource_unavailable_try_again — if any native handle type manipulated is not available.
(4.2) — operation_not_permitted — if the thread does not have the privilege to perform the operation.
(4.3) — invalid_argument — if any native handle type manipulated as part of mutex construction is incorrect.

The implementation provides lock and unlock operations, as described below. For purposes of determining the existence of a data race, these behave as atomic operations (??). The lock and unlock operations on a single mutex shall appear to occur in a single total order. [Note: This can be viewed as the modification order (??) of the mutex. — end note] [Note: Construction and destruction of an object of a mutex type}
need not be thread-safe; other synchronization should be used to ensure that mutex objects are initialized and visible to other threads. — end note]

The expression \texttt{m.lock()} shall be well-formed and have the following semantics:

- **Requires—Expects:** If \( m \) is of type \texttt{mutex}, \texttt{timed_mutex}, \texttt{shared_mutex}, or \texttt{shared_timed_mutex}, the calling thread does not own the mutex.
- **Effects:** Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.
- **Ensures:** The calling thread owns the mutex.
- **Return type:** \texttt{void}.
- **Synchronization:** Prior \texttt{unlock()} operations on the same object shall synchronize with (??) this operation.

**Throws:** \texttt{system\_error} when an exception is required (32.2.2).

**Error conditions:**

- \texttt{operation\_not\_permitted} — if the thread does not have the privilege to perform the operation.
- \texttt{resource\_deadlock\_would\_occur} — if the implementation detects that a deadlock would occur.

The expression \texttt{m.try\_lock()} shall be well-formed and have the following semantics:

- **Requires—Expects:** If \( m \) is of type \texttt{mutex}, \texttt{timed_mutex}, \texttt{shared_mutex}, or \texttt{shared_timed_mutex}, the calling thread does not own the mutex.
- **Effects:** Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and \texttt{try\_lock()} immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. \[Note: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (??). — end note\] An implementation should ensure that \texttt{try\_lock()} does not consistently return \texttt{false} in the absence of contending mutex acquisitions.

- **Return type:** \texttt{bool}.
- **Returns:** \texttt{true} if ownership of the mutex was obtained for the calling thread, otherwise \texttt{false}.
- **Synchronization:** If \texttt{try\_lock()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (??) this operation. \[Note: Since \texttt{lock()} does not synchronize with a failed subsequent \texttt{try\_lock()}, the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures. — end note\]

- **Throws:** Nothing.

The expression \texttt{m.unlock()} shall be well-formed and have the following semantics:

- **Requires—Expects:** The calling thread shall own the mutex.
- **Effects:** Releases the calling thread’s ownership of the mutex.
- **Return type:** \texttt{void}.
- **Synchronization:** This operation synchronizes with (??) subsequent lock operations that obtain ownership on the same object.

- **Throws:** Nothing.

### 32.5.3.2.1 Class \texttt{mutex}

```cpp
namespace std { 
    class mutex { 
        public: 
            constexpr mutex() noexcept; 
            ~mutex(); 
            mutex(const mutex&) = delete; 
            mutex& operator=(const mutex&) = delete; 
            void lock(); 
            bool try_lock(); 
            void unlock();
        }
    }
```

§ 32.5.3.2.1
The class `mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

[Note: After a thread \( A \) has called `unlock()`, releasing a mutex, it is possible for another thread \( B \) to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread \( A \) appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread \( A \) doesn’t access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. —end note]

The class `mutex` shall meet all of the mutex requirements (32.5.3). It shall be a standard-layout class (??).

A thread that owns a `mutex` object may acquire additional levels of ownership by calling `lock()` or `try_lock()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_mutex` object, additional calls to `try_lock()` shall fail, and additional calls to `lock()` shall throw an exception of type `system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()` and `try_lock()`. Only when all levels of ownership have been released may ownership be acquired by another thread.

The behavior of a program is undefined if:

1. it destroys a `mutex` object owned by any thread or
2. a thread terminates while owning a `mutex` object.

32.5.3.3 Timed mutex types

The `timed mutex types` are the standard library types `timed_mutex`, `recursive_timed_mutex`, and `shared_timed_mutex`. They shall meet the requirements set out below. In this description, \( m \) denotes an object of
a mutex type, \texttt{rel\_time} denotes an object of an instantiation of \texttt{duration} (??), and \texttt{abs\_time} denotes an object of an instantiation of \texttt{time\_point} (??).

2 The timed mutex types \texttt{timed\_mutex} and \texttt{shared\_timed\_mutex} shall meet the \texttt{Cpp17TimedLockable} requirements (32.2.5.4).

3 The expression \texttt{m.try\_lock\_for} (\texttt{rel\_time}) shall be well-formed and have the following semantics:

\begin{itemize}
\item \textbf{Requires:} If \texttt{m} is of type \texttt{timed\_mutex} or \texttt{shared\_timed\_mutex}, the calling thread does not own the mutex.
\item \textbf{Effects:} The function attempts to obtain ownership of the mutex within the relative timeout (32.2.4) specified by \texttt{rel\_time}. If the time specified by \texttt{rel\_time} is less than or equal to \texttt{rel\_time.zero()}, the function attempts to obtain ownership without blocking (as if by calling \texttt{try\_lock()}). The function shall return within the timeout specified by \texttt{rel\_time} only if it has obtained ownership of the mutex object. [\textit{Note:} As with \texttt{try\_lock()}, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. —end note]
\item \textbf{Return type:} \texttt{bool}.
\item \textbf{Returns:} \texttt{true} if ownership was obtained, otherwise \texttt{false}.
\item \textbf{Synchronization:} If \texttt{try\_lock\_for()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (??) this operation.
\item \textbf{Throws:} Timeout-related exceptions (32.2.4).
\end{itemize}

4 The expression \texttt{m.try\_lock\_until} (\texttt{abs\_time}) shall be well-formed and have the following semantics:

\begin{itemize}
\item \textbf{Requires:} If \texttt{m} is of type \texttt{timed\_mutex} or \texttt{shared\_timed\_mutex}, the calling thread does not own the mutex.
\item \textbf{Effects:} The function attempts to obtain ownership of the mutex. If \texttt{abs\_time} has already passed, the function attempts to obtain ownership without blocking (as if by calling \texttt{try\_lock()}). The function shall return before the absolute timeout (32.2.4) specified by \texttt{abs\_time} only if it has obtained ownership of the mutex object. [\textit{Note:} As with \texttt{try\_lock()}, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. —end note]
\item \textbf{Return type:} \texttt{bool}.
\item \textbf{Returns:} \texttt{true} if ownership was obtained, otherwise \texttt{false}.
\item \textbf{Synchronization:} If \texttt{try\_lock\_until()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (??) this operation.
\item \textbf{Throws:} Timeout-related exceptions (32.2.4).
\end{itemize}

5 The class \texttt{timed\_mutex} provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a \texttt{timed\_mutex} object, attempts by another thread to acquire ownership of that object will fail (for
try_lock() or block (for lock(), try_lock_for(), and try_lock_until()) until the owning thread has released ownership with a call to unlock() or the call to try_lock_for() or try_lock_until() times out (having failed to obtain ownership).

2 The class `timed_mutex` shall meet all of the timed mutex requirements (32.5.3.3). It shall be a standard-layout class (??).

3 The behavior of a program is undefined if:

(3.1) — it destroys a `timed_mutex` object owned by any thread,

(3.2) — a thread that owns a `timed_mutex` object calls lock(), try_lock(), try_lock_for(), or try_lock_until() on that object, or

(3.3) — a thread terminates while owning a `timed_mutex` object.

32.5.3.3.2 Class `recursive_timed_mutex` [thread.timedmutex.recursive]

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

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

            void lock(); // blocking
            bool try_lock() noexcept;

            template<class Rep, class Period>
                bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

            void unlock();

            using native_handle_type = implementation-defined; // see 32.2.3
            native_handle_type native_handle(); // see 32.2.3
    };
}

1 The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for try_lock()) or block (for lock(), try_lock_for(), and try_lock_until()) until the owning thread has completely released ownership or the call to try_lock_for() or try_lock_until() times out (having failed to obtain ownership).

2 The class `recursive_timed_mutex` shall meet all of the timed mutex requirements (32.5.3.3). It shall be a standard-layout class (??).

3 A thread that owns a `recursive_timed_mutex` object may acquire additional levels of ownership by calling lock(), try_lock(), try_lock_for(), or try_lock_until() on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_timed_mutex` object, additional calls to try_lock(), try_lock_for(), or try_lock_until() shall fail, and additional calls to lock() shall throw an exception of type system_error. A thread shall call unlock() once for each level of ownership acquired by calls to lock(), try_lock(), try_lock_for(), and try_lock_until(). Only when all levels of ownership have been released ownership of the object be acquired by another thread.

4 The behavior of a program is undefined if:

(4.1) — it destroys a `recursive_timed_mutex` object owned by any thread, or

(4.2) — a thread terminates while owning a `recursive_timed_mutex` object.

32.5.3.4 Shared mutex types [thread.sharedmutex.requirements]

1 The standard library types `shared_mutex` and `shared_timed_mutex` are shared mutex types. Shared mutex types shall meet the requirements of mutex types (32.5.3.2), and additionally shall meet the requirements set out below. In this description, m denotes an object of a shared mutex type.
In addition to the exclusive lock ownership mode specified in 32.5.3.2, shared mutex types provide a shared lock ownership mode. Multiple execution agents can simultaneously hold a shared lock ownership of a shared mutex type. But no execution agent shall hold a shared lock while another execution agent holds an exclusive lock on the same shared mutex type, and vice-versa. The maximum number of execution agents which can share a shared lock on a single shared mutex type is unspecified, but shall be at least 10000. If more than the maximum number of execution agents attempt to obtain a shared lock, the excess execution agents shall block until the number of shared locks are reduced below the maximum amount by other execution agents releasing their shared lock.

The expression \texttt{m.lock\_shared()} shall be well-formed and have the following semantics:

\begin{description}
  \item[Requires] The calling thread has no ownership of the mutex.
  \item[Expects] Blocks the calling thread until shared ownership of the mutex can be obtained for the calling thread. If an exception is thrown then a shared lock shall not have been acquired for the current thread.
  \item[Ensures] The calling thread has a shared lock on the mutex.
  \item[Return type] \texttt{void}.
  \item[Synchronization] Prior unlock() operations on the same object shall synchronize with (??) this operation.
  \item[Throws] \texttt{system\_error} when an exception is required (32.2.2).
\end{description}

\begin{itemize}
  \item \texttt{operation\_not\_permitted} \quad if the thread does not have the privilege to perform the operation.
  \item \texttt{resource\_deadlock\_would\_occur} \quad if the implementation detects that a deadlock would occur.
\end{itemize}

The expression \texttt{m.unlock\_shared()} shall be well-formed and have the following semantics:

\begin{description}
  \item[Requires] The calling thread shall hold a shared lock on the mutex.
  \item[Expects] Releases a shared lock on the mutex held by the calling thread.
  \item[Return type] \texttt{void}.
  \item[Synchronization] This operation synchronizes with (??) subsequent lock() operations that obtain ownership on the same object.
  \item[Throws] Nothing.
\end{description}

The expression \texttt{m.try\_lock\_shared()} shall be well-formed and have the following semantics:

\begin{description}
  \item[Requires] The calling thread has no ownership of the mutex.
  \item[Expects] Attempts to obtain shared ownership of the mutex for the calling thread without blocking. If shared ownership is not obtained, there is no effect and \texttt{try\_lock\_shared()} immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread.
  \item[Return type] \texttt{bool}.
  \item[Returns] \texttt{true} if the shared ownership lock was acquired, \texttt{false} otherwise.
  \item[Synchronization] If \texttt{try\_lock\_shared()} returns \texttt{true}, prior unlock() operations on the same object synchronize with (??) this operation.
  \item[Throws] Nothing.
\end{description}

\section{Class \texttt{shared\_mutex} [thread.sharedmutex.class]}

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

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

        // exclusive ownership
        void lock(); // blocking
bool try_lock();
void unlock();

// shared ownership
void lock_shared();  // blocking
bool try_lock_shared();
void unlock_shared();

using native_handle_type = implementation-defined;    // see 32.2.3
native_handle_type native_handle();  // see 32.2.3

1 The class shared_mutex provides a non-recursive mutex with shared ownership semantics.
2 The class shared_mutex shall meet all of the shared mutex requirements (32.5.3.4). It shall be a standard-layout class (??).
3 The behavior of a program is undefined if:
   - it destroys a shared_mutex object owned by any thread,
   - a thread attempts to recursively gain any ownership of a shared_mutex, or
   - a thread terminates while possessing any ownership of a shared_mutex.
4 shared_mutex may be a synonym for shared_timed_mutex.

### 32.5.3.5 Shared timed mutex types

The standard library type shared_timed_mutex is a shared timed mutex type. Shared timed mutex types shall meet the requirements of timed mutex types (32.5.3.3), shared mutex types (32.5.3.4), and additionally shall meet the requirements set out below. In this description, m denotes an object of a shared timed mutex type, rel_type denotes an object of an instantiation of duration (??), and abs_time denotes an object of an instantiation of time_point (??).

The expression m.try_lock_shared_for(rel_time) shall be well-formed and have the following semantics:

- **Requires:** The calling thread has no ownership of the mutex.
- **Effects:** Attempts to obtain shared lock ownership for the calling thread within the relative timeout (32.2.4) specified by rel_time. If the time specified by rel_time is less than or equal to rel_time.zero(), the function attempts to obtain ownership without blocking (as if by calling try_lock_shared()). The function shall return true within the timeout specified by rel_time only if it has obtained shared ownership of the mutex object. [Note: As with try_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note] If an exception is thrown then a shared lock shall not have been acquired for the current thread.
- **Return type:** bool.
- **Returns:** true if the shared lock was acquired, false otherwise.
- **Synchronization:** If try_lock_shared_for() returns true, prior unlock() operations on the same object synchronize with (??) this operation.
- **Throws:** Timeout-related exceptions (32.2.4).

The expression m.try_lock_shared_until(abs_time) shall be well-formed and have the following semantics:

- **Requires:** The calling thread has no ownership of the mutex.
- **Effects:** The function attempts to obtain shared ownership of the mutex. If abs_time has already passed, the function attempts to obtain shared ownership without blocking (as if by calling try_lock_shared()). The function shall return before the absolute timeout (32.2.4) specified by abs_time only if it has obtained shared ownership of the mutex object. [Note: As with try_lock(), there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note] If an exception is thrown then a shared lock shall not have been acquired for the current thread.
Return type: bool.

Returns: true if the shared lock was acquired, false otherwise.

Synchronization: If try_lock_shared_until() returns true, prior unlock() operations on the same object synchronize with this operation.

Throws: Timeout-related exceptions (32.2.4).

32.5.3.5.1 Class shared_timed_mutex

namespace std {
    class shared_timed_mutex {
        public:
            shared_timed_mutex();
            ~shared_timed_mutex();
            shared_timed_mutex(const shared_timed_mutex&) = delete;
            shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;

            // exclusive ownership
            void lock(); // blocking
            bool try_lock();
            template<class Rep, class Period>
                bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
            void unlock();

            // shared ownership
            void lock_shared(); // blocking
            bool try_lock_shared();
            template<class Rep, class Period>
                bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
            void unlock_shared();
        }
    }
}

The class shared_timed_mutex provides a non-recursive mutex with shared ownership semantics.

The class shared_timed_mutex shall meet all of the shared timed mutex requirements (32.5.3.5). It shall be a standard-layout class.

The behavior of a program is undefined if:
1. it destroys a shared_timed_mutex object owned by any thread,
2. a thread attempts to recursively gain any ownership of a shared_timed_mutex, or
3. a thread terminates while possessing any ownership of a shared_timed_mutex.

32.5.4 Locks

A lock is an object that holds a reference to a lockable object and unlock the lockable object during the lock’s destruction (such as when leaving block scope). An execution agent can use a lock to aid in managing ownership of a lockable object in an exception safe manner. A lock is said to own a lockable object if it is currently managing the ownership of that lockable object for an execution agent. A lock does not manage the lifetime of the lockable object it references. [Note: Locks are intended to ease the burden of unlocking the lockable object under both normal and exceptional circumstances. — end note]

Some lock constructors take tag types which describe what should be done with the lockable object during the lock’s construction.

namespace std {
    struct defer_lock_t { }; // do not acquire ownership of the mutex
    struct try_to_lock_t { }; // try to acquire ownership of the mutex
        // without blocking
    struct adopt_lock_t { }; // assume the calling thread has already
// obtained mutex ownership and manage it

inline constexpr defer_lock_t defer_lock { };  
inline constexpr try_to_lock_t try_to_lock { };  
inline constexpr adopt_lock_t adopt_lock { };

32.5.4.1 Class template lock_guard

namespace std {
    template<class Mutex>
    class lock_guard {
    public:
        using mutex_type = Mutex;

        explicit lock_guard(mutex_type& m);
        lock_guard(mutex_type& m, adopt_lock_t);
        ~lock_guard();

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

    private:
        mutex_type& pm;  // exposition only
    };
}

1 An object of type lock_guard controls the ownership of a lockable object within a scope. A lock_guard object maintains ownership of a lockable object throughout the lock_guard object’s lifetime (??). The behavior of a program is undefined if thelockable object referenced by pm does not exist for the entire lifetime of the lock_guard object. The supplied Mutex type shall meet the Cpp17BasicLockable requirements (32.2.5.2).

    explicit lock_guard(mutex_type& m);

2 Requires: If mutex_type is not a recursive mutex, the calling thread does not own the mutex m.

3 Effects: Initializes pm with m. Calls m.lock().

    lock_guard(mutex_type& m, adopt_lock_t);

4 Requires: The calling thread owns the mutex m.

5 Effects: Initializes pm with m.

6 Throws: Nothing.

    ~lock_guard();

7 Effects: As if by pm.unlock().

32.5.4.2 Class template scoped_lock

namespace std {
    template<class... MutexTypes>
    class scoped_lock {
    public:
        using mutex_type = Mutex;  // If MutexTypes... consists of the single type Mutex

        explicit scoped_lock(MutexTypes&&... m);
        explicit scoped_lock(adopt_lock_t, MutexTypes&&... m);
        ~scoped_lock();

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

    private:
        tuple<MutexTypes&&...> pm;  // exposition only

    };

§ 32.5.4.2
An object of type scoped_lock controls the ownership of lockable objects within a scope. A scoped_lock object maintains ownership of lockable objects throughout the scoped_lock object’s lifetime. The behavior of a program is undefined if the lockable objects referenced by pm do not exist for the entire lifetime of the scoped_lock object. When sizeof...(MutexTypes) is 1, the supplied Mutex type shall meet the Cpp17BasicLockable requirements (32.2.5.2). Otherwise, each of the mutex types shall meet the Cpp17Lockable requirements (32.2.5.3).

```
explicit scoped_lock(MutexTypes&... m);
```

```
Requires-Expects: If a MutexTypes type is not a recursive mutex, the calling thread does not own the corresponding mutex element of m.
```

```
Effects: Initializes pm with tie(m...). Then if sizeof...(MutexTypes) is 0, no effects. Otherwise if sizeof...(MutexTypes) is 1, then m.lock(). Otherwise, lock(m...).
```

```
explicit scoped_lock(adopt_lock_t, MutexTypes&... m);
```

```
Requires-Expects: The calling thread owns all the mutexes in m.
```

```
Effects: Initializes pm with tie(m...).
```

```
Throws: Nothing.
```

```
~scoped_lock();
```

```
Effects: For all i in [0, sizeof...(MutexTypes)), get<i>(pm).unlock().
```

### 32.5.4.3 Class template unique_lock

```
namespace std {

template<class Mutex>
class unique_lock {
    public:
        using mutex_type = Mutex;

        // 32.5.4.3.1, construct/copy/destroy
        unique_lock() noexcept;
        explicit unique_lock(mutex_type& m);
        unique_lock(mutex_type& m, defer_lock_t) noexcept;
        unique_lock(mutex_type& m, try_to_lock_t);
        unique_lock(mutex_type& m, adopt_lock_t);
        template<class Clock, class Duration>
            unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
        template<class Rep, class Period>
            unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
        ~unique_lock();

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

        unique_lock(unique_lock&& u) noexcept;
        unique_lock& operator=(unique_lock&& u);

        // 32.5.4.3.2, locking
        void lock();
        bool try_lock();
        template<class Rep, class Period>
            bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
        template<class Clock, class Duration>
            bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
        void unlock();

    
```

§ 32.5.4.3
void swap(unique_lock& u) noexcept;
mutex_type* release() noexcept;

bool owns_lock() const noexcept;
explicit operator bool () const noexcept;
mutex_type* mutex() const noexcept;

private:
mutex_type* pm;       // exposition only
bool owns;            // exposition only

};

template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

An object of type unique_lock controls the ownership of a lockable object within a scope. Ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another unique_lock object. Objects of type unique_lock are not copyable but are movable. The behavior of a program is undefined if the contained pointer pm is not null and the lockable object pointed to by pm does not exist for the entire remaining lifetime of the unique_lock object. The supplied Mutex type shall meet the Cpp17BasicLockable requirements (32.2.5.2).

[Note: unique_lock<Mutex> meets the Cpp17BasicLockable requirements. If Mutex meets the Cpp17Lockable requirements (32.2.5.3), unique_lock<Mutex> also meets the Cpp17Lockable requirements; if Mutex meets the Cpp17TimedLockable requirements (32.2.5.4), unique_lock<Mutex> also meets the Cpp17TimedLockable requirements. — end note]

32.5.4.3.1 Constructors, destructor, and assignment

unique_lock() noexcept;
Effects: Constructs an object of type unique_lock.
Ensures: pm == 0 and owns == false.

explicit unique_lock(mutex_type& m);
Requires: mutex_type is not a recursive mutex and acquire() is not called at construction or after construction.
Effects: Constructs an object of type unique_lock and calls m.lock().
Ensures: pm == addressof(m) and owns == true.

unique_lock(mutex_type& m, defer_lock_t) noexcept;
Effects: Constructs an object of type unique_lock.
Ensures: pm == addressof(m) and owns == false.

unique_lock(mutex_type& m, try_to_lock_t);
Requires: mutex_type is not a recursive mutex and acquire() is not called at construction or after construction.
Effects: Constructs an object of type unique_lock and calls m.try_lock().
Ensures: pm == addressof(m) and owns == res, where res is the value returned by the call to m.try_lock().

unique_lock(mutex_type& m, adopt_lock_t);
Requires: The calling thread owns the mutex.
Effects: Constructs an object of type unique_lock.
Ensures: pm == addressof(m) and owns == true.
Throws: Nothing.
template<class Clock, class Duration>
unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);

Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex.
The supplied Mutex type shall meet the Cpp17TimedLockable requirements (32.2.5.4).
Effects: Constructs an object of type unique_lock and calls m.try_lock_until(abs_time).
Ensures: pm == addressof(m) and owns == res, where res is the value returned by the call to m.try_lock_until(abs_time).

template<class Rep, class Period>
unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);

Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex.
The supplied Mutex type shall meet the Cpp17TimedLockable requirements (32.2.5.4).
Effects: Constructs an object of type unique_lock and calls m.try_lock_for(rel_time).
Ensures: pm == addressof(m) and owns == res, where res is the value returned by the call to m.try_lock_for(rel_time).

unique_lock(unique_lock&& u) noexcept;

Ensures: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.

unique_lock& operator=(unique_lock&& u);

Effects: If owns calls pm->unlock().
Ensures: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this construction), u.pm == 0 and u.owns == false.

[Note: With a recursive mutex it is possible for both *this and u to own the same mutex before the assignment. In this case, *this will own the mutex after the assignment and u will not. — end note]

Throws: Nothing.

~unique_lock();

Effects: If owns calls pm->unlock().

32.5.4.3.2 Locking [thread.lock.unique.locking]

void lock();

Effects: As if by pm->lock().
Ensures: owns == true.

Throws: Any exception thrown by pm->lock(). system_error when an exception is required (32.2.2).

Error conditions:
(4.1) operation_not_permitted — if pm is nullptr.
(4.2) resource_deadlock_would_occur — if on entry owns is true.

bool try_lock();

Requires—Expects: The supplied Mutex shall meet the Cpp17Lockable requirements (32.2.5.3).
Effects: As if by pm->try_lock().
Returns: The value returned by the call to try_lock().
Ensures: owns == res, where res is the value returned by the call to try_lock().

Throws: Any exception thrown by pm->try_lock(). system_error when an exception is required (32.2.2).

Error conditions:
(10.1) operation_not_permitted — if pm is nullptr.
(10.2) resource_deadlock_would_occur — if on entry owns is true.
template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

11 **Requires**—**Expects**: The supplied Mutex type shall meet the Cpp17TimedLockable requirements (32.2.5.4).
12 **Effects**: As if by pm->try_lock_until(abs_time).
13 **Returns**: The value returned by the call to try_lock_until(abs_time).
14 **Ensures**: owns == res, where res is the value returned by the call to try_lock_until(abs_time).
15 **Throws**: Any exception thrown by pm->try_lock_until(). system_error when an exception is required (32.2.2).
16 **Error conditions**:
17 (16.1) — operation_not_permitted — if pm is nullptr.
18 (16.2) — resource_deadlock_would_occur — if on entry owns is true.

template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

17 **Requires**—**Expects**: The supplied Mutex type shall meet the Cpp17TimedLockable requirements (32.2.5.4).
18 **Effects**: As if by pm->try_lock_for(rel_time).
19 **Returns**: The value returned by the call to try_lock_until(rel_time).
20 **Ensures**: owns == res, where res is the value returned by the call to try_lock_for(rel_time).
21 **Throws**: Any exception thrown by pm->try_lock_for(). system_error when an exception is required (32.2.2).
22 **Error conditions**:
23 (22.1) — operation_not_permitted — if pm is nullptr.
24 (22.2) — resource_deadlock_would_occur — if on entry owns is true.

void unlock();

23 **Effects**: As if by pm->unlock().
24 **Ensures**: owns == false.
25 **Throws**: system_error when an exception is required (32.2.2).
26 **Error conditions**:
27 (26.1) — operation_not_permitted — if on entry owns is false.

### 32.5.4.3.4 Observers [thread.lock.unique.obs]

bool owns_lock() const noexcept;

1 **Returns**: owns.

2 explicit operator bool() const noexcept;

1 **Returns**: owns.
mutex_type *mutex() const noexcept;

Returns: pm.

32.5.4.4  Class template shared_lock

namespace std {

    template<class Mutex>
    class shared_lock {
    public:
        using mutex_type = Mutex;

        // 32.5.4.4.1, construct/copy/destroy
        shared_lock() noexcept;
        explicit shared_lock(mutex_type&m); // blocking
        shared_lock(mutex_type&m, defer_lock_t) noexcept;
        shared_lock(mutex_type&m, try_to_lock_t);
        shared_lock(mutex_type&m, adopt_lock_t);
        template<class Clock, class Duration>
            shared_lock(mutex_type&m, const chrono::time_point<Clock, Duration>& abs_time);
        template<class Rep, class Period>
            shared_lock(mutex_type&m, const chrono::duration<Rep, Period>& rel_time);
        ~shared_lock();

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

        shared_lock(shared_lock&& u) noexcept;
        shared_lock& operator=(shared_lock&& u) noexcept;

        // 32.5.4.4.2, locking
        void lock(); // blocking
        bool try_lock();
        template<class Rep, class Period>
            bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
        template<class Clock, class Duration>
            bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
        void unlock();

        // 32.5.4.4.3, modifiers
        void swap(shared_lock& u) noexcept;
        mutex_type* release() noexcept;

        // 32.5.4.4.4, observers
        bool owns_lock() const noexcept;
        explicit operator bool () const noexcept;
        mutex_type* mutex() const noexcept;

    private:
        mutex_type* pm; // exposition only
        bool owns; // exposition only
    };

    template<class Mutex>
        void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
    }

1 An object of type shared_lock controls the shared ownership of a lockable object within a scope. Shared ownership of the lockable object may be acquired at construction or after construction, and may be transferred, after acquisition, to another shared_lock object. Objects of type shared_lock are not copyable but are movable. The behavior of a program is undefined if the contained pointer pm is not null and the lockable object pointed to by pm does not exist for the entire remaining lifetime of the shared_lock object. The supplied Mutex type shall meet the shared mutex requirements (32.5.3.5).

2 [Note: shared_lock<Mutex> meets the Cpp17TimedLockable requirements (32.2.5.4). — end note]
32.5.4.4.1 Constructors, destructor, and assignment

shared_lock() noexcept;
1   Effects: Constructs an object of type shared_lock.
2     Ensures: pm == nullptr and owns == false.

explicit shared_lock(mutex_type& m);
3   Requires: The calling thread does not own the mutex for any ownership mode.
4     Effects: Constructs an object of type shared_lock and calls m.lock_shared().
5     Ensures: pm == addressof(m) and owns == true.

shared_lock(mutex_type& m, defer_lock_t) noexcept;
6     Effects: Constructs an object of type shared_lock.
7     Ensures: pm == addressof(m) and owns == false.

shared_lock(mutex_type& m, try_to_lock_t);
8     Requires: The calling thread does not own the mutex for any ownership mode.
9     Effects: Constructs an object of type shared_lock and calls m.try_lock_shared().
10    Ensures: pm == addressof(m) and owns == res where res is the value returned by the call to m.try_lock_shared().

shared_lock(mutex_type& m, adopt_lock_t);
11    Requires: The calling thread has shared ownership of the mutex.
12    Effects: Constructs an object of type shared_lock.
13     Ensures: pm == addressof(m) and owns == true.

template<class Clock, class Duration>
14     shared_lock(mutex_type& m,
15          const chrono::time_point<Clock, Duration>& abs_time);
16     Requires: The calling thread does not own the mutex for any ownership mode.
17     Effects: Constructs an object of type shared_lock and calls m.try_lock_shared_until(abs_time).
18     Ensures: pm == addressof(m) and owns == res where res is the value returned by the call to m.try_lock_shared_until(abs_time).

template<class Rep, class Period>
19     shared_lock(mutex_type& m,
20          const chrono::duration<Rep, Period>& rel_time);
21     Requires: The calling thread does not own the mutex for any ownership mode.
22     Effects: Constructs an object of type shared_lock and calls m.try_lock_shared_for(rel_time).
23     Ensures: pm == addressof(m) and owns == res where res is the value returned by the call to m.try_lock_shared_for(rel_time).

~shared_lock();
24     Effects: If owns calls pm->unlock_shared().

shared_lock(shared_lock&& sl) noexcept;
25     Ensures: pm == sl.p.pm and owns == sl.p.owns (where sl.p is the state of sl just prior to this construction), sl.pm == nullptr and sl.owns == false.

shared_lock& operator=(shared_lock&& sl) noexcept;
26     Effects: If owns calls pm->unlock_shared().
27     Ensures: pm == sl.p.pm and owns == sl.p.owns (where sl.p is the state of sl just prior to this assignment), sl.pm == nullptr and sl.owns == false.
32.5.4.4.2 Locking

void lock();
   Effects: As if by pm->lock_shared().
   Ensures: owns == true.
   Throws: Any exception thrown by pm->lock_shared(). system_error when an exception is required (32.2.2).
   Error conditions:
   — operation_not_permitted — if pm is nullptr.
   — resource_deadlock_would_occur — if on entry owns is true.

bool try_lock();
   Effects: As if by pm->try_lock_shared().
   Returns: The value returned by the call to pm->try_lock_shared().
   Ensures: owns == res, where res is the value returned by the call to pm->try_lock_shared().
   Throws: Any exception thrown by pm->try_lock_shared(). system_error when an exception is required (32.2.2).
   Error conditions:
   — operation_not_permitted — if pm is nullptr.
   — resource_deadlock_would_occur — if on entry owns is true.

template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
   Effects: As if by pm->try_lock_shared_until(abs_time).
   Returns: The value returned by the call to pm->try_lock_shared_until(abs_time).
   Ensures: owns == res, where res is the value returned by the call to pm->try_lock_shared_until(abs_time).
   Throws: Any exception thrown by pm->try_lock_shared_until(abs_time). system_error when an exception is required (32.2.2).
   Error conditions:
   — operation_not_permitted — if pm is nullptr.
   — resource_deadlock_would_occur — if on entry owns is true.

template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
   Effects: As if by pm->try_lock_shared_for(rel_time).
   Returns: The value returned by the call to pm->try_lock_shared_for(rel_time).
   Ensures: owns == res, where res is the value returned by the call to pm->try_lock_shared_for(rel_time).
   Throws: Any exception thrown by pm->try_lock_shared_for(rel_time). system_error when an exception is required (32.2.2).
   Error conditions:
   — operation_not_permitted — if pm is nullptr.
   — resource_deadlock_would_occur — if on entry owns is true.

void unlock();
   Effects: As if by pm->unlock_shared().
   Ensures: owns == false.
   Throws: system_error when an exception is required (32.2.2).
Error conditions:

— operation_not_permitted — if on entry owns is false.

32.5.4.4.3 Modifiers

```cpp
void swap(shared_lock& sl) noexcept;
```

Effects: Swaps the data members of *this and sl.

```cpp
mutex_type* release() noexcept;
```

Returns: The previous value of pm.

Ensures: pm == nullptr and owns == false.

```cpp
template<class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;
```

Effects: As if by x.swap(y).

32.5.4.4.4 Observers

```cpp
bool owns_lock() const noexcept;
```

Returns: owns.

```cpp
explicit operator bool() const noexcept;
```

Returns: owns.

```cpp
mutex_type* mutex() const noexcept;
```

Returns: pm.

32.5.5 Generic locking algorithms

```cpp
template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

Requires: Each template parameter type shall meet the Cpp17Lockable requirements. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

Effects: Calls try_lock() for each argument in order beginning with the first until all arguments have been processed or a call to try_lock() fails, either by returning false or by throwing an exception. If a call to try_lock() fails, unlock() is called for all prior arguments with no further calls to try_lock().

Returns: -1 if all calls to try_lock() returned true, otherwise a zero-based index value that indicates the argument for which try_lock() returned false.

```cpp
template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
```

Requires: Each template parameter type shall meet the Cpp17Lockable requirements. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

Effects: All arguments are locked via a sequence of calls to lock(), try_lock(), or unlock() on each argument. The sequence of calls does not result in deadlock, but is otherwise unspecified. [Note: A deadlock avoidance algorithm such as try-and-back-off must be used, but the specific algorithm is not specified to avoid over-constraining implementations. — end note] If a call to lock() or try_lock() throws an exception, unlock() is called for any argument that had been locked by a call to lock() or try_lock().

32.5.6 Call once

32.5.6.1 Struct once_flag

```cpp
namespace std {
    struct once_flag {
        constexpr once_flag() noexcept;
        once_flag(const once_flag&) = delete;
        once_flag& operator=(const once_flag&) = delete;
    };
}
```

§ 32.5.6.1
The class `once_flag` is an opaque data structure that `call_once` uses to initialize data without causing a data race or deadlock.

```cpp
constexpr once_flag() noexcept;
```

**Effects:** Constructs an object of type `once_flag`.

**Synchronization:** The construction of a `once_flag` object is not synchronized.

**Ensures:** The object’s internal state is set to indicate to an invocation of `call_once` with the object as its initial argument that no function has been called.

### 32.5.6.2 Function call_once

```
template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);
```

**Mandates:** `is_invocable_v<Callable, Args...>` is true.

**Effects:** An execution of `call_once` that does not call its `func` is a passive execution. An execution of `call_once` that calls its `func` is an active execution. An active execution shall calls `INVOKE(std::forward<Callable>(func), std::forward<Args>(args)...)` If such a call to `func` throws an exception the execution is exceptional, otherwise it is returning. An exceptional execution propagates the exception to the caller of `call_once`. Among all executions of `call_once` for any given `once_flag`: at most one shall be a returning execution; if there is a returning execution, it shall be the last active execution; and there are passive executions only if there is a returning execution. [Note: Passive executions allow other threads to reliably observe the results produced by the earlier returning execution. — end note]

**Synchronization:** For any given `once_flag`: all active executions occur in a total order; completion of an active execution synchronizes with the start of the next one in this total order; and the returning execution synchronizes with the return from all passive executions.

**Throws:** `system_error` when an exception is required (32.2.2), or any exception thrown by `func`.

**Example:**

```cpp
// global flag, regular function
void init();
std::once_flag flag;

void f() {
  std::call_once(flag, init);
}

// function static flag, function object
struct initializer {
  void operator()();
};

void g() {
  static std::once_flag flag2;
  std::call_once(flag2, initializer());
}

// object flag, member function
class information {
  std::once_flag verified;
  void verifier();
  public:
    void verify() { std::call_once(verified, &information::verifier, *this); }
};
```

—end example
32.6 Condition variables

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class `condition_variable` provides a condition variable that can only wait on an object of type `unique_lock<mutex>`, allowing the implementation to be more efficient. Class `condition_variable_any` provides a general condition variable that can wait on objects of user-supplied lock types.

Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

The executions of `notify_one` and `notify_all` shall be atomic. The executions of `wait`, `wait_for`, and `wait_until` shall be performed in three atomic parts:

1. the release of the mutex and entry into the waiting state;
2. the unblocking of the wait; and
3. the reacquisition of the lock.

The implementation shall behave as if all executions of `notify_one`, `notify_all`, and each part of the `wait`, `wait_for`, and `wait_until` executions are executed in a single unspecified total order consistent with the 'happens before' order.

Condition variable construction and destruction need not be synchronized.

32.6.1 Header `<condition_variable>` synopsis

```cpp
namespace std {
    class condition_variable;
    class condition_variable_any;

    void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

    enum class cv_status { no_timeout, timeout };
}
```

32.6.2 Non-member functions

```cpp
void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

1 **Required—Expects:** lk is locked by the calling thread and either

(1.1) — no other thread is waiting on cond, or

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

2 **Effects:** Transfers ownership of the lock associated with lk into internal storage and schedules cond to be notified when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed. This notification shall be as if is equivalent to:

    lk.unlock();
    cond.notify_all();

3 **Synchronization:** The implied lk.unlock() call is sequenced after the destruction of all objects with thread storage duration associated with the current thread.

4 [*Note:* The supplied lock will be held until the thread exits, and care should be taken to ensure that this does not cause deadlock due to lock ordering issues. After calling `notify_all_at_thread_exit` it is recommended that the thread should be exited as soon as possible, and that no blocking or time-consuming tasks are run on that thread. — *end note*]

5 [*Note:* It is the user’s responsibility to ensure that waiting threads do not erroneously assume that the thread has finished if they experience spurious wakeups. This typically requires that the condition being waited for is satisfied while holding the lock on lk, and that this lock is not released and reacquired prior to calling `notify_all_at_thread_exit`. — *end note*]
32.6.3 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;
        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);

        using native_handle_type = implementation-defined; // see 32.2.3
        native_handle_type native_handle(); // see 32.2.3
    }
}

The class condition_variable shall be a standard-layout class (??).

condition_variable();

Effects: Constructs an object of type condition_variable.

Throws: system_error when an exception is required (32.2.2).

Error conditions:

— resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

~condition_variable();

Requires—Expects: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they could 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.

void notify_one() noexcept;

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

void notify_all() noexcept;

Effects: Unblocks all threads that are blocked waiting for *this.

§ 32.6.3
void wait(unique_lock<mutex>& lock);

   Requires—Expect:  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, or wait_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.

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

   Requires—Expect:  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, or wait_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—Expect:  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, or wait_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 (32.2.4) 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 (32.2.4) specified by abs_time expired, otherwise cv_status::no_timeout.

Throws: Timeout-related exceptions (32.2.4).

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

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

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

Returns: cv_status::timeout if the relative timeout (32.2.4) 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 (32.2.4).

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

Requirements—Expects: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(31.1) — no other thread is waiting on this condition_variable object or
(31.2) — lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_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 (32.2.4) 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);

Requirements—Expects: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(37.1) — no other thread is waiting on this condition_variable object or
(37.2) — lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.
Effects: Equivalent to:
```
return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));
```

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

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

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

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

Throws: Timeout-related exceptions (32.2.4) or any exception thrown by \texttt{pred}.

### 32.6.4 Class \texttt{condition\_variable\_any} [thread.condition.condvarany]

A \texttt{Lock} type shall meet the \texttt{Cpp17BasicLockable} requirements (32.2.5.2). [Note: All of the standard mutex types meet this requirement. If a \texttt{Lock} type other than one of the standard mutex types or a \texttt{unique\_lock} wrapper for a standard mutex type is used with \texttt{condition\_variable\_any}, the user should ensure that any necessary synchronization is in place with respect to the predicate associated with the \texttt{condition\_variable\_any} instance. — end note]

```cpp
namespace std {
    class condition_variable_any {
        public:
            condition_variable_any();
            ~condition_variable_any();

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

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

            // 32.6.4.1, noninterruptible waits
            template<class Lock>
                void wait(Lock& lock);
            template<class Lock, class Predicate>
                void wait(Lock& lock, Predicate pred);

            template<class Lock, class Clock, class Duration>
                cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
            template<class Lock, class Clock, class Duration, class Predicate>
                bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
            template<class Lock, class Rep, class Period>
                cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
            template<class Lock, class Rep, class Period, class Predicate>
                bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);

            // 32.6.4.2, interruptible waits
            template<class Lock, class Predicate>
                bool wait_until(Lock& lock, Predicate pred, stop_token stoken);
            template<class Lock, class Clock, class Duration, class Predicate>
                bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred, stop_token stoken);
            template<class Lock, class Rep, class Period, class Predicate>
                bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred, stop_token stoken);
    }
}
```

Effects: Constructs an object of type \texttt{condition\_variable\_any}.

\[ § 32.6.4 \]
Throws: bad_alloc or system_error when an exception is required (32.2.2).

Error conditions:

(4.1) resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

(4.2) operation_not_permitted — if the thread does not have the privilege to perform the operation.

~condition_variable_any();

Requires—Expects: 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.

void notify_one() noexcept;

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

void notify_all() noexcept;

Effects: Unblocks all threads that are blocked waiting for *this.

32.6.4.1 Noninterruptible waits

template<class Lock>
void wait(Lock& lock);

Effects:

(1.1) Atomically calls lock.unlock() and blocks on *this.

(1.2) When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.

(1.3) The function will unblock when signaled by a call to notify_one(), 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 is locked by the calling thread.

Throws: Nothing.

template<class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);

Effects: Equivalent to:

while (!pred())
    wait(lock);

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

Effects:

(6.1) Atomically calls lock.unlock() and blocks on *this.

(6.2) When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.

(6.3) The function will unblock when signaled by a call to notify_one(), a call to notify_all(), expiration of the absolute timeout (32.2.4) specified by abs_time, or spuriously.

(6.4) 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 is locked by the calling thread.
Returns: cv_status::timeout if the absolute timeout (32.2.4) specified by abs_time expired, otherwise cv_status::no_timeout.

Throws: Timeout-related exceptions (32.2.4).

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

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

Returns: cv_status::timeout if the relative timeout (32.2.4) 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 is locked by the calling thread.

Throws: Timeout-related exceptions (32.2.4).

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

Effects: Equivalent to:
```
while (!pred()) {
  if (wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
  return true;
}
```

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

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

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

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

### 32.6.4.2 Interruptible waits [thread.condvarany.intwait]

The following wait functions will be notified when there is a stop request on the passed stop_token. In that case the functions return immediately, returning false if the predicate evaluates to false.

```
template<class Lock, class Predicate>
bool wait_until(Lock& lock, Predicate pred, stop_token stoken);
```

Effects: Registers for the duration of this call *this to get notified on a stop request on stoken during this call and then equivalent to:
```
while (!stoken.stopRequested()) {
  if (pred())
    return true;
  wait(lock);
}
```

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether there was a stop request. — end note]

Ensures: lock is locked by the calling thread.

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

Throws: Any exception thrown by pred.
template<class Lock, class Clock, class Duration, class Predicate>
bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time,
               Predicate pred, stop_token stoken);

Effects: Registers for the duration of this call *this to get notified on a stop request on stoken during
d this call and then equivalent to:

while (!stoken.stop_requested()) {
    if (pred())
        return true;
    if (cv.wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
    return pred();
}

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

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

Ensures: lock is locked by the calling thread.

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

Throws: Timeout-related exceptions (32.2.4), or any exception thrown by pred.

template<class Lock, class Rep, class Period, class Predicate>
bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time,
              Predicate pred, stop_token stoken);

Effects: Equivalent to:

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

32.7 Semaphore

Semaphores are lightweight synchronization primitives used to constrain concurrent access to a shared
resource. They are widely used to implement other synchronization primitives and, whenever both are
applicable, can be more efficient than condition variables.

A counting semaphore is a semaphore object that models a non-negative resource count. A binary semaphore
is a semaphore object that has only two states. A binary semaphore should be more efficient than the default
implementation of a counting semaphore with a unit resource count.

32.7.1 Header <semaphore> synopsis

namespace std {
    template<ptrdiff_t least_max_value = implementation-defined>
    class counting_semaphore;

    using binary_semaphore = counting_semaphore<1>;
}

32.7.2 Class template counting_semaphore

namespace std {
    template<ptrdiff_t least_max_value = implementation-defined>
    class counting_semaphore {
    public:
        static constexpr ptrdiff_t max() noexcept;

        constexpr explicit counting_semaphore(ptrdiff_t desired);
        ~counting_semaphore();

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

        //... other member functions...
    }
void release(ptrdiff_t update = 1);
void acquire();
bool try_acquire() noexcept;
template<class Rep, class Period>
  bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
template<class Clock, class Duration>
  bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);

private:
  ptrdiff_t counter;   // exposition only
};

1 Class template counting_semaphore maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the thread will block until another thread increments the counter by releasing the semaphore.

2 least_max_value shall be non-negative; otherwise the program is ill-formed.

3 Concurrent invocations of the member functions of counting_semaphore, other than its destructor, do not introduce data races.

static constexpr ptrdiff_t max() noexcept;

Returns: The maximum value of counter. This value is greater than or equal to least_max_value.

constexpr explicit counting_semaphore(ptrdiff_t desired);

Expects: desired >= 0 is true, and desired <= max() is true.
Effects: Initializes counter with desired.
Throws: Nothing.

void release(ptrdiff_t update = 1);

Expects: update >= 0 is true, and update <= max() - counter is true.
Effects: Atomically execute counter += update. Then, unblocks any threads that are waiting for counter to be greater than zero.
Synchronization: Strongly happens before invocations of try_acquire that observe the result of the effects.
Throws: system_error when an exception is required (32.2.2).
Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

bool try_acquire() noexcept;

Effects:
(13.1) With low probability, returns immediately. An implementation should ensure that try_acquire does not consistently return false in the absence of contending acquisitions.
(13.2) Otherwise, atomically check whether counter is greater than zero and, if so, decrement counter by one.
Returns: true if counter was decremented, otherwise false.

void acquire();

Effects: Repeatedly performs the following steps, in order:
(15.1) Evaluates try_acquire. If the result is true, returns.
(15.2) Blocks on *this until counter is greater than zero.
Throws: system_error when an exception is required (32.2.2).
Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).
template<class Clock, class Duration>
bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);

Effects: Repeatedly performs the following steps, in order:

- Evaluates try_acquire(). If the result is true, returns true.
- Blocks on *this until counter is greater than zero or until the timeout expires. If it is unblocked
  by the timeout expiring, returns false.

The timeout expires (32.2.4) when the current time is after abs_time (for try_acquire_until) or
when at least rel_time has passed from the start of the function (for try_acquire_for).

Throws: Timeout-related exceptions (32.2.4), or system_error when a non-timeout-related exception
is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

32.8 Coordination types

This subclause describes various concepts related to thread coordination, and defines the coordination types
latch and barrier. These types facilitate concurrent computation performed by a number of threads.

32.8.1 Latches

A latch is a thread coordination mechanism that allows any number of threads to block until an expected
number of threads arrive at the latch (via the count_down function). The expected count is set when the
latch is created. An individual latch is a single-use object; once the expected count has been reached, the
latch cannot be reused.

32.8.1.1 Header <latch> synopsis

namespace std {
  class latch;
}

32.8.1.2 Class latch

namespace std {
  class latch {
  public:
    constexpr explicit latch(ptrdiff_t expected);
    ~latch();
    latch(const latch&) = delete;
    latch& operator=(const latch&) = delete;
    void count_down(ptrdiff_t update = 1);
    bool try_wait() const noexcept;
    void wait() const;
    void arrive_and_wait(ptrdiff_t update = 1);
  private:
    ptrdiff_t counter;  // exposition only
  };
}

A latch maintains an internal counter that is initialized when the latch is created. Threads can block on the
latch object, waiting for counter to be decremented to zero.

Concurrent invocations of the member functions of latch, other than its destructor, do not introduce data
races.

constexpr explicit latch(ptrdiff_t expected);

Expects: expected >= 0 is true.

Effects: Initializes counter with expected.

Throws: Nothing.
void count_down(ptrdiff_t update = 1);

Expects: \( update \geq 0 \) is true, and \( update \leq counter \) is true.

Effects: Atomically decrements \( counter \) by \( update \). If \( counter \) is equal to zero, unblocks all threads blocked on \(*this\).

Synchronization: Strongly happens before the returns from all calls that are unblocked.

Throws: \texttt{system\_error} when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

bool try_wait() const noexcept;

Returns: With very low probability \texttt{false}. Otherwise \( counter == 0 \).

void wait() const;

Effects: If \( counter \) equals zero, returns immediately. Otherwise, blocks on \(*this\) until a call to \texttt{count\_down} that decrements \( counter \) to zero.

Throws: \texttt{system\_error} when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

void arrive_and_wait(ptrdiff_t update = 1);

Effects: Equivalent to:
\begin{verbatim}
  count_down(update);
  wait();
\end{verbatim}

32.8.2 Barriers

A barrier is a thread coordination mechanism whose lifetime consists of a sequence of barrier phases, where each phase allows at most an expected number of threads to block until the expected number of threads arrive at the barrier. [Note: A barrier is useful for managing repeated tasks that are handled by multiple threads. — end note]

32.8.2.1 Header <barrier> synopsis

```cpp
namespace std {
  template<class CompletionFunction = see below>
  class barrier;
}
```

32.8.2.2 Class template barrier

```cpp
namespace std {
  template<class CompletionFunction = see below>
  class barrier {
    public:
      using arrival_token = see below;

      constexpr explicit barrier(ptrdiff_t expected,
        CompletionFunction f = CompletionFunction());
      ~barrier();

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

      [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
      void wait(arrival_token&& arrival) const;

      void arrive_and_wait();
      void arrive_and_drop();

    private:
      CompletionFunction completion; // exposition only
  };
}
```
Each barrier phase consists of the following steps:

1. The expected count is decremented by each call to `arrive` or `arrive_and_drop`.
2. When the expected count reaches zero, the phase completion step is run. For the specialization with the default value of the `CompletionFunction` template parameter, the completion step is run as part of the call to `arrive` or `arrive_and_drop` that caused the expected count to reach zero. For other specializations, the completion step is run on one of the threads that arrived at the barrier during the phase.
3. When the completion step finishes, the expected count is reset to what was specified by the expected argument to the constructor, possibly adjusted by calls to `arrive_and_drop`, and the next phase starts.

Each phase defines a phase synchronization point. Threads that arrive at the barrier during the phase can block on the phase synchronization point by calling `wait`, and will remain blocked until the phase completion step is run.

The phase completion step that is executed at the end of each phase has the following effects:

1. Invokes the completion function, equivalent to `completion()`.
2. Unblocks all threads that are blocked on the phase synchronization point.

The end of the completion step strongly happens before the returns from all calls that were unblocked by the completion step. For specializations that do not have the default value of the `CompletionFunction` template parameter, the behavior is undefined if any of the barrier object’s member functions other than `wait` are called while the completion step is in progress.

Concurrent invocations of the member functions of `barrier`, other than its destructor, do not introduce data races. The member functions `arrive` and `arrive_and_drop` execute atomically.

CompletionFunction shall meet the `Cpp17MoveConstructible` (Table ??) and `Cpp17Destructible` (Table ??) requirements. `is_nothrow_invocable_v<CompletionFunction&>` shall be true.

The default value of the `CompletionFunction` template parameter is an unspecified type, such that, in addition to satisfying the requirements of `CompletionFunction`, it meets the `Cpp17DefaultConstructible` requirements (Table ??) and `completion()` has no effects.

`barrier::arrival_token` is an unspecified type, such that it meets the `Cpp17MoveConstructible` (Table ??), `Cpp17MoveAssignable` (Table ??), and `Cpp17Destructible` (Table ??) requirements.

```cpp
constexpr explicit barrier(ptrdiff_t expected, 
CompletionFunction f = CompletionFunction());
```

`Expected: expected >= 0 is true.
Effects: Sets both the initial expected count for each barrier phase and the current expected count for the first phase to expected. Initializes completion with std::move(f). Starts the first phase. [Note: If expected is 0 this object can only be destroyed. — end note]` 

```cpp
[[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
```

`Expected: update > 0 is true, and update is less than or equal to the expected count for the current barrier phase.
Effects: Constructs an object of type arrival_token that is associated with the phase synchronization point for the current phase. Then, decrements the expected count by update.
Synchronization: The call to arrive strongly happens before the start of the phase completion step for the current phase.
Returns: The constructed arrival_token object.
Throws: system_error when an exception is required (32.2.2).
Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

[Note: This call can cause the completion step for the current phase to start. — end note]`
void wait(arrival_token&& arrival) const;
18  
   Expects: arrival is associated with the phase synchronization point for the current phase or the 
immediately preceding phase of the same barrier object.
19  
   Effects: Blocks at the synchronization point associated with std::move(arrival) until the phase 
completion step of the synchronization point’s phase is run. [Note: If arrival is associated with the 
synchronization point for a previous phase, the call returns immediately. — end note]
20  
   Throws: system_error when an exception is required (32.2.2).
21  
   Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).
22
void arrive_and_wait();
23  
   Effects: Equivalent to: wait(arrive()).
24
void arrive_and_drop();
25  
   Expects: The expected count for the current barrier phase is greater than zero.
26  
   Effects: Decrements the initial expected count for all subsequent phases by one. Then decrements the 
expected count for the current phase by one.
27  
   Synchronization: The call to arrive_and_drop strongly happens before the start of the phase completion 
step for the current phase.
28  
   Throws: system_error when an exception is required (32.2.2).
29  
   Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).
30  
   [Note: This call can cause the completion step for the current phase to start. — end note]
31
32.9 Futures
32.9.1 Overview
32.9 describes components that a C++ program can use to retrieve in one thread the result (value or exception) 
from a function that has run in the same thread or another thread. [Note: These components are not 
restricted to multi-threaded programs but can be useful in single-threaded programs as well. — end note]
33
32.9.2 Header <future> synopsis
34
namespace std {
35  
   enum class future_errc {
52       broken_promise = implementation-defined,
53       future_already_retrieved = implementation-defined,
54       promise_already_satisfied = implementation-defined,
55       no_state = implementation-defined
56    };
57  
   enum class launch : unspecified {
59       async = unspecified,
60       deferred = unspecified,
61           implementation-defined
62    };
63  
   enum class future_status {
64       ready,
65       timeout,
66       deferred
67    };
68  
   template<> struct is_error_code_enum<future_errc> : public true_type { };
69  
   error_code make_error_code(future_errc e) noexcept;
70  
   error_condition make_error_condition(future_errc e) noexcept;
71  
   const error_category& future_category() noexcept;
72  
   class future_error;
73
§ 32.9.2
46
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 Alloc>
struct uses_allocator<promise<R>, Alloc>;

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;
// not defined

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

template<class F, class... Args>
[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(F&& f, Args&&... args);

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

template<class F, class... Args>
[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(launch policy, F&& f, Args&&... args);

1 The enum type launch is a bitmask type (?) with elements launch::async and launch::deferred. [Note: Implementations can provide bitmasks to specify restrictions on task interaction by functions launched by async() applicable to a corresponding subset of available launch policies. Implementations can extend the behavior of the first overload of async() by adding their extensions to the launch policy under the “as if” rule. —end note]

2 The enum values of future_errc are distinct and not zero.

32.9.3 Error handling

const error_category& future_category() noexcept;

Returns: A reference to an object of a type derived from class error_category.

The object’s default_error_condition and equivalent virtual functions shall behave as specified for the class error_category. The object’s name virtual function shall return a pointer to the string "future".

error_code make_error_code(future_errc e) noexcept;

Returns: error_code(static_cast<int>(e), future_category()).

error_condition make_error_condition(future_errc e) noexcept;

Returns: error_condition(static_cast<int>(e), future_category()).

32.9.4 Class future_error

namespace std {

class future_error : public logic_error {
public:
    explicit future_error(future_errc e);

    const error_code& code() const noexcept;
    const char* what() const noexcept;

§ 32.9.4
private:
    error_code ec_;  // exposition only
};

explicit future_error(future_errc e);
  
Effects: Constructs an object of class future_error and initializes ec_ with make_error_code(e).

const error_code& code() const noexcept;
  
Returns: ec_.

const char* what() const noexcept;
  
Returns: An ntbs incorporating code().message().

### 32.9.5 Shared state

Many of the classes introduced in this subclause use some state to communicate results. This shared state consists of some state information and some (possibly not yet evaluated) result, which can be a (possibly void) value or an exception. [Note: Futures, promises, and tasks defined in this clause reference such shared state. — end note]

[Note: The result can be any kind of object including a function to compute that result, as used by async when policy is launch::deferred. — end note]

An asynchronous return object is an object that reads results from a shared state. A waiting function of an asynchronous return object is one that potentially blocks to wait for the shared state to be made ready. If a waiting function can return before the state is made ready because of a timeout (32.2.5), then it is a timed waiting function, otherwise it is a non-timed waiting function.

An asynchronous provider is an object that provides a result to a shared state. The result of a shared state is set by respective functions on the asynchronous provider. [Note: Such as promises or tasks. — end note] The means of setting the result of a shared state is specified in the description of those classes and functions that create such a state object.

When an asynchronous return object or an asynchronous provider is said to release its shared state, it means:

1. if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and
2. the return object or provider gives up its reference to its shared state; and
3. these actions will not block for the shared state to become ready, except that it may block if all of the following are true: the shared state was created by a call to std::async, the shared state is not yet ready, and this was the last reference to the shared state.

When an asynchronous provider is said to make its shared state ready, it means:

1. first, the provider marks its shared state as ready; and
2. second, the provider unblocks any execution agents waiting for its shared state to become ready.

When an asynchronous provider is said to abandon its shared state, it means:

1. first, if that state is not ready, the provider
   1.1 stores an exception object of type future_error with an error condition of broken_promise within its shared state; and then
   1.2 makes its shared state ready;
   2. second, the provider releases its shared state.

A shared state is ready only if it holds a value or an exception ready for retrieval. Waiting for a shared state to become ready may invoke code to compute the result on the waiting thread if so specified in the description of the class or function that creates the state object.

Calls to functions that successfully set the stored result of a shared state synchronize with (??) calls to functions successfully detecting the ready state resulting from that setting. The storage of the result (whether normal or exceptional) into the shared state synchronizes with (??) the successful return from a call to a waiting function on the shared state.
Some functions (e.g., `promise::set_value_at_thread_exit`) delay making the shared state ready until the calling thread exits. The destruction of each of that thread’s objects with thread storage duration (?) is sequenced before making that shared state ready.

Access to the result of the same shared state may conflict (?). [Note: This explicitly specifies that the result of the shared state is visible in the objects that reference this state in the sense of data race avoidance (?). For example, concurrent accesses through references returned by `shared_future::get()` (32.9.8) must either use read-only operations or provide additional synchronization. — end note]

32.9.6 Class template `promise` [futures.promise]

```cpp
class promise {
public:
  promise();
  template<class Allocator>
  promise(allocator_arg_t, const Allocator& a);
  promise(promise&& rhs) noexcept;
  promise(const promise&) = delete;
  ~promise();

  // assignment
  promise& operator=(promise&& rhs) noexcept;
  promise& operator=(const promise&) = delete;
  void swap(promise& other) noexcept;

  // retrieving the result
  future<R> get_future();

  // setting the result
  void set_value(see below);
  void set_exception(exception_ptr p);

  // setting the result with deferred notification
  void set_value_at_thread_exit(see below);
  void set_exception_at_thread_exit(exception_ptr p);
};
```

1 The implementation shall provide the template `promise` and two specializations, `promise<R&>` and `promise<void>`. These differ only in the argument type of the member functions `set_value` and `set_value_at_thread_exit`, as set out in their descriptions, below.

2 The `set_value`, `set_exception`, `set_value_at_thread_exit`, and `set_exception_at_thread_exit` member functions behave as though they acquire a single mutex associated with the promise object while updating the promise object.

```cpp
template<class R, class Alloc>
struct uses_allocator<promise<R>, Alloc> : true_type {
};
```

3 Requires: Alloc shall meet the `Cpp17Allocator` requirements (Table ??).

```cpp
promise();
template<class Allocator>
promise(allocator_arg_t, const Allocator& a);
```

4 Effects: Constructs a promise object and creates a shared state. The second constructor uses the allocator `a` to allocate memory for the shared state.
promise(promise&& rhs) noexcept;
   Effects: Constructs a new promise object and transfers ownership of the shared state of rhs (if any) to the newly-constructed object.
   Ensures: rhs has no shared state.
~promise();
   Effects: Abandons any shared state (32.9.5).

promise& operator=(promise&& rhs) noexcept;
   Effects: Abandons any shared state (32.9.5) and then as if promise(std::move(rhs)).swap(*this).
   Returns: *this.

void swap(promise& other) noexcept;
   Effects: Exchanges the shared state of *this and other.
   Ensures: *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

future<R> get_future();
   Returns: A future<R> object with the same shared state as *this.
Synchronization: Calls to this function do not introduce data races (??) with calls to set_value, set_exception, set_value_at_thread_exit, or set_exception_at_thread_exit. [Note: Such calls need not synchronize with each other. — end note]
   Throws: future_error if *this has no shared state or if get_future has already been called on a promise with the same shared state as *this.
   Error conditions:
   — future_already_retrieved if get_future has already been called on a promise with the same shared state as *this.
   — no_state if *this has no shared state.

void promise::set_value(const R& r);
void promise::set_value(R&& r);
void promise<R&>::set_value(R& r);
void promise<void>::set_value();
   Effects: Atomically stores the value r in the shared state and makes that state ready (32.9.5).
   Throws:
   — future_error if its shared state already has a stored value or exception, or
   — for the first version, any exception thrown by the constructor selected to copy an object of R, or
   — for the second version, any exception thrown by the constructor selected to move an object of R.
   Error conditions:
   — promise_already_satisfied if its shared state already has a stored value or exception.
   — no_state if *this has no shared state.

void set_exception(exception_ptr p);
   Requires—Expects: p is not null.
   Effects: Atomically stores the exception pointer p in the shared state and makes that state ready (32.9.5).
   Throws: future_error if its shared state already has a stored value or exception.
   Error conditions:
   — promise_already_satisfied if its shared state already has a stored value or exception.
   — no_state if *this has no shared state.

void promise::set_value_at_thread_exit(const R& r);
void promise::set_value_at_thread_exit(R&& r);
void promise<R&>::set_value_at_thread_exit(R& r);
void promise<void>::set_value_at_thread_exit();

Effects: Stores the value r in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws:
(24.1) — future_error if its shared state already has a stored value or exception, or
(24.2) — for the first version, any exception thrown by the constructor selected to copy an object of R, or
(24.3) — for the second version, any exception thrown by the constructor selected to move an object of R.

Error conditions:
(25.1) — promise_already_satisfied if its shared state already has a stored value or exception.
(25.2) — no_state if *this has no shared state.

void set_exception_at_thread_exit(exception_ptr p);

Requires — Expects: p is not null.

Effects: Stores the exception pointer p in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws: future_error if an error condition occurs.

Error conditions:
(29.1) — promise_already_satisfied if its shared state already has a stored value or exception.
(29.2) — no_state if *this has no shared state.

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

Effects: As if by x.swap(y).

32.9.7 Class template future

The class template future defines a type for asynchronous return objects which do not share their shared state with other asynchronous return objects. A default-constructed future object has no shared state. A future object with shared state can be created by functions on asynchronous providers (32.9.5) or by the move constructor and shares its shared state with the original asynchronous provider. The result (value or exception) of a future object can be set by calling a respective function on an object that shares the same shared state.

[Note: Member functions of future do not synchronize with themselves or with member functions of shared_future. — end note]

The effect of calling any member function other than the destructor, the move-assignment operator, share, or valid on a future object for which valid() == false is undefined. [Note: It is valid to move from a future object for which valid() == false. — end note] [Note: Implementations should detect this case and throw an object of type future_error with an error condition of future_errc::no_state. — end note]

namespace std {
    template<class R>
    class future {
    public:
        future() noexcept;
        future(future&&) noexcept;
        future(const future&) = delete;
        ~future();
        future& operator=(const future&) = delete;
        future& operator=(future&&) noexcept;
        shared_future<R> share() noexcept;
    }
// retrieving the value
see below get();

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

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;
};
}

The implementation \textit{shall provide} the template \texttt{future} and two specializations, \texttt{future<R&>} and \texttt{future<void>}. These differ only in the return type and return value of the member function \texttt{get}, as set out in its description, below.

\texttt{future()} noexcept;
\begin{itemize}
\item \textbf{Effects:} Constructs an empty future object that does not refer to a shared state.
\item \textbf{Ensures:} valid() == false.
\end{itemize}

\texttt{future(future&& rhs) noexcept;}
\begin{itemize}
\item \textbf{Effects:} Move constructs a future object that refers to the shared state that was originally referred to by \texttt{rhs} (if any).
\item \textbf{Ensures:}
\begin{itemize}
\item (8.1) valid() returns the same value as \texttt{rhs.valid()} prior to the constructor invocation.
\item (8.2) \texttt{rhs.valid()} == false.
\end{itemize}
\end{itemize}

\texttt{\texttt{~future}();}
\begin{itemize}
\item \textbf{Effects:}
\begin{itemize}
\item (9.1) Releases any shared state (32.9.5);
\item (9.2) Destroys *this.
\end{itemize}
\end{itemize}

\texttt{future& operator=(future&& rhs) noexcept;}
\begin{itemize}
\item \textbf{Effects:}
\begin{itemize}
\item (10.1) Releases any shared state (32.9.5).
\item (10.2) Move assigns the contents of \texttt{rhs} to *this.
\end{itemize}
\item \textbf{Ensures:}
\begin{itemize}
\item (11.1) valid() returns the same value as \texttt{rhs.valid()} prior to the assignment.
\item (11.2) \texttt{rhs.valid()} == false.
\end{itemize}
\end{itemize}

\texttt{shared_future<R> share() noexcept;}
\begin{itemize}
\item \textbf{Returns:} \texttt{shared_future<R>}(std::move(*this)).
\item \textbf{Ensures:} valid() == false.
\end{itemize}

\texttt{R future::get();}
\texttt{R& future<R&>::get();}
\texttt{void future<void>::get();}

[\textit{Note:} As described above, the template and its two required specializations differ only in the return type and return value of the member function \texttt{get}. \textit{— end note}]
\begin{itemize}
\item \textbf{Effects:}
\begin{itemize}
\item (15.1) Wait\texttt{s} until the shared state is ready, then retrieves the value stored in the shared state;
\item (15.2) Releases any shared state (32.9.5).
\end{itemize}
\end{itemize}


Returns:

16.1 future::get() returns the value v stored in the object’s shared state as std::move(v).
16.2 future<R&>::get() returns the reference stored as value in the object’s shared state.
16.3 future<void>::get() returns nothing.

Throws: The stored exception, if an exception was stored in the shared state.

Ensures: valid() == false.

bool valid() const noexcept;
Returns: true only if *this refers to a shared state.

void wait() const;
Effects: Blocks until the shared state is ready.

template<class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared
state is ready or until the relative timeout (32.2.4) specified by rel_time has expired.

Returns:

17.1 future_status::deferred if the shared state contains a deferred function.
17.2 future_status::ready if the shared state is ready.
17.3 future_status::timeout if the function is returning because the relative timeout (32.2.4) specified
by rel_time has expired.

Throws: timeout-related exceptions (32.2.4).

template<class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared
state is ready or until the absolute timeout (32.2.4) specified by abs_time has expired.

Returns:

18.1 future_status::deferred if the shared state contains a deferred function.
18.2 future_status::ready if the shared state is ready.
18.3 future_status::timeout if the function is returning because the absolute timeout (32.2.4)
specified by abs_time has expired.

Throws: timeout-related exceptions (32.2.4).

32.9.8 Class template shared_future

The class template shared_future defines a type for asynchronous return objects which can share
their shared state with other asynchronous return objects. A default-constructed shared_future object has
no shared state. A shared_future object with shared state can be created by conversion from a future object
and shares its shared state with the original asynchronous provider (32.9.5) of the shared state. The result
(value or exception) of a shared_future object can be set by calling a respective function on an object that
shares the same shared state.

Note: Member functions of shared_future do not synchronize with themselves, but they synchronize with
the shared state. — end note

Note: It is valid to copy or move from a shared_future object for which valid() is false. — end note

Note: Implementations should detect this case and throw an object of type future_error with an error
condition of future_errc::no_state. — end note

namespace std {
    template<class R>
    class shared_future {

§ 32.9.8 53
public:
    shared_future() noexcept;
    shared_future(const shared_future& rhs) noexcept;
    shared_future(future<R>&&) noexcept;
    shared_future(shared_future&& rhs) noexcept;
    ~shared_future();
    shared_future& operator=(const shared_future& rhs) noexcept;
    shared_future& operator=(shared_future&& rhs) noexcept;

    // retrieving the value
    see below get() const;

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

The implementation shall provides the template `shared_future` and two specializations, `shared_future<R&>` and `shared_future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

```
shared_future() noexcept;
    Effects: Constructs an empty shared_future object that does not refer to a shared state.
    Ensures: valid() == false.

shared_future(const shared_future& rhs) noexcept;
    Effects: Constructs a shared_future object that refers to the same shared state as rhs (if any).
    Ensures: valid() returns the same value as rhs.valid().

shared_future(future<R>&& rhs) noexcept;
shared_future(shared_future&& rhs) noexcept;
    Effects: Move constructs a shared_future object that refers to the shared state that was originally referred to by rhs (if any).
    Ensures:
    (10.1) valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
    (10.2) rhs.valid() == false.

-shared_future();
    Effects:
    (11.1) Replaces any shared state (32.9.5);
    (11.2) destroys *this.

shared_future& operator=(shared_future&& rhs) noexcept;
    Effects:
    (12.1) Replaces any shared state (32.9.5);
    (12.2) move assigns the contents of rhs to *this.
    Ensures:
    (13.1) valid() returns the same value as rhs.valid() returned prior to the assignment.
    (13.2) rhs.valid() == false.
```
```cpp
shared_future& operator=(const shared_future& rhs) noexcept;

Effects:
(14.1) — Releases any shared state (32.9.5);
(14.2) — assigns the contents of rhs to *this. [Note: As a result, *this refers to the same shared state as rhs (if any). — end note]

Ensures: valid() == rhs.valid().

const R& shared_future::get() const;
R& shared_future<R&>::get() const;
void shared_future<void>::get() const;

[Note: As described above, the template and its two required specializations differ only in the return type and return value of the member function get. — end note]

[Note: Access to a value object stored in the shared state is unsynchronized, so programmers should apply only those operations on R that do not introduce a data race (??). — end note]

Effects: wait()s until the shared state is ready, then retrieves the value stored in the shared state.

Returns:
(19.1) — shared_future::get() returns a const reference to the value stored in the object’s shared state. [Note: Access through that reference after the shared state has been destroyed produces undefined behavior; this can be avoided by not storing the reference in any storage with a greater lifetime than the shared_future object that returned the reference. — end note]
(19.2) — shared_future<R&>::get() returns the reference stored as value in the object’s shared state.
(19.3) — shared_future<void>::get() returns nothing.

Throws: The stored exception, if an exception was stored in the shared state.

bool valid() const noexcept;

Returns: true only if *this refers to a shared state.

void wait() const;

Effects: Blocks until the shared state is ready.

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

Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared state is ready or until the relative timeout (32.2.4) specified by rel_time has expired.

Returns:
(24.1) — future_status::deferred if the shared state contains a deferred function.
(24.2) — future_status::ready if the shared state is ready.
(24.3) — future_status::timeout if the function is returning because the relative timeout (32.2.4) specified by rel_time has expired.

Throws: timeout-related exceptions (32.2.4).

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

Effects: None if the shared state contains a deferred function (32.9.9), otherwise blocks until the shared state is ready or until the absolute timeout (32.2.4) specified by abs_time has expired.

Returns:
(27.1) — future_status::deferred if the shared state contains a deferred function.
(27.2) — future_status::ready if the shared state is ready.
(27.3) — future_status::timeout if the function is returning because the absolute timeout (32.2.4) specified by abs_time has expired.

Throws: timeout-related exceptions (32.2.4).

§ 32.9.8 55
```
32.9.9 Function template async

The function template `async` provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a `future` object with which it shares a shared state.

```cpp
template<class F, class... Args>
[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(F&& f, Args&&... args);

template<class F, class... Args>
[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(launch policy, F&& f, Args&&... args);
```

1. Requires: `F` and each `T_i` in `Args` shall meet the `Cpp17MoveConstructible` requirements, and `INVOKE(decay-copy(std::forward<F>(f)),
  decay-copy(std::forward<Args>(args))...)` // see ??, 32.4.2.2 shall be a valid expression.

2. Mandates: The following are all true:
   - `is_move_constructible<decay_t<F>>`,
   - `(is_move_constructible<decay_t<T_i>> &&...)`, and
   - `is_invocable_v<decay_t<F>, decay_t<Args>...>`.

3. Expects: `F` and each `T_i` in `Args` meet the `Cpp17MoveConstructible` requirements.

4. Effects: The first function behaves the same as a call to the second function with a policy argument of `launch::async | launch::deferred` and the same arguments for `F` and `Args`. The second function creates a shared state that is associated with the returned `future` object. The further behavior of the second function depends on the policy argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):
   - If `launch::async` is set in policy, calls `INVOKE(decay-copy(std::forward<F>(f)),
     decay-copy(std::forward<Args>(args))...)` (??, 32.4.2.2) as if in a new thread of execution represented by a `thread` object with the calls to `decay-copy` being evaluated in the thread that called `async`. Any return value is stored as the result in the shared state. Any exception propagated from the execution of `INVOKE(decay-copy(std::forward<F>(f)),
     decay-copy(std::forward<Args>(args))...)` is stored as the exceptional result in the shared state. The `thread` object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.
   - If `launch::deferred` is set in policy, stores `decay-copy(std::forward<F>(f))` and `decay-copy(std::forward<Args>(args))...` in the shared state. These copies of `f` and `args` constitute a deferred function. Invocation of the deferred function evaluates `INVOKE(std::move(g),
  std::move(xyz))` where `g` is the stored value of `decay-copy(std::forward<F>(f))` and `xyz` is the stored copy of `decay-copy(std::forward<Args>(args))...`. Any return value is stored as the result in the shared state. Any exception propagated from the execution of the deferred function is stored as the exceptional result in the shared state. The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (32.9.5) on an asynchronous return object referring to this shared state shall invokes the deferred function in the thread that called the waiting function. Once evaluation of `INVOKE(std::move(g),
  std::move(xyz))` begins, the function is no longer considered deferred. [Note: If this policy is specified together with other policies, such as when using a policy value of `launch::async | launch::deferred`, implementations should defer invocation or the selection of the policy when no more concurrency can be effectively exploited. — end note]
   - If no value is set in the launch policy, or a value is set that is neither specified in this document nor by the implementation, the behavior is undefined.

5. Returns: An object of type `future<invoke_result_t<decay_t<F>, decay_t<Args>...>>` that refers to the shared state created by this call to `async`. [Note: If a future obtained from `async` is moved outside the local scope, other code that uses the future should be aware that the future’s destructor can block for the shared state to become ready. — end note]

6. Synchronization: Regardless of the provided policy argument,
the invocation of `async` synchronizes with (??) the invocation of `f`. [Note: This statement applies even when the corresponding `future` object is moved to another thread. — end note] ; and

the completion of the function `f` is sequenced before (??) the shared state is made ready. [Note: `f` might not be called at all, so its completion might never happen. — end note]

If the implementation chooses the `launch::async` policy,

— a call to a waiting function on an asynchronous return object that shares the shared state created by this `async` call shall block until the associated thread has completed, as if joined, or else time out (32.4.2.5);

— the associated thread completion synchronizes with (??) the return from the first function that successfully detects the ready status of the shared state or with the return from the last function that releases the shared state, whichever happens first.

7 Throws: `system_error` if `policy == launch::async` and the implementation is unable to start a new thread, or `std::bad_alloc` if memory for the internal data structures could not be allocated.

Error conditions:

— `resource_unavailable_try_again` — if `policy == launch::async` and the system is unable to start a new thread.

8 [Example:

```cpp
int work1(int value);
int work2(int value);
int work(int value) {
  auto handle = std::async([&]{ return work2(value); });
  int tmp = work1(value);
  return tmp + handle.get();  // #1
}
```

[Note: Line #1 might not result in concurrency because the `async` call uses the default policy, which may use `launch::deferred`, in which case the lambda might not be invoked until the `get()` call; in that case, `work1` and `work2` are called on the same thread and there is no concurrency. — end note] — end example]

32.9.10 Class template packaged_task

The class template `packaged_task` defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.

When the `packaged_task` object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the shared state. Any futures that share the shared state will then be able to access the stored result.

```cpp
namespace std {
  template<class> class packaged_task;  // not defined

  template<class R, class... ArgTypes>
  class packaged_task<R(ArgTypes...)>
  {
    public:
    // construction and destruction
    packaged_task() noexcept;
    template<class F>
    explicit packaged_task(F&& f);
    packaged_task();

    // no copy
    packaged_task(const packaged_task&) = delete;
    packaged_task& operator=(const packaged_task&) = delete;

    // move support
    packaged_task(packaged_task&& rhs) noexcept;
    packaged_task& operator=(packaged_task&& rhs) noexcept;
    void swap(packaged_task& other) noexcept;

    bool valid() const noexcept;
  }

  § 32.9.10 57
```
// result retrieval
future<R> get_future();

// execution
void operator()(ArgTypes...);
void make_ready_at_thread_exit(ArgTypes...);
void reset();
};

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

32.9.10.1 Member functions

packaged_task() noexcept;

Effects: Constructs a packaged_task object with The object has no shared state and no stored task.

template<class F>
packaged_task(F&& f);

Constraints: remove_cvref_t<F> is not the same type as packaged_task<R(ArgTypes...)>

Requires: invokes<R>({t_1, t_2, ..., t_N}) (??), where t_1, t_2, ..., t_N are values of the corresponding types in ArgTypes..., shall be a valid expression. Invoking a copy of f shall behave the same as invoking f.

Mandates: is_invocable_r_v<R, F&, ArgTypes...> is true.

Expects: Invoking a copy of f behaves the same as invoking f.

Remarks: This constructor shall not participate in overload resolution if remove_cvref_t<F> is the same type as packaged_task<R(ArgTypes...)>

Effects: Constructs a new packaged_task object with a shared state and initializes the object's stored task with std::forward<F>(f).

Throws: Any exceptions thrown by the copy or move constructor of f, or bad_alloc if memory for the internal data structures could not be allocated.

packaged_task(packaged_task&& rhs) noexcept;

Effects: Constructs a new packaged_task object and transfers Transfers ownership of rhs's shared state to *this, leaving rhs with no shared state. Moves the stored task from rhs to *this.

Ensures: rhs has no shared state.

packaged_task& operator=(packaged_task&& rhs) noexcept;

Effects:
— Releases any shared state (32.9.5);
— calls packaged_task(std::move(rhs)).swap(*this).

~packaged_task();

Effects: Abandons any shared state (32.9.5).

void swap(packaged_task& other) noexcept;

Effects: Exchanges the shared states and stored tasks of *this and other.

Ensures: *this has the same shared state and stored task (if any) as other prior to the call to swap. other has the same shared state and stored task (if any) as *this prior to the call to swap.

bool valid() const noexcept;

Returns: true only if *this has a shared state.
future<R> get_future();

Returns: A future object that shares the same shared state as *this.

Synchronization: Calls to this function do not introduce data races (??) with calls to operator() or make_ready_at_thread_exit. [Note: Such calls need not synchronize with each other. — end note] 

Throws: A future_error object if an error occurs.

Error conditions:
(18.1) — future_already_retrieved if get_future has already been called on a packaged_task object with the same shared state as *this.
(18.2) — no_state if *this has no shared state.

void operator()(ArgTypes... args);

Effects: As if by INVOKE<R>(f, t1, t2, ..., tN) (??), where f is the stored task of *this and t1, t2, ..., tN are the values in args.... If the task returns normally, the return value is stored as the asynchronous result in the shared state of *this, otherwise the exception thrown by the task is stored. The shared state of *this is made ready, and any threads blocked in a function waiting for the shared state of *this to become ready are unblocked.

Throws: A future_error exception object if there is no shared state or the stored task has already been invoked.

Error conditions:
(21.1) — promise_already_satisfied if the stored task has already been invoked.
(21.2) — no_state if *this has no shared state.

void make_ready_at_thread_exit(ArgTypes... args);

Effects: As if by INVOKE<R>(f, t1, t2, ..., tN) (??), where f is the stored task and t1, t2, ..., tN are the values in args.... If the task returns normally, the return value is stored as the asynchronous result in the shared state of *this, otherwise the exception thrown by the task is stored. In either case, this shall be done without making that state ready (32.9.5) immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws: future_error if an error condition occurs.

Error conditions:
(24.1) — promise_already_satisfied if the stored task has already been invoked.
(24.2) — no_state if *this has no shared state.

void reset();

Effects: As if *this = packaged_task(std::move(f)), where f is the task stored in *this. [Note: This constructs a new shared state for *this. The old state is abandoned (32.9.5). — end note]

Throws:
(26.1) — bad_alloc if memory for the new shared state could not be allocated.
(26.2) — any exception thrown by the move constructor of the task stored in the shared state.
(26.3) — future_error with an error condition of no_state if *this has no shared state.

32.9.10.2Globals [futures.task.nonmembers]

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

Effects: As if by x.swap(y).