Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 P0660R10
Date: 2019-07-19
Reply to: Nicolai Josuttis (nico@josuttis.de),
Lewis Baker (lbaker@fb.com)
Billy O'Neal (bion@microsoft.com)
Herb Sutter (hsutter@microsoft.com),
Anthony Williams (anthony@justsoftwaresolutions.co.uk)
Audience: SG1, LEWG, LWG
Prev. Version: www.wg21.link/P0660R9, www.wg21.link/P1287R0

Stop Token and Joining Thread, Rev 10

New in R10

- `request_stop()` returns whether the stop state was changed.
- `stop_callback` has type member `callback_type`.
- `stop_callback` constructor supports copying/conversion of callback.
- `stop_callback` deduction guide now deduces to decayed type.
- Class `jthread` is now part of header `<thread>`.
- Copy missing definitions from class `std::thread` to class `std::jthread`.
- Extend cross references to `std::thread`.
- Several (other) fixes from LWG multiple reviews.
- Several editorial fixes.

New in R9

- Fixes as requested by LEWG in Kona 2/2019.
  - clarify constness of get `stop_stop_source()` and `get_stop_token()`
- Fixes as requested from wording review by SG1 in Kona 2/2019.
  - clarify concurrency issues
- Fixes of small errors and typos.

New in R8

As requested at the LEWG meeting in San Diego 2018:

- Terminology (especially rename `interrupt_token` to `stop_token`).
- Add a deduction guide for `stop_callback`
- Add `std::nostopstate_t` to create stop tokens that don't share a stop state
- Make comparisons hidden friends
- Several clarifications in wording

New in R7

- Adopt www.wg21.link/P1287 as discussed in the SG1 meeting in San Diego 2018, which includes:
  - Add callbacks for interrupt tokens.
  - Split into `interrupt_token` and `interrupt_source`.

New in R6

- User `condition_variable_any` instead of `consition_variable` to avoid all possible races, deadlocks, and unintended undefined behavior.
- Clarify future binary compatibility for interrupt handling (mention requirements for future callback support and allow `bad_alloc` exceptions on waits.

New in R5

As requested at the SG1 meeting in Seattle 2018:

— Removed exception class `std::interrupted` and the `throw_if_interrupted()` API.

— Removed all TLS extensions and extensions to `std::this_thread`.

— Added support to let `jhread` call a callable that either takes the interrupt token as additional first argument or doesn't get it (taking just all passed arguments).

New in R4

— Removed interruptible CV waiting members that don't take a predicate.

— Removed adding a new `cv_status` value `interrupted`.

— Added CV members for interruptible timed waits.

— Renamed CV members that wait interruptible.

— Several minor fixes (e.g. on `noexcept`) and full proposed wording.

Purpose

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

For a full discussion fo the motivation, see `www.wg21.link/p0660r0` and `www.wg21.link/p0660r1`.

A default implementation exists at: `http://github.com/josuttis/jthread`. Note that the proposed functionality can be fully implemented on top of the existing C++ standard library without special OS support.

Basis examples

Basis `jthead` examples

— At the end of its lifetime a `jthread` automatically signals a request to stop the started thread (if still joinable) and joins:

```
void testJThreadWithToken()
{
    std::jthread t([] (std::stop_token stoken) {
                     while (!stoken.stop_requested()) {
                       //...
                     }
                   });
    //...
} // jthread destructor signals requests to stop and therefore ends the started thread and joins
```

The stop could also be explicitly requested with `t.request_stop()`.

— If the started thread doesn't take a stop token, the destructor still has the benefit of calling `join()` (if still joinable):

```
void testJThreadJoining()
{
    std::jthread t([] {
                     //...
                   });
    //...
} // jthread destructor calls join()
```

This is a significant improvement over `std::thread` where you had to program the following to get the same behavior (which is common in many scenarios):

```
void compareWithStdThreadJoining()
{
    std::thread t([] {
                    //...
                  });
    try {
      //...
    }
```

```
    catch (...) {
      j.join();
      throw;  // rethrow
    }
    t.join();
}
```

— An extended CV API enables to interrupt CV waits using the passed stop token (i.e. interrupting the CV wait without polling):

```
void testInterruptibleCVWait()
{
  bool ready = false;
  std::mutex readyMutex;
  std::condition_variable_any readyCV;
  std::jthread t([&ready, &readyMutex, &readyCV] (std::stop_token st) {
                   while (...) {
                     ...
                     {
                       std::unique_lock lg{readyMutex};
                       readyCV.wait_until(lg,
                                          [&ready] {
                                             return ready;
                                          },
                                          st);  // also ends wait on stop request for st
                     }
                     ...
                   }
                 });
  ...
} // jthread destructor signals stop request and therefore unblocks the CV wait and ends the started thread
```

Basis `istop_source`/`stop_token` examples

```
// create stop_source and stop_token:
std::stop_source ssrc;
std::stop_token stok{ssrc.get_token()};

// register callback
bool cb1called{false};
auto cb1 = [&]{ cb1called = true; };
std::stop_callback scb1{stok, cb1};  // stop_callback copies cb1
assert(!cb1called);

// request stop
ssrc.request_stop();                 // calls all currently registered callbacks
assert(cb1called);

// register another callback
bool cb2called{false};
std::stop_callback scb2{stok,
                        [&]{ cb2called = true; }
                       };            // immediately calls callback (moved into scb2)
assert(cb2called);
```

Feature Test Macro

Update [tab:support.ft] with

```
__cpp_lib_jthread
```

and the corresponding value for the headers `<stop_token>` and `<thread>`.

Design Discussion

Problems with "interrupt"

Earlier versions of this paper used the names `interrupt_token`, `interrupt_source` and `interrupt_callback` to refer to the abstraction used to signal interrupt.

However, the term "interrupt" already has common usage in industry and typically refers to something which can be interrupted and then return back to the non-interrupted state.

For example, hardware interrupts are raised when some event happens and then once the interrupt is handled the system returns back to the non-interrupted state, allowing the interrupt to be raised again.

The boost::thread library also uses the term "interrupt" to refer to an operation that can be raised many times and when the interrupt is handled the state is reset back to non-interrupted.

This is different from the semantics of the abstraction proposed in this paper which has the semantics that once it has been signalled it never returns to the non-signalled state. Thus the term "interrupt" seems inappropriate and is likely to lead to confusion.

Alternative names

There was some discussion in at LEWG at San Diego about alternative names for `interrupt_token` and there were two candidates: `cancellation_token` and `stop_token`.

The term `cancellation_token` has precedent in other C++ libraries. For example, Microsoft's PPL uses the names 'cancellation_token', 'cancellation_token_source' and 'cancellation_registration'.

The use of the "cancel" term also has precedent in the Networking TS which defines methods such as `basic_waitable_timer::cancel()` and `basic_socket::cancel()` and makes use of `std::errc::operation_canceled` as an error code in response to a request to cancel the operation.

However, some concerns were raised about the potential for confusion if a `std::jthread::cancel()` method were added as some may confuse this as somehow being related to the semantics of `pthread_cancel()` which is able to cancel a thread at an arbitrary point rather than cooperatively at well-defined cancellation points.

A straw poll was taken in LEWG at San Diego and the group favoured `stop_token`.

A suggestion was also made to introduce the use of the term "request" to more clearly communicate the asynchronous and cooperative nature of the abstraction. This suggestion has been adopted.

As a result the proposed names for the types and methods are now as follows:

```
class stop_token {
 public:
  ...
  [[nodiscard]] bool stop_requested() const noexcept;
  [[nodiscard]] bool stop_possible() const noexcept;
};

class stop_source {
 public:
  ...
  [[nodiscard]] bool stop_requested() const noexcept;
  [[nodiscard]] bool stop_possible() const noexcept;
  bool request_stop() noexcept;
};

template<Invocable Callback>
class stop_callback {
 public:
  ...
};
```

Callback Registration/Deregistration

An important capability for asynchronous use-cases for `stop_token` is the ability to attach a callback to the `stop_token` that will be called if a request to stop is made. The motivations for this are discussed in more detail in P1287R0.

Registration of a callback is performed by constructing a `stop_callback` object, passing the constructor both a `stop_token` and a `Invocable` object that is invoked if/when a call to `request_stop()` is made.

For example:

```
void cancellable_operation(std::stop_token stoken = {})
{
  auto handle = begin_operation();
  std::stop_callback cb{ stoken, [&] { cancel_operation(handle); }};
  ...
  auto result = end_operation(handle);
}
```

When a `stop_callback` object is constructed, if the `stop_token` has already received a request to stop then the callback is immediately invoked inside the constructor. Otherwise, the callback is registered with the `stop_token` and is later invoked if/when some thread calls `request_stop()` on an associated `stop_source`.

The callback registration is guaranteed to be performed atomically. If there is a concurrent call to `request_stop()` from another thread then either the current thread will see the request to stop and immediately invoke the callback on the current thread or the other thread will see the callback registration and will invoke the callback before returning from `request_stop()`.

When the `stop_callback` object is destructed the callback is deregistered from the list of callbacks associated with the `stop_token`'s shared state the callback is guaranteed not to be called after the `stop_callback` destructor returns.

Note that there is a potential race here between the callback being deregistered and a call to `request_stop()` being made on another thread which could invoke the callback. If the callback has not yet started executing on the other thread then the callback is deregistered and is never called. Otherwise, if the callback has already started executing on another thread then the call to `~stop_callback()` will block the current thread until the callback returns.

If the call to the `stop_callback` destructor is made from within the the invocation of the callback on the same thread then the destructor does not block waiting for the callback to return as this would cause a deadlock. Instead, the destructor returns immediately without waiting for the callback to return.

Other Hints

It is intentional that class `std::jthread` supports the full API of `std::thread` (i.e., by supporting to start the thread without taking a stop token as first parameter) to be able to replace any usage of `std::thread` by `std::jthread` without further code changes.

The terminology was carefully selected with the following reasons

— With a stop token we neither "interrupt" nor "cancel" something. We request a stop that cooperatively has to get handled.

— `stop_possible()` helps to avoid adding new callbacks or checking for stop states. The name was selected to have a common and pretty self-explanatory name that is shared by both `stop_source`s and `stop_token`s.

Initialization of `stop_callback`s

Class `stop_callback` was carefully designed to be able to accept any callback, whether passed as lvalue or rvalue and whether an implicit conversion to a specified callback type is requested.

The deduction guide for `stop_callback`s deduces the callback argument type (`callback_type`) such that the constructor performs a decayed copy of the callback argument. Because the constructors are function templates, type conversions are possible.

Given:

```
struct ImplicitArg {
};
struct ExplicitArg {
};
struct MyCallback {
  MyCallback(ImplicitArg x);
  explicit MyCallback(ExplicitArg x);
  void operator()() const;
};
```

we have the following behavior:

Table 1 — `stop_callback` initializations

Example	Behavior
`auto stop = [] { ... };` `stop_callback cb{token, stop};`	Constructs copy of `stop` lambda in `cb`.
`auto stop = [] { ... };` `stop_callback cb{token, std::move(stop)};`	Moves `stop` into `cb`.
`auto stop = [] { ... };` `stop_callback cb{token, std::ref(stop)};`	Captures lvalue reference to stop in `cb`. Deduces `callback_type` as `std::reference_wrapper<decltype(stop)>`.
`stop_callback cb{token, [] { ... }};`	Move constructs temporary lambda into cb.

`stop_callback<function<void()>> cb{`
`  token, [] { ... }`
`};`
	Directly constructs function member of `cb` from lambda. No temporary/move operation.

`function<void()> f = [] { ... };` `stop_callback cb{token, f};`	Copies `f` into `cb`. Deduces `callback_type` as `function<void()>`.
`function<void()> f = [] { ... };` `stop_callback<function<void()>> cb{token, f};`	Copies `f` into `cb`.

`auto cb = [token] {`
`  function<void()> f;`
`  if (cond) {`
`  f = [] { ... };`
`  } else {`
`  f = [] { ... };`
`  }`
`  return stop_callback{token, f};`
`}();`
	Initializes `cb` with a copy of `f`. Deduces `callback_type` as `function<void()>`.

`ImplicitArg i;` `stop_callback<MyCallback> cb{token, i};`	Directly constructs `myCallback` member of `cb` from `i` by calling implicit constructor.
`ExplicitArg e;` `stop_callback<MyCallback> cb{token, e};`	Directly constructs myCallback member of `cb` from `e` by calling explicit constructor.

`stop_callback<MyCallback> cb =`
`  [&]() -> stop_callback<MyCallback> {`
`  ExplicitArg e;`
`  return {token, e};`
`  }();`
	Ill-formed. Implicit construction not supported.

`stop_callback<MyCallback> cb =`
`  [&]() -> stop_callback<MyCallback> {`
`  ImplicitArg i;`
`  return {token, i};`
`  }();`
	Ill-formed. Implicit construction not supported.

Acknowledgements

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

Proposed Wording

All against N4762.

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

32 Thread support library [thread]

32.1 General [jthread.general]

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

Table 2 — Thread support library summary

	Subclause	Header(s)
32.2	Requirements	
32.3	Stop Tokens	`<stop_token>`
32.4	Threads	`<thread>`
32.5	Mutual exclusion	`<mutex>` `<shared_mutex>`
32.6	Condition variables	`<condition_variable>`
??	Futures	`<future>`

32.2 Requirements [thread.req]

...

32.3 Stop Tokens [thread.stoptoken]

32.3.1 Stop Token Introduction [thread.stoptoken.intro]

1 This clause describes components that can be used to asynchonously 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*.

2 `stop_source`, `stop_token`, and `stop_callback` implement semantics of shared ownership of a *(*stop state). Any `stop_source`, `stop_token` or `stop_callback` that shares ownership of the same stop state is an *(*associated) `stop_source`, `stop_token` or `stop_callback`, respectively. The last remaining owner of the stop state automatically releases the resources associated with the stop state.

3 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 `stop_callback` class template which will be called in the event that a stop request is made.

A stop request made via a `stop_source` will be visible to all associated `stop_token` and `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 `stop_callback` object are called when a stop request is first made by any associated `stop_source` object.

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

32.3.2 Header <stop_token> synopsis [thread.stoptoken.syn]

```
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 stop_token [stoptoken]

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

```
namespace std {
  class stop_token {
  public:
    // 32.3.3.1 create, copy, destroy:
    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;
```

```
    // 32.3.3.5 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 `stop_token` constructors [stoptoken.constr]

```
stop_token() noexcept;
```

1 *Ensures:* `stop_possible()` is `false` and `stop_requested()` is `false`. [*Note*: Because the created `stop_token` object can never receive a stop request, no resources are allocated for a stop state. — *end note*]

```
stop_token(const stop_token& rhs) noexcept;
```

2 *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;
```

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

32.3.3.2 `stop_token` destructor [stoptoken.destr]

```
~stop_token();
```

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

32.3.3.3 `stop_token` assignment [stoptoken.assign]

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

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

2 *Returns:* `*this`.

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

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

4 *Returns:* `*this`.

32.3.3.4 `stop_token` swap [stoptoken.swap]

```
void swap(stop_token& rhs) noexcept;
```

1 *Effects:* Exchanges the values of `*this` and `rhs`.

32.3.3.5 `stop_token` members [stoptoken.mem]

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

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

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

2 *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`.

32.3.3.6 `stop_token` comparisons [stoptoken.cmp]

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

1 *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;
```

2 *Returns:* `!(lhs==rhs)`.

32.3.3.7 Specialized algorithms [stoptoken.special]

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

1 *Effects:* Equivalent to: `x.swap(y)`.

32.3.4 Class `stop_source` [stopsource]

1 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;
  };
  inline constexpr nostopstate_t nostopstate{};

  class stop_source {
  public:
    // 32.3.4.1 create, copy, destroy:
    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;

    // 32.3.4.5 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 `stop_source` constructors [stopsource.constr]

```
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`.

32.3.4.2 `stop_source` destructor [stopsource.destr]

```
~stop_source();
```

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

32.3.4.3 `stop_source` assignment [stopsource.assign]

```
stop_source& operator=(const stop_source& rhs) noexcept;
```

1 *Effects:* Equivalent to: `stop_source(rhs).swap(*this)`.

2 *Returns:* `*this`.

```
stop_source& operator=(stop_source&& rhs) noexcept;
```

3 *Effects:* Equivalent to: `stop_source(std::move(rhs)).swap(*this)`.

4 *Returns:* `*this`.

32.3.4.4 `stop_source` swap [stopsource.swap]

```
void swap(stop_source& rhs) noexcept;
```

1 *Effects:* Exchanges the values of `*this` and `rhs`.

32.3.4.5 `stop_source` members [stopsource.mem]

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

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

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

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

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

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

```
bool request_stop() noexcept;
```

4 *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*]

5 *Ensures:* `stop_possible()` is `false` or `stop_requested()` is `true`.

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

32.3.4.6 `stop_source` comparisons [stopsource.cmp]

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

1 *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;
```

2 *Returns:* `!(lhs==rhs)`.

32.3.4.7 Specialized algorithms [stopsource.special]

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

1 *Effects:* Equivalent to: `x.swap(y)`.

32.3.5 Class Template `stop_callback` [stopcallback]

1

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

    // 32.3.5.1 create, destroy:
    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;

  private:
    Callback callback;      // exposition only
  };

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

2 *Mandates:* `stop_callback` is instantiated with an argument for the template parameter `Callback` that satisfies both `Invocable` and `Destructible`.

3 *Expects:* `stop_callback` is instantiated with an argument for the template parameter `Callback` that models both `Invocable` and `Destructible`.

32.3.5.1 `stop_callback` constructors and destructor [stopcallback.constr]

```
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>);
```

1 *Constraints:* `Callback` and `C` satisfy `Constructible<Callback, C>`.

2 *Expects:* `Callback` and `C` model `Constructible<Callback, C>`.

3 *Effects:* Initializes `callback` with `static_cast<C&&>(cb)`. If `st.stop_requested()` is `true`, `static_cast<Callback&&>(callback)()` is evaluated in the current thread before the constructor returns. Otherwise, if `st` has ownership of a stop state, acquires shared ownership of that stop state and registers the callback with that stop state such that `static_cast<Callback&&>(callback)()` is evaluated by the first call to `request_stop()` on an associated `stop_source`.

4 *Remarks:* If evaluating `static_cast<Callback&&>(callback)()` exits via an exception, then `terminate()` is called.

5 *Throws:* Any exception thrown by the initialization of `callback`.

`~stop_callback();`

6 *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 `stop_callback`. If `callback` is concurrently executing on another thread, then the return from the invocation of `callback` strongly happens before (**??**) `callback` is destroyed. If `callback` is executing on the current thread, then the destructor does not block (**??**) waiting for the return from the invocation of `callback`. Releases ownership of the stop state, if any.

32.4 Threads [thread.threads]

...

32.4.1 Header <thread> synopsis [thread.syn]

```
namespace std {
  class thread;

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

  // 32.4.3 class jthread
  class jthread;

  ...
}
```

32.4.2 Class thread [thread.thread.class]

...

32.4.3 Class jthread [thread.jthread.class]

1 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 ability to request that the thread stops and then joins the started thread.

[*Editorial note:* This color signals differences in behavior compared to class `std::thread`]

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

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

    // members
    void swap(jthread&) noexcept;

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

    void join();
    void detach();
    [[nodiscard]] id get_id() const noexcept;
    [[nodiscard]] native_handle_type native_handle();     // see ??

    // stop token handling
    [[nodiscard]] stop_source get_stop_source() noexcept;
    [[nodiscard]] stop_token get_stop_token() const noexcept;
    bool request_stop() noexcept;

    friend void swap(jthread& lhs, jthread& rhs) noexcept;

    // static members
    [[nodiscard]] static unsigned int hardware_concurrency() noexcept;

  private:
    stop_source ssource;                // exposition only
  };
}
```

32.4.3.1 `jthread` constructors [thread.jthread.constr]

```
jthread() noexcept;
```

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

2 *Ensures:* `get_id() == id()` is `true` and `ssource.stop_possible()` is `false`.

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

3 *Requires:* `F` and each T_i in `Args` shall satisfy the *Cpp17MoveConstructible* requirements. *INVOKE*(*decay-copy*(std::forward<F>(f)), ssource.get_token(), *decay-copy*(std::forward<Args>(args))...) or *INVOKE*(*decay-copy*(std::forward<F>(f)), *decay-copy*(std::forward<Args>(args))...) (??) are valid expressions.

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

5 *Effects:* Initializes `ssource` and constructs an object of type `jthread`. The new thread of execution executes *INVOKE*(*decay-copy*(std::forward<F>(f)), ssource.get_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.

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

7 *Ensures:* `get_id() != id()` is `true` and `ssource.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*]

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

9 *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;
```

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

11 *Ensures:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction. `ssource` has the value of `x.ssource` prior to the start of construction and `x.ssource.stop_possible()` is `false`.

32.4.3.2 `jthread` destructor [thread.jthread.destr]

```
~jthread();
```

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

32.4.3.3 `jthread` assignment [thread.jthread.assign]

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

1 *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.

2 *Ensures:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment. `ssource` has the value of `x.ssource` prior to the assignment and `x.ssource.stop_possible()` is `false`.

3 *Returns:* `*this`.

32.4.3.4 Members [thread.jthread.member]

```
void swap(jthread& x) noexcept;
```

1 *Effects:* Exchanges the values of `*this` and `x`.

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

2 *Returns:* `get_id() != id()`.

```
void join();
```

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

4 *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*]

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

6 *Throws:* `system_error` when an exception is required (**??**).

7 *Error conditions:*

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

(7.2) — `no_such_process` — if the thread is not valid.

(7.3) — `invalid_argument` — if the thread is not joinable.

```
void detach();
```

8 *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.

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

10 *Throws:* `system_error` when an exception is required (**??**).

11 *Error conditions:*

(11.1) — `no_such_process` — if the thread is not valid.

(11.2) — `invalid_argument` — if the thread is not joinable.

```
id get_id() const noexcept;
```

12 *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`.

32.4.3.5 `jthread` stop members [thread.jthread.stop]

```
[[nodiscard]] stop_source get_stop_source() noexcept
```

1 *Effects:* Equivalent to: `return ssource;`

```
[[nodiscard]] stop_token get_stop_token() const noexcept
```

2 *Effects:* Equivalent to: `return ssource.get_token();`

```
bool request_stop() noexcept;
```

3 *Effects:* Equivalent to: `return ssource.request_stop();`

32.4.3.6 Specialized algorithms [thread.jthread.special]

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

1 *Effects:* Equivalent to: `x.swap(y)`.

32.4.3.7 Static members [thread.jthread.static]

```
unsigned hardware_concurrency() noexcept;
```

1 *Returns:* `thread::hardware_concurrency()`.

32.5 Mutual exclusion [thread.mutex]

...

32.6 Condition variables [thread.condition]

...

32.6.1 Header `<condition_variable>` synopsis [condition.variable.syn]

...

32.6.2 Non-member functions [thread.condition.nonmember]

...

32.6.3 Class `condition_variable` [thread.condition.condvar]

...

32.6.4 Class `condition_variable_any` [thread.condition.condvarany]

...

```
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);

  };
}
```

```
condition_variable_any();
```

1 *Effects:* Constructs an object of type `condition_variable_any`.

2 *Throws:* `bad_alloc` or `system_error` when an exception is required (**??**).

3 *Error conditions:*

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

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

```
~condition_variable_any();
```

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

5 *Effects:* Destroys the object.

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

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

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

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

32.6.4.1 Noninterruptible waits [thread.condvarany.wait]

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

1 *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 requested by a call to `notify_one()`, a call to `notify_all()`, or spuriously.

2 *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*]

3 *Ensures:* `lock` is locked by the calling thread.

4 *Throws:* Nothing.

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

5 *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);
```

6 *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 requested by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (**??**) 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.

7 *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*]

8 *Ensures:* `lock` is locked by the calling thread.

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

10 *Throws:* Timeout-related exceptions (**??**).

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

11 *Effects:* Equivalent to:

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

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

13 *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*]

14 *Ensures:* `lock` is locked by the calling thread.

15 *Throws:* Timeout-related exceptions (**??**).

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

16 *Effects:* Equivalent to:

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

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

18 [*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);
```

19 *Effects:* Equivalent to:

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

32.6.4.2 Interruptible waits [thread.condvarany.interruptwait]

1 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);
```

2 *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.stop_requested()) {
  if (pred())
    return true;
  wait(lock);
}
return pred();
```

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

4 *Ensures:* `lock` is locked by the calling thread.

5 *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*]

6 *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);
```

7 *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.stop_requested()) {
  if (pred())
    return true;
  if (cv.wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
}
return pred();
```

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

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

10 *Ensures:* `lock` is locked by the calling thread.

11 *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*]

12 *Throws:* Timeout-related exceptions (**??**), 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);
```

13 *Effects:* Equivalent to:

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

32.7 Other Fixes []

In 6.8.2.2 Forward progress [intro.progress]/8:

It is implementation-defined whether the implementation-created thread of execution that executes main (6.8.3.1) and the threads of execution created by `std::thread` (32.3.2) or `std::jthread` (32.4.3) provide concurrent forward progress guarantees.

In 16.5.1.2 Headers [headers] in tab:headers.cpp add:

`<stop_token>`

In 17.12.3.4 Resumption [coroutine.handle.resumption]/1:

Resuming a coroutine via resume, operator(), or destroy on an execution agent other than the one on which it was suspended has implementation-defined behavior unless each execution agent either is an instance of `std::thread` or `std::jthread`, or is the thread that executes `main`.

In 25.3.3 Effect of execution policies on algorithm execution [algorithms.parallel.exec]/6:

The invocations of element access functions in parallel algorithms invoked with an execution policy object of type execution::parallel_policy are permitted to execute either in the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` (32.3.2) or `jthread` (32.4.3) provide concurrent forward progress guarantees (6.8.2.2), then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined.

In 26.3.1 Header <cfenv> synopsis [cfenv.syn]/2:

The floating-point environment has thread storage duration (6.6.5.2). The initial state for a threads floating-point environment is the state of the floating-point environment of the thread that constructs the corresponding `thread` object (32.3.2) or `jthread` object (32.4.3) at the time it constructed the object.