Document Number: D3904
Date: 2014-02-06

Reply to: Artur Laksberg
Microsoft Corporation
arturl@microsoft.com

Working Draft, Technical Specification for C++ Extensions for Concurrency

Contents

1 General [general]

1.1 Scope [general.scope]

This Technical Specification describes requirements for the implementation of a number of concurrency extensions that can be used in computer programs written in the C++ programming language. The extensions described by this Technical Specification are realizable across a broad class of computer architectures.

This Technical Specification is non-normative. Some of the functionality described by this Technical Specification may be considered for standardization in a future version of C++, but it is not currently part of any C++ standard. Some of the functionality in this Technical Specification may never be standardized, and other functionality may be standardized in a substantially changed form.

The goal of this Technical Specification is to enhance the existing practice for concurrency in the C++ standard algorithms library. It gives advice on extensions to those vendors who wish to provide them.

1.2 Normative references [general.references]

The following reference document is indepensible for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

– ISO/IEC 14882:2011, Programming Languages – C++

ISO/IEC 14882:2011 is herein called the C++ Standard. The library described in ISO/IEC 14882:2011 clauses 17-30 is herein called the C++ Standard Library.

Unless otherwise specified, the whole of the C++ Standard Library introduction [lib.library] is included into this Technical Specification by reference.

1.3 Namespaces and headers [general.namespaces]

Some of the extensions described in this Technical Specification represent types and functions that are currently not part of the C++ Standards Library, and because these extensions are experimental, they should not be declared directly within namespace `std`. Instead, such extensions are declared in namespace `std::experimental`.

[*Note:* Once standardized, these components are expected to be promoted to namespace `std`. – *end note*]

Unless otherwise specified, references to such entities described in this Technical Specification are assumed to be qualified with `std::experimental`, and references to entities described in the C++ Standard Library are assumed to be qualified with `std::`.

1.4 Terms and definitions [general.defns]

For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

2 Executors and Schedulers [exec]

2.1 General [exec.general]

This proposal includes two abstract base classes, `executor` and `scheduled_executor` (the latter of which inherits from the former); several concrete classes that inherit from `executor` or `scheduled_executor`; and several utility functions.

Executors library summary

Subclause	Header(s)
V.1 [executors.base]	`<executor>`
V.2 [executors.classes]	
V.2.1 [executors.classes.thread_pool]	`<thread_pool>`
V.2.2 [executors.classes.serial]	`<serial_executor>`
V.2.3 [executors.classes.loop]	`<loop_executor>`
V.2.4 [executors.classes.inline]	`<inline_executor>`
V.2.5 [executors.classes.thread]	`<thread_executor>`

2.2 V.1 Executor base classes [executors.base]

The `<executor>` header defines abstract base classes for executors, as well as non-member functions that operate at the level of those abstract base classes.

Header `<executor>` synopsis

```
class executor;
class scheduled_executor;
```

2.2.1 V.1.1 Class executor [executors.base.executor]

Class `executor` is an abstract base class defining an abstract interface of objects that are capable of scheduling and coordinating work submitted by clients. Work units submitted to an executor may be executed in one or more separate threads. Implementations are required to avoid data races when work units are submitted concurrently.

All closures are defined to execute on some thread, but which thread is largely unspecified. As such accessing a `thread_local` variable is defined behavior, though it is unspecified which thread's `thread_local` will be accessed.

The initiation of a work unit is not necessarily ordered with respect to other initiations. [*Note:* Concrete executors may, and often do, provide stronger initiation order guarantees. Users may, for example, obtain serial execution guarantees by using the `serial_executor` wrapper.– *end note*] There is no defined ordering of the execution or completion of closures added to the executor. [*Note:* The consequence is that closures should not wait on other closures executed by that executor. Mutual exclusion for critical sections is fine, but it can't be used for signalling between closures. Concrete executors may provide stronger execution order guarantees.– *end note*]

```
class executor {
```

```
public:
    virtual ~executor();
    virtual void add(function<void()> closure) =0;
    virtual size_t uninitiated_task_count() const =0;
};
```

```
executor::~executor()
```

Effects: Destroys the executor.

Synchronization: All closure initiations happen before the completion of the executor destructor. [*Note:* This means that closure initiations don't leak past the executor lifetime, and programmers can protect against data races with the destruction of the environment. There is no guarantee that all closures that have been added to the executor will execute, only that if a closure executes it will be initiated before the destructor executes. In some concrete subclasses the destructor may wait for task completion and in others the destructor may discard uninitiated tasks. – *end note*]

Remark: If an executor is destroyed inside a closure running on that executor object, the behavior is undefined. [*Note:* one possible behavior is deadlock. – *end note*]

```
void executor::add(std::function<void> closure);
```

Effects: The specified function object shall be scheduled for execution by the executor at some point in the future. May throw exceptions if add cannot complete (due to shutdown or other conditions).

Synchronization: completion of closure on a particular thread happens before destruction of that thread's thread-duration variables. [*Note:* The consequence is that closures may use thread-duration variables, but in general such use is risky. In general executors don't make guarantees about which thread an individual closure executes in. – *end note*]

Error conditions: The invoked closure should not throw an exception.

```
size_t executor::uninitiated_task_count();
```

Returns: the number of function objects waiting to be executed. [*Note:* this is intended for logging/debugging and for coarse load balancing decisions. Other uses are inherently risky because other threads may be executing or adding closures.– *end note*]

2.2.2 V.1.2 Class `scheduled_executor` [executors.base.scheduled_executor]

Class `scheduled_executor` is an abstract base class that extends the executor interface by allowing clients to pass in work items that will be executed some time in the future.

```
class scheduled_executor : public executor {
public:
    virtual void add_at(const chrono::system_clock::time_point& abs_time,
                        function<void()> closure) = 0;
    virtual void add_after(const chrono::system_clock::duration& rel_time,
                           function<void()> closure) = 0;
};
```

```
void add_at(const chrono::system_clock::time_point& abs_time,
 function<void()> closure);
```

Effects: The specified function object shall be scheduled for execution by the executor at some point in the future no sooner than the time represented by `abs_time`.

Synchronization: completion of closure on a particular thread happens before destruction of that thread's thread-duration variables.

Error conditions: The invoked closure should not throw an exception.

```
void add_after(const chrono::system_clock::duration& rel_time, function<void()> closure);
```

Effects: The specified function object shall be scheduled for execution by the executor at some point in the future no sooner than time `rel_time` from now.

Synchronization: completion of closure on a particular thread happens before destruction of that thread's thread-duration variables.

Error conditions: The invoked closure should not throw an exception.

2.3 V.2 Concrete executor classes [executors.classes]

This section defines executor classes that encapsulate a variety of closure- execution policies.

2.3.1 V.2.1 Class `thread_pool` [executors.classes.thread__pool]

Header <thread_pool> synopsis

```
class thread_pool;
```

Class `thread_pool` is a simple thread pool class that creates a fixed number of threads in its constructor and that multiplexes closures onto them.

```
class thread_pool : public scheduled_executor {
   public:
   explicit thread_pool(int num_threads);
   ~thread_pool();
   // [executor methods omitted]
};
```

```
thread_pool::thread_pool(int num_threads)
```

Effects: Creates an executor that runs closures on `num_threads` threads.

Throws: `system_error` if the threads can't be created and `started.thread_pool::~thread_pool()`

```
thread_pool::~thread_pool()
```

Effects: Waits for closures (if any) to complete, then joins and destroys the threads.

2.3.2 V.2.2 Class `serial_executor` [executors.classes.serial]

Header `<serial_executor>` synopsis

```
class serial_executor;
```

Class `serial_executor` is an adaptor that runs its closures by scheduling them on another (not necessarily single-threaded) executor. It runs added closures inside a series of closures added to an underlying executor in such a way so that the closures execute serially. For any two closures `c1` and `c2` added to a `serial_executor` `e`, either the completion of `c1` happens before (1.10 [intro.multithread]) the execution of `c2` begins, or vice versa. If `e.add(c1)` happens before `e.add(c2)`, then `c1` is executed before `c2`.

The number of `add()` calls on the underlying executor is unspecified, and if the underlying executor guarantees an ordering on its closures, that ordering won't necessarily extend to closures added through a `serial_executor`. [*Note:* this is because serial_executor can batch add() calls to the underlying executor. – *end note*]

```
class serial_executor : public executor {
public
    explicit serial_executor(executor& underlying_executor);
    virtual ~serial_executor();
    executor& underlying_executor();
    // [executor methods omitted]
};
```

```
serial_executor::serial_executor(executor& underlying_executor)
```

Requires: `underlying_executor` shall not be null.

Effects: Creates a `serial_executor` that executes closures in FIFO order by passing them to `underlying_executor`. [*Note:* several `serial_executor` objects may share a single underlying executor. – *end note*]

```
serial_executor::~serial_executor()
```

Effects: Finishes running any currently executing closure, then destroys all remaining closures and returns. If a `serial_executor` is destroyed inside a closure running on that `serial_executor` object, the behavior is undefined. [*Note:* one possible behavior is deadlock. – *end note*]

```
executor& serial_executor::underlying_executor()
```

Returns: The underlying executor that was passed to the constructor.

2.4 V.2.3 Class loop_executor [executors.classes.loop]

Header `<loop_executor>` synopsis

```
class loop_executor;
```

Class `loop_executor` is a single-threaded executor that executes closures by taking control of a host thread. Closures are executed via one of three closure- executing methods: `loop()`, `run_queued_closures()`, and `try_run_one_closure()`. Closures are executed in FIFO order. Closure-executing methods may not be called concurrently with each other, but may be called concurrently with other member functions.

```
class loop_executor : public executor {
public:
    loop_executor();
    virtual ~loop_executor();
    void loop();
    void run_queued_closures();
    bool try_run_one_closure();
    void make_loop_exit();
    // [executor methods omitted]
};
```

`loop_executor::loop_executor()`

Effects: Creates a `loop_executor` object. Does not spawn any threads.

`loop_executor::~loop_executor()`

Effects: Destroys the `loop_executor` object. Any closures that haven't been executed by a closure-executing method when the destructor runs will never be executed.

Synchronization: Must not be called concurrently with any of the closure-executing methods.

`void loop_executor::loop()`

Effects: Runs closures on the current thread until `make_loop_exit()` is called.

Requires: No closure-executing method is currently running.

`void loop_executor::run_queued_closures()`

Effects: Runs closures that were already queued for execution when this function was called, returning either when all of them have been executed or when `make_loop_exit()` is called. Does not execute any additional closures that have been added after this function is called. Invoking `make_loop_exit()` from within a closure run by `run_queued_closures()` does not affect the behavior of subsequent closure-executing methods. [*Note:* this requirement disallows an implementation like

```
void run_queued_closures() {
    add([](){make_loop_exit();});
    loop();
}
```

because that would cause early exit from a subsequent invocation of `loop()`. – *end note*]

Requires: No closure-executing method is currently running.

Remarks: This function is primarily intended for testing.

`bool loop_executor::try_run_one_closure()`

Effects: If at least one closure is queued, this method executes the next closure and returns.

Returns: `true` if a closure was run, otherwise `false`.

Requires: No closure-executing method is currently running.

Remarks: This function is primarily intended for testing.

```
void loop_executor::make_loop_exit()
```

Effects: Causes `loop()` or `run_queued_closures()` to finish executing closures and return as soon as the current closure has finished. There is no effect if `loop()` or `run_queued_closures()` isn't currently executing. [*Note:* `make_loop_exit()` is typically called from a closure. After a closure- executing method has returned, it is legal to call another closure-executing function. – *end note*]

2.5 V.2.4 Class inline_executor [executors.classes.inline]

Header `<inline_executor>` synopsis

```
class inline_executor;
```

Class `inline_executor` is a simple executor which intrinsically only provides the `add()` interface as it provides no queuing and instead immediately executes work on the calling thread. This is effectively an adapter over the executor interface but keeps everything on the caller's context.

```
class inline_executor : public executor {
public
    explicit inline_executor();
    // [executor methods omitted]
};
```

```
inline_executor::inline_executor()
```

Effects: Creates a dummy executor object which only responds to the `add()` call by immediately executing the provided function in the caller's thread.

2.6 V.2.5 Class thread_executor [executors.classes.thread]

Header `<thread_executor>` synopsis

```
class thread_executor;
```

Class `thread_executor` is a simple executor that executes each task (closure) on its own `std::thread` instance.

```
class thread_executor : public executor {
public:
    explicit thread_executor();
    ~thread_executor();
    // [executor methods omitted]
};
```

```
thread_executor::thread_executor()
```

Effects: Creates an executor that runs each closure on a separate thread.

```
thread_executor::~thread_executor()
```

Effects: Waits for all added closures (if any) to complete, then joins and destroys the threads.

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

3.1 General [futures.general]

The extensions proposed here are an evolution of the functionality of `std::future` and `std::shared_future`. The extensions enable wait free composition of asynchronous operations.

3.2 30.6.6 Class template `future` [futures.unique-future]

To the class declaration found in 30.6.6/3, add the following to the public functions:

```
bool ready() const;
future(future<future<R>>&& rhs) noexcept;

template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;

template<typename R2>
future<R2> future<R>::unwrap();
```

Between 30.6.6/8 & 30.6.6/9, add the following:

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

Effects: constructs a `future` object by moving the instance referred to by `rhs` and unwrapping the inner future (see `unwrap()`).

Postconditions:

- `valid()` returns the same value as `rhs.valid()` prior to the constructor invocation.
- `rhs.valid() == false`.

After 30.6.6/24, add the following:

```
template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

Notes: The three functions differ only by input parameters. The first only takes a callable object which accepts a `future` object as a parameter. The second function takes an `executor` as the first parameter and a callable object as the second parameter. The third function takes a launch policy as the first parameter and a callable object as the second parameter. In cases where `decltype(func(*this))` is `future<R>`, the resulting type is `future<R>` instead of `future<future<R>>`.

Effects:

- The continuation is called when the object's shared state is ready (has a value or exception stored).
- The continuation launches according to the specified launch policy or executor.
- When the executor or launch policy is not provided the continuation inherits the parent's launch policy or executor.
- If the parent was created with `std::promise` or with a `packaged_task` (has no associated launch policy), the continuation behaves the same as the third overload with a policy argument of `launch::async | launch::deferred` and the same argument for `func`.
- If the parent has a policy of `launch::deferred` and the continuation does not have a specified launch policy or scheduler, then the parent is filled by immediately calling `wait()`, and the policy of the antecedent is `launch::deferred`

Returns: An object of type `future<decltype(func(*this))>` that refers to the shared state created by the continuation.

Postconditions:

- The `future` object is moved to the parameter of the continuation function
- `valid() == false` on original `future` object immediately after it returns

```
template<typename R2>
future<R2> future<R>::unwrap()
```

Notes:

- `R` is a `future<R2>` or `shared_future<R2>`
- Removes the outer-most future and returns a proxy to the inner future. The proxy is a representation of the inner future and it holds the same value (or exception) as the inner future.

Effects:

- `future<R2> X = future<future<R2>>.unwrap()`, returns a `future<R2>` that becomes ready when the shared state of the inner future is ready. When the inner future is ready, its value (or exception) is moved to the shared state of the returned future.
- `future<R2> Y = future<shared_future<R2>>.unwrap()`, returns a `future<R2>` that becomes ready when the shared state of the inner future is ready. When the inner `shared_future` is ready, its value (or exception) is copied to the shared state of the returned future.
- If the outer future throws an exception, and `get()` is called on the returned future, the returned future throws the same exception as the outer future. This is the case because the inner future didn't exit

Returns: a `future` of type `R2`. The result of the inner `future` is moved out (`shared_future` is copied out) and stored in the shared state of the returned future when it is ready or the result of the inner future throws an exception.

Postcondition: The returned future has `valid() == true`, regardless of the *validity of the inner future.

[*Example:*

```
future<int> work1(int value);
int work(int value) {
    future<future<int>> f1 = std::async([=] {return work1(value); });
    future<int> f2 = f1.unwrap();
    return f2.get();
}
```

– *end example*]

```
bool is_ready() const;
```

Returns: `true` if the shared state is ready, `false` if it isn't.

3.3 30.6.7 Class template `shared_future` [futures.shared_future]

To the class declaration found in 30.6.7/3, add the following to the public functions:

```
bool is_ready() const;
template<typename F>

auto then(F&& func) -> future<decltype(func(*this))>;
template<typename F>

auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;
template<typename F>

auto then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

After 30.6.7/26, add the following:

```
template<typename F>
auto shared_future::then(F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto shared_future::then(executor &ex, F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto shared_future::then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

Notes: The three functions differ only by input parameters. The first only takes a callable object which accepts a `shared_future` object as a parameter. The second function takes an `executor` as the first parameter and a callable object as the second parameter. The third function takes a launch policy as the first parameter and a callable object as the second parameter.

In cases where `decltype(func(*this))` is `future<R>`, the resulting type is `future<R>` instead of `future<future<R>>`.

Effects:

- The continuation is called when the object's shared state is ready (has a value or exception stored).
- The continuation launches according to the specified policy or executor.

- When the scheduler or launch policy is not provided the continuation inherits the parent's launch policy or executor.
- If the parent was created with `std::promise` (has no associated launch policy), the continuation behaves the same as the third function with a policy argument of `launch::async | launch::deferred` and the same argument for `func`.
- If the parent has a policy of `launch::deferred` and the continuation does not have a specified launch policy or scheduler, then the parent is filled by immediately calling `wait`, and the policy of the antecedent is `launch::deferred`

Returns: An object of type `future<decltype(func(*this))>` that refers to the shared state created by the continuation.

Postcondition: The `shared_future` passed to the continuation function is a copy of the original `shared_future`

- `valid() == true` on the original `shared_future` object

```
template<typename R2>
future<R2> shared_future<R>::unwrap();
```

Requires: `R` is a `future<R2>` or `shared_future<R2>`

Notes: Removes the outer-most `shared_future` and returns a proxy to the inner future. The proxy is a representation of the inner future and it holds the same value (or exception) as the inner future.

Effects:

- `future<R2> X = shared_future<future<R2>>.unwrap()`, returns a `future<R2>` that becomes ready when the shared state of the inner future is ready. When the inner future is ready, its value (or exception) is moved to the shared state of the returned future.
- `future<R2> Y = shared_future<shared_future<R2>>.unwrap()`, returns a `future<R2>` that becomes ready when the shared state of the inner future is ready. When the inner `shared_future` is ready, its value (or exception) is copied to the shared state of the returned future.
- If the outer future throws an exception, and `get()` is called on the returned future, the returned future throws the same exception as the outer future. This is the case because the inner future didn't exit.

Returns: a future of type `R2`. The result of the inner future is moved out (`shared_future` is copied out) and stored in the shared state of the returned future when it is ready or the result of the inner future throws an exception.

Postcondition: The returned future has `valid() == true`, regardless of the validity of the inner future.

```
bool is_ready() const;
```

Returns: `true` if the shared state is ready, `false` if it isn't.

3.4 30.6.X Function template `when_all` [futures.when-all]

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

template <typename... T>
see below when_all(T&&... futures);
```

Requires: `T` is of type `future<R>` or `shared_future<R>`.

Notes:

- There are two variations of `when_all`. The first version takes a pair of `InputIterators`. The second takes any arbitrary number of `future<R0>` and `shared_future<R1>` objects, where `R0` and `R1` need not be the same type.
- Calling the first signature of `when_all` where `InputIterator` index first equals index last, returns a future with an empty vector that is immediately ready.
- Calling the second signature of `when_any` with no arguments returns a `future<tuple<>>` that is immediately ready.

Effects:

- Each `future` and `shared_future` is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection.
- The future returned by `when_all` will not throw an exception, but the futures held in the output collection may.

Returns:

- `future<tuple<>>` if `when_all` is called with zero arguments.
- `future<vector<future<R>>>` if the input cardinality is unknown at compile and the iterator pair yields `future<R>`. `R` may be `void`. The order of the futures in the output vector will be the same as given by the input iterator.
- `future<vector<shared_future<R>>>` if the input cardinality is unknown at compile time and the iterator pair yields `shared_future<R>`. `R` may be `void`. The order of the futures in the output vector will be the same as given by the input iterator.
- `future<tuple<future<R0>, future<R1>, future<R2>...>>` if inputs are fixed in number. The inputs can be any arbitrary number of `future` and `shared_future` objects. The type of the element at each position of the tuple corresponds to the type of the argument at the same position. Any of `R0`, `R1`, `R2`, etc. may be `void`.

Postconditions:

- All input `future<T>`s `valid() == false`
- All output `shared_future<T>` `valid() == true`

3.5 30.6.X Function template `when_any` [futures.when_any]

```
template <class InputIterator>
see below when_any(InputIterator first, InputIterator last);

template <typename... T>
see below when_any(T&&... futures);
```

Requires: `T` is of type `future<R>` or `shared_future<R>`. All `R` types must be the same.

Notes:

- There are two variations of `when_any`. The first version takes a pair of `InputIterators`. The second takes any arbitrary number of `future<R>` and `shared_future<R>` objects, where `R` need not be the same type.
- Calling the first signature of `when_any` where `InputIterator` index first equals index last, returns a future with an empty vector that is immediately ready.
- Calling the second signature of `when_any` with no arguments returns a `future<tuple<>>` that is immediately ready.

Effects:

- Each future and `shared_future` is waited upon. When at least one is ready, all the futures are copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection.
- The future returned by `when_any` will not throw an exception, but the futures held in the output collection may.

Returns:

- `future<tuple<>>` if `when_any` is called with zero arguments.
- `future<vector<future<R>>>` if the input cardinality is unknown at compile time and the iterator pair yields `future<R>`. `R` may be void. The order of the futures in the output vector will be the same as given by the input iterator.
- `future<vector<shared_future<R>>>` if the input cardinality is unknown at compile time and the iterator pair yields `shared_future<R>`. `R` may be `void`. The order of the futures in the output vector will be the same as given by the input iterator.
- `future<tuple<future<R0>, future<R1>, future<R2>...>>` if inputs are fixed in number. The inputs can be any arbitrary number of `future` and `shared_future` objects. The type of the element at each position of the tuple corresponds to the type of the argument at the same position. Any of `R0`, `R1`, `R2`, etc. maybe `void`.

Postconditions:

- All input `future<T>`s `valid() == false`
- All input `shared_future<T> valid() == true`

3.6 30.6.X Function template `when_any_swapped` [futures.when_any_swapped]

`template <class InputIterator>`
see below `when_any_swapped(InputIterator first, InputIterator last);`

Requires: `InputIterator`'s value type shall be convertible to `future<R>` or `shared_future<R>`. All `R` types must be the same.

Notes:

- The function `when_any_swapped` takes a pair of `InputIterators`.
- Calling `when_any_swapped` where `InputIterator` index first equals index last, returns a `future` with an empty vector that is immediately ready.

Effects:

- Each future and shared_future is waited upon. When at least one is ready, all the futures are copied into the collection of the output (returned) future.
- After the copy, the future or shared_future that was first detected as being ready swaps its position with that of the last element of the result collection, so that the ready future or shared_future may be identified in constant time. Only one future or shared_future is thus moved.
- The future returned by when_any_swapped will not throw an exception, but the futures held in the output collection may.

Returns:

- future<vector<future<R>>> if the input cardinality is unknown at compile time and the iterator pair yields future<R>. R may be void.
- future<vector<shared_future<R>>> if the input cardinality is unknown at compile time and the iterator pair yields shared_future<R>. R may be void.

Postconditions:

- All input future<T>s valid() == false
- All input shared_future<T> valid() == true

3.7 30.6.X Function template make_ready_future [futures.make__ready__future]

```
template <typename T>
future<typename decay<T>::type> make_ready_future(T&& value);

future<void> make_ready_future();
```

Effects: The value that is passed in to the function is moved to the shared state of the returned function if it is an rvalue. Otherwise the value is copied to the shared state of the returned function.

Returns:

- future<T>, if function is given a value of type T
- future<void>, if the function is not given any inputs.

Postcondition:

- Returned future<T>, valid() == true
- Returned future<T>, is_ready() = true

3.8 30.6.8 Function template async [futures.async]

Change 30.6.8/1 as follows:

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.

```
template <class F, class... Args>
future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>
async(F&& f, Args&&... args);

template <class F, class... Args>
future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>
async(launch policy, F&& f, Args&&... args);

template<class F, class... Args>
future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>
async(executor &ex, F&& f, Args&&... args);
```

Change 30.6.8/3 as follows:

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 and third functions 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 `policy & launch::async` is non-zero - calls `INVOKE (DECAY_COPY (std::forward<F>(f)), DECAY_COPY (std::forward<Args>(args))...)` (20.8.2, 30.3.1.2) 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 `policy & launch::deferred` is non-zero - 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))....` The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (30.6.4) on an asynchronous return object referring to this shared state shall invoke 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*]

The further behavior of the third function is as follows:

The `executor::add()` function is given a `function<void ()>` which calls `INVOKE (DECAY_COPY (std::forward<F>(f)) DECAY_COPY (std::forward<Args>(args))...)`. The implementation of the executor is decided by the programmer.

Change 30.6.8/8 as follows:

Remarks: The first signature shall not participate in overload resolution if `decay<F>::type` is `std:: launch` or `std::executor`.