TECHNICAL SPECIFICATION

ISO/IEC TS 19571

First edition
2016-02-01

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

Langages de programmation — Spécification technique pour C ++ Extensions pour la concurrence

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

Reference number
ISO/IEC TS 19571:2016(E)

© ISO/IEC 2016

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2016, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2. www.iso.org/directives

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received. www.iso.org/patents

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC1.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

© ISO/IEC 2016 — All rights reserved

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

1 General [general]

1.1 Namespaces, headers, and modifications to standard classes [general.namespaces]

1 Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:

- modify an existing interface in the C++ Standard Library in-place,
- are declared in a namespace whose name appends `::experimental::concurrency_v1` to a namespace defined in the C++ Standard Library, such as `std`, or
- are declared in a subnamespace of a namespace described in the previous bullet, whose name is not the same as an existing subnamespace of namespace `std`.

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

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

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

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

```
#include <meow>
```

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

Table 1 — C++ library headers

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

1.2 Future plans (Informative) [general.plans]

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

2 The C++ committee intends to release a new version of this technical specification approximately every year, containing the library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::concurrency_v2`, `std::experimental::concurrency_v3`, etc., with the most recent implemented version inlined into `std::experimental`.

3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::concurrency_vN` segment of its namespace and by removing the `experimental/` prefix from its header's path.

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

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

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

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

Table 2 — Significant features in this technical specification

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

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

2.1 General [futures.general]

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

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

```
#include <future>

namespace std {
  namespace experimental {
  inline namespace concurrency_v1 {

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

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

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

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

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

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

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

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

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

    template <class Sequence>
    struct when_any_result;

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

  } // namespace concurrency_v1
  } // namespace experimental

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

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

} // namespace std
```

2.3 Class template `future` [futures.unique_future]

iTeh STANDARD PREVIEW
(standards.iteh.ai)

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

ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

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

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

      // retrieving the value
      see below get();

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

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

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

  };

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

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

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

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

4 *Postconditions:*

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)

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

ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510dc317b/iso-iec-ts-19571-2016

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

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

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

— When the object's shared state is ready, the continuation `INVOKE(DECAY_COPY(std::forward<F>(func)), std::move(*this))` is called on an unspecified thread of execution with the call to `DECAY_COPY()` being evaluated in the thread that called `then`.

— Any value returned from the continuation is stored as the result in the shared state of the resulting `future`. Any exception propagated from the execution of the continuation is stored as the exceptional result in the shared state of the resulting `future`.

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

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)

— *end example*]
— *end note*]

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

ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d8a10550-d4f5-45ee-8410-4ca510de3176/iso-iec-ts-19571-2016

11 `bool is_ready() const;`

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

2.4 Class template `shared_future` [futures.shared_future]

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

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

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

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

      // retrieving the value
      see below get();

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

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

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

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

iTeh STANDARD PREVIEW
(standards.iteh.ai)

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

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

ISO/IEC TS 19571:2016
https://standards.iteh.ai/catalog/standards/sist/d6a16558-a4fb-49ec-b4f6-4ca510dc317b/iso-iec-ts-19571-2016

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

4 *Postconditions:*

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

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