D.4. <future> header

The <future> header provides facilities for handling asynchronous results from operations that may be performed on another thread.

Header contents

```
namespace std
{
    enum class future_status {
        ready, timeout, deferred };

    enum class future_errc
    {

         broken_promise,
         future_already_retrieved,
        promise_already_satisfied,
        no_state
     };

     class future_error;

     const error_category& future_category();

     error_code make_error_code(future_errc e);
     error_condition make_error_condition(future_errc e);

     template<typename ResultType>
     class future;

     template<typename ResultType>
     class shared_future;

     template<typename ResultType>
     class promise;

     template<typename FunctionSignature>
     class packaged_task; // no definition provided

     template<typename ResultType,typename ... Args>
     class packaged_task<ResultType (Args...)>;

     enum class launch {
         async, deferred
     };
     template<typename FunctionType,typename ... Args>
     future<result_of<FunctionType(Args...)>::type>
     async(FunctionType&& func,Args&& ... args);
```

```
    template<typename FunctionType,typename ... Args>
    future<result_of<FunctionType(Args...)>::type>
    async(std::launch policy,FunctionType&& func,Args&& ... args);

}
```

D.4.1. std::future class template

The `std::future` class template provides a means of waiting for an asynchronous result from another thread, in conjunction with the `std::promise` and `std::packaged_task` class templates and the `std::async` function template, which can be used to provide that asynchronous result. Only one `std::future` instance references any given asynchronous result at any time.

Instances of `std::future` are `MoveConstructible` and `MoveAssignable` but not `CopyConstructible` or `CopyAssignable`.

Class definition

```
template<typename ResultType>
class future
{
public:
    future() noexcept;
    future(future&&) noexcept;
    future& operator=(future&&) noexcept;
    ~future();

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

    shared_future<ResultType> share();

    bool valid() const noexcept;

    see description get();

    void wait();

    template<typename Rep,typename Period>
    future_status wait_for(
        std::chrono::duration<Rep,Period> const& relative_time);

    template<typename Clock,typename Duration>
    future_status wait_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time);
};
```

Std::Future Default Constructor

Constructs a `std::future` object without an associated asynchronous result.

Declaration

```
future() noexcept;
```

Effects

Constructs a new `std::future` instance.

Postconditions

`valid()` returns `false`.

Throws

Nothing.

Std::Future Move Constructor

Constructs one `std::future` object from another, transferring ownership of the asynchronous result associated with the other `std::future` object to the newly constructed instance.

Declaration

```
future(future&& other) noexcept;
```

Effects

Move-constructs a new `std::future` instance from `other`.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the constructor is associated with the newly constructed `std::future` object. `other` has no associated asynchronous result. `this->valid()` returns the same value that `other.valid()` returned before the invocation of this constructor. `other.valid()` returns `false`.

Throws

Nothing.

Std::Future Move Assignment Operator

Transfers ownership of the asynchronous result associated with the one `std::future` object to another.

Declaration

```
future(future&& other) noexcept;
```

Effects

Transfers ownership of an asynchronous state between `std::future` instances.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the constructor is associated with `*this`. `other` has no associated asynchronous result. The ownership of the asynchronous state (if any) associated with `*this` prior to the call is released, and the state

destroyed if this is the last reference. `this->valid()` returns the same value that `other.valid()` returned before the invocation of this constructor. `other.valid()` returns `false`.

Throws

Nothing.

Std::Future Destructor

Destroys a `std::future` object.

Declaration

```
~future();
```

Effects

Destroys `*this`. If this is the last reference to the asynchronous result associated with `*this` (if any), then destroy that asynchronous result.

Throws

Nothing

Std::Future::Share Member Function

Constructs a new `std::shared_future` instance and transfers ownership of the asynchronous result associated with `*this` to this newly constructed `std::shared_future` instance.

Declaration

```
shared_future<ResultType> share();
```

Effects

As-if `shared_future<ResultType>(std::move(*this))`.

Postconditions

The asynchronous result associated with `*this` prior to the invocation of `share()` (if any) is associated with the newly constructed `std::shared_future` instance. `this->valid()` returns `false`.

Throws

Nothing.

Std::Future::Valid Member Function

Checks if a `std::future` instance is associated with an asynchronous result.

Declaration

```
bool valid() const noexcept;
```

Returns

`true` if the `*this` has an associated asynchronous result, `false` otherwise.

Throws

Nothing.

Std::Future::Wait Member Function

If the state associated with `*this` contains a deferred function, invokes the deferred function. Otherwise, waits until the asynchronous result associated with an instance of `std::future` is ready.

Declaration

```
void wait();
```

Preconditions

`this->valid()` would return `true`.

Effects

If the associated state contains a deferred function, invokes the deferred function and stores the returned value or thrown exception as the asynchronous result. Otherwise, blocks until the asynchronous result associated with `*this` is *ready*.

Throws

Nothing.

Std::Future::Wait_For Member Function

Waits until the asynchronous result associated with an instance of `std::future` is ready or until a specified time period has elapsed.

Declaration

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time);
```

Preconditions

`this->valid()` would return `true`.

Effects

If the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that hasn't yet started execution, returns immediately without blocking. Otherwise blocks until the asynchronous result associated with `*this` is *ready* or the time period specified by `relative_time` has elapsed.

Returns

`std::future_status::deferred` if the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that hasn't yet started execution, `std::future_status::ready` if the asynchronous result associated with `*this` is *ready*, `std::future_status::timeout` if the time period specified by `relative_time` has elapsed.

Note

The thread may be blocked for longer than the specified duration. Where possible, the elapsed

time is determined by a steady clock.

Throws

Nothing.

Std::Future::Wait_Until Member Function

Waits until the asynchronous result associated with an instance of `std::future` is ready or until a specified time period has elapsed.

Declaration

```
template<typename Clock,typename Duration>
future_status wait_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Preconditions

`this->valid()` would return `true`.

Effects

If the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that hasn't yet started execution, returns immediately without blocking. Otherwise blocks until the asynchronous result associated with `*this` is *ready* or `Clock::now()` returns a time equal to or later than `absolute_time`.

Returns

`std::future_status::deferred` if the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that hasn't yet started execution, `std::future_status::ready` if the asynchronous result associated with `*this` is *ready*, `std::future_status::timeout` if `Clock::now()` returns a time equal to or later than `absolute_time`.

Note

There's no guarantee as to how long the calling thread will be blocked, only that if the function returns `std::future_status::timeout`, then `Clock::now()` returns a time equal to or later than `absolute_time` at the point at which the thread became unblocked.

Throws

Nothing.

Std::Future::Get Member Function

If the associated state contains a deferred function from a call to `std::async`, invokes that function and returns the result; otherwise, waits until the asynchronous result associated with an instance of `std::future` is ready, and then returns the stored value or throw the stored exception.

Declaration

```
void future<void>::get();
```

```
R& future<R&>::get();
R future<R>::get();
```

Preconditions

`this->valid()` would return `true`.

Effects

If the state associated with `*this` contains a deferred function, invokes the deferred function and returns the result or propagates any thrown exception.

Otherwise, blocks until the asynchronous result associated with `*this` is *ready*. If the result is a stored exception, throws that exception. Otherwise, returns the stored value.

Returns

If the associated state contains a deferred function, the result of the function invocation is returned. Otherwise, if `ResultType` is `void`, the call returns normally. If `ResultType` is `R&` for some type `R`, the stored reference is returned. Otherwise, the stored value is returned.

Throws

The exception thrown by the deferred exception or stored in the asynchronous result, if any.

Postcondition

```
this->valid()==false
```

D.4.2. std::shared_future class template

The `std::shared_future` class template provides a means of waiting for an asynchronous result from another thread, in conjunction with the `std::promise` and `std::packaged_task` class templates and `std::async` function template, which can be used to provide that asynchronous result. Multiple `std::shared_future` instances can reference the same asynchronous result.

Instances of `std::shared_future` are `CopyConstructible` and `CopyAssignable`. You can also move-construct a `std::shared_future` from a `std::future` with the same `ResultType`.

Accesses to a given instance of `std::shared_future` aren't synchronized. It's therefore *not safe* for multiple threads to access the same `std::shared_future` instance without external synchronization. But accesses to the associated state are synchronized, so it *is* safe for multiple threads to each access separate instances of `std::shared_future` that share the same associated state without external synchronization.

Class definition

```
template<typename ResultType>
class shared_future
{
public:
    shared_future() noexcept;
    shared_future(future<ResultType>&&) noexcept;

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

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

    bool valid() const noexcept;

    see description  get() const;

    void wait() const;

    template<typename Rep,typename Period>
    future_status wait_for(
        std::chrono::duration<Rep,Period> const& relative_time) const;

    template<typename Clock,typename Duration>
    future_status wait_until(
        std::chrono::time_point<Clock,Duration> const& absolute_time)
    const;
};
```

Std::Shared_Future Default Constructor

Constructs a `std::shared_future` object without an associated asynchronous result.

Declaration

```
shared_future() noexcept;
```

Effects

Constructs a new `std::shared_future` instance.

Postconditions

`valid()` returns `false` for the newly constructed instance.

Throws

Nothing.

Std::Shared_Future Move Constructor

Constructs one `std::shared_future` object from another, transferring ownership of the asynchronous result associated with the other `std::shared_future` object to the newly constructed instance.

Declaration

```
shared_future(shared_future&& other) noexcept;
```

Effects

Constructs a new `std::shared_future` instance.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the constructor is associated with the newly constructed `std::shared_future` object. `other` has no associated asynchronous result.

Throws

Nothing.

Std::Shared_Future Move-from-Std::Future Constructor

Constructs a `std::shared_future` object from a `std::future`, transferring ownership of the asynchronous result associated with the `std::future` object to the newly constructed `std::shared_future`.

Declaration

```
shared_future(std::future<ResultType>&& other) noexcept;
```

Effects

Constructs a new `std::shared_future` instance.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the constructor is associated with the newly constructed `std::shared_future` object. `other` has no associated asynchronous result.

Throws

Nothing.

Std::Shared_Future Copy Constructor

Constructs one `std::shared_future` object from another, so that both the source and the copy refer to the asynchronous result associated with the source `std::shared_future` object, if any.

Declaration

```
shared_future(shared_future const& other);
```

Effects

Constructs a new `std::shared_future` instance.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the constructor is associated with the newly constructed `std::shared_future` object *and* `other`.

Throws

Nothing.

Std::Shared_Future Destructor

Destroys a `std::shared_future` object.

Declaration

```
~shared_future();
```

Effects

Destroys `*this`. If there's no longer a `std::promise` or `std::packaged_task` instance associated with the asynchronous result associated with `*this`, and this is the last `std::shared_future` instance associated with that asynchronous result, destroys that asynchronous result.

Throws

Nothing.

Std::Shared_Future::Valid Member Function

Checks if a `std::shared_future` instance is associated with an asynchronous result.

Declaration

```
bool valid() const noexcept;
```

Returns

`true` if the `*this` has an associated asynchronous result, `false` otherwise.

Throws

Nothing.

Std::Shared_Future::Wait Member Function

If the state associated with `*this` contains a deferred function, invokes the deferred function. Otherwise, waits until the asynchronous result associated with an instance of `std::shared_future` is ready.

Declaration

```
void wait() const;
```

Preconditions

`this->valid()` would return `true`.

Effects

Calls to `get()` and `wait()` from multiple threads on `std::shared_future` instances that share the same associated state are serialized. If the associated state contains a deferred function, the first call to `get()` or `wait()` invokes the deferred function and stores the returned value or thrown exception as the asynchronous result.

Blocks until the asynchronous result associated with `*this` is *ready*.

Throws

Nothing.

Std::Shared_Future::Wait_For Member Function

Waits until the asynchronous result associated with an instance of `std::shared_future` is ready or until a specified time period has elapsed.

Declaration

```
template<typename Rep,typename Period>
future_status wait_for(
    std::chrono::duration<Rep,Period> const& relative_time) const;
```

Preconditions

`this->valid()` would return `true`.

Effects

If the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that has not yet started execution, returns immediately without blocking. Otherwise, blocks until the asynchronous result associated with `*this` is *ready* or the time period specified by `relative_time` has elapsed.

Returns

`std::future_status::deferred` if the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that hasn't yet started execution,
`std::future_status::ready` if the asynchronous result associated with `*this` is *ready*,
`std::future_status::timeout` if the time period specified by `relative_time` has elapsed.

Note

The thread may be blocked for longer than the specified duration. Where possible, the elapsed time is determined by a steady clock.

Throws

Nothing.

Std::Shared_Future::Wait_Until Member Function

Waits until the asynchronous result associated with an instance of `std::shared_future` is ready or until a specified time period has elapsed.

Declaration

```
template<typename Clock,typename Duration>
bool wait_until(
    std::chrono::time_point<Clock,Duration> const& absolute_time) const;
```

Preconditions

`this->valid()` would return `true`.

Effects

If the asynchronous result associated with `*this` contains a deferred function arising from a call to

`std::async` that hasn't yet started execution, returns immediately without blocking. Otherwise, blocks until the asynchronous result associated with `*this` is *ready* or `Clock::now()` returns a time equal to or later than `absolute_time`.

Returns

`std::future_status::deferred` if the asynchronous result associated with `*this` contains a deferred function arising from a call to `std::async` that hasn't yet started execution, `std::future_status::ready` if the asynchronous result associated with `*this` is *ready*, `std::future_status::timeout` if `Clock::now()` returns a time equal to or later than `absolute_time`.

Note

There's no guarantee as to how long the calling thread will be blocked, only that if the function returns `std::future_status::timeout`, then `Clock::now()` returns a time equal to or later than `absolute_time` at the point at which the thread became unblocked.

Throws

Nothing.

Std::Shared_Future::Get Member Function

If the associated state contains a deferred function from a call to `std::async`, invokes that function and return the result. Otherwise, waits until the asynchronous result associated with an instance of `std::shared_future` is ready, and then returns the stored value or throws the stored exception.

Declaration

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

Preconditions

`this->valid()` would return `true`.

Effects

Calls to `get()` and `wait()` from multiple threads on `std::shared_future` instances that share the same associated state are serialized. If the associated state contains a deferred function, the first call to `get()` or `wait()` invokes the deferred function and stores the returned value or thrown exception as the asynchronous result.

Blocks until the asynchronous result associated with `*this` is *ready*. If the asynchronous result is a stored exception, throws that exception. Otherwise, returns the stored value.

Returns

If `ResultType` is `void`, returns normally. If `ResultType` is `R&` for some type `R`, returns the stored reference. Otherwise, returns a `const` reference to the stored value.

Throws

The stored exception, if any.

D.4.3. std::packaged_task class template

The `std::packaged_task` class template packages a function or other callable object so that when the function is invoked through the `std::packaged_task` instance, the result is stored as an asynchronous result for retrieval through an instance of `std::future`.

Instances of `std::packaged_task` are `MoveConstructible` and `MoveAssignable` but not `CopyConstructible` or `CopyAssignable`.

Class definition

```
template<typename FunctionType>
class packaged_task; // undefined

template<typename ResultType,typename... ArgTypes>
class packaged_task<ResultType(ArgTypes...)>
{
public:

    packaged_task() noexcept;
    packaged_task(packaged_task&&) noexcept;
    ~packaged_task();

    packaged_task& operator=(packaged_task&&) noexcept;

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

    void swap(packaged_task&) noexcept;

    template<typename Callable>
    explicit packaged_task(Callable&& func);

    template<typename Callable,typename Allocator>
    packaged_task(std::allocator_arg_t, const Allocator&,Callable&&);

    bool valid() const noexcept;
    std::future<ResultType> get_future();
    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(ArgTypes...);
    void reset();
};
```

Std::Packaged_Task Default Constructor

Constructs a `std::packaged_task` object.

Declaration

```
packaged_task() noexcept;
```

Effects

Constructs a `std::packaged_task` instance with no associated task or asynchronous result.

Throws

Nothing.

Std::Packaged_Task Construction from a Callable Object

Constructsa `std::packaged_task` object with an associated task and asynchronous result.

Declaration

```
template<typename Callable>
packaged_task(Callable&& func);
```

Preconditions

The expression `func(args...)` shall be valid, where each element `args-`*i* in `args...` shall be a value of the corresponding type `ArgTypes-`*i* in `ArgTypes...`. The return value shall be convertible to `ResultType`.

Effects

Constructs a `std::packaged_task` instance with an associated asynchronous result of type `ResultType` that isn't *ready* and an associated task of type `Callable` that's a copy of `func`.

Throws

An exception of type `std::bad_alloc` if the constructor is unable to allocate memory for the asynchronous result. Any exception thrown by the copy or move constructor of `Callable`.

Std::Packaged_Task Construction From A Callable Object With An Allocator

Constructs a `std::packaged_task` object with an associated task and asynchronous result, using the supplied allocator to allocate memory for the associated asynchronous result and task.

Declaration

```
template<typename Allocator,typename Callable>
packaged_task(
    std::allocator_arg_t, Allocator const& alloc,Callable&& func);
```

Preconditions

The expression `func(args...)` shall be valid, where each element `args-`*i* in `args...` shall be a value of the corresponding type `ArgTypes-`*i* in `ArgTypes...`. The return value shall be convertible to `ResultType`.

Effects

Constructs a `std::packaged_task` instance with an associated asynchronous result of type `ResultType` that isn't *ready* and an associated task of type `Callable` that's a copy of `func`. The memory for the asynchronous result and task is allocated through the allocator `alloc` or a copy thereof.

Throws

Any exception thrown by the allocator when trying to allocate memory for the asynchronous result

or task. Any exception thrown by the copy or move constructor of `Callable`.

Std::Packaged_Task Move Constructor

Constructs one `std::packaged_task` object from another, transferring ownership of the asynchronous result and task associated with the other `std::packaged_task` object to the newly constructed instance.

Declaration

```
packaged_task(packaged_task&& other) noexcept;
```

Effects

Constructs a new `std::packaged_task` instance.

Postconditions

The asynchronous result and task associated with `other` prior to the invocation of the constructor is associated with the newly constructed `std::packaged_task` object. `other` has no associated asynchronous result.

Throws

Nothing.

Std::Packaged_Task Move-Assignment Operator

Transfers ownership of the asynchronous result associated with one `std::packaged_task` object to another.

Declaration

```
packaged_task& operator=(packaged_task&& other) noexcept;
```

Effects

Transfers ownership of the asynchronous result and task associated with `other` to `*this`, and discards any prior asynchronous result, as-if `std::packaged_task(other).swap(*this)`.

Postconditions

The asynchronous result and task associated with `other` prior to the invocation of the move-assignment operator is associated with the `*this`. `other` has no associated asynchronous result.

Returns

```
*this
```

Throws

Nothing.

Std::Packaged_Task::Swap Member Function

Exchanges ownership of the asynchronous results associated with two `std::packaged_task` objects.

Declaration

```
void swap(packaged_task& other) noexcept;
```

Effects

Exchanges ownership of the asynchronous results and tasks associated with `other` and `*this`.

Postconditions

The asynchronous result and task associated with `other` prior to the invocation of `swap` (if any) is associated with `*this`. The asynchronous result and task associated with `*this` prior to the invocation of `swap` (if any) is associated with `other`.

Throws

Nothing.

Std::Packaged_Task Destructor

Destroys a `std::packaged_task` object.

Declaration

```
~packaged_task();
```

Effects

Destroys `*this`. If `*this` has an associated asynchronous result, and that result doesn't have a stored task or exception, then that result becomes *ready* with a `std::future_error` exception with an error code of `std::future_errc::broken_promise`.

Throws

Nothing.

Std::Packaged_Task::Get_Future Member Function

Retrieves a `std::future` instance for the asynchronous result associated with `*this`.

Declaration

```
std::future<ResultType> get_future();
```

Preconditions

`*this` has an associated asynchronous result.

Returns

A `std::future` instance for the asynchronous result associated with `*this`.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::future_already_retrieved` if a `std::future` has already been obtained for this asynchronous result through a prior call to `get_future()`.

Std::Packaged_Task::Reset Member Function

Associates a `std::packaged_task` instance with a new asynchronous result for the same task.

Declaration

```
void reset();
```

Preconditions

`*this` has an associated asynchronous task.

Effects

As-if `*this=packaged_task(std::move(f))`, where `f` is the stored task associated with `*this`.

Throws

An exception of type `std::bad_alloc` if memory couldn't be allocated for the new asynchronous result.

Std::Packaged_Task::Valid Member Function

Checks whether `*this` has an associated task and asynchronous result.

Declaration

```
bool valid() const noexcept;
```

Returns

`true` if `*this` has an associated task and asynchronous result, `false` otherwise.

Throws

Nothing.

Std::Packaged_Task::Operator() Function Call Operator

Invokes the task associated with a `std::packaged_task` instance, and stores the return value or exception in the associated asynchronous result.

Declaration

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

Preconditions

`*this` has an associated task.

Effects

Invokes the associated task `func` as-if `INVOKE(func,args...)`. If the invocation returns normally, stores the return value in the asynchronous result associated with `*this`. If the invocation returns with an exception, stores the exception in the asynchronous result associated with `*this`.

Postconditions

The asynchronous result associated with `*this` is *ready* with a stored value or exception. Any threads blocked waiting for the asynchronous result are unblocked.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::promise_already_satisfied` if the asynchronous result already has a stored value or exception.

Synchronization

A successful call to the function call operator synchronizes-with a call to `std::future<ResultType>::get()` or `std::shared_future<ResultType>::get()`, which retrieves the value or exception stored.

Std::Packaged_Task::Make_Ready_At_Thread_Exit Member Function

Invokes the task associated with a `std::packaged_task` instance, and stores the return value or exception in the associated asynchronous result without making the associated asynchronous result *ready* until thread exit.

Declaration

```
void make_ready_at_thread_exit(ArgTypes... args);
```

Preconditions

`*this` has an associated task.

Effects

Invokes the associated task `func` as-if `INVOKE(func,args...)`. If the invocation returns normally, stores the return value in the asynchronous result associated with `*this`. If the invocation returns with an exception, stores the exception in the asynchronous result associated with `*this`. Schedules the associated asynchronous state to be made *ready* when the current thread exits.

Postconditions

The asynchronous result associated with `*this` has a stored value or exception but isn't *ready* until the current thread exits. Threads blocked waiting for the asynchronous result will be unblocked when the current thread exits.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::promise_already_satisfied` if the asynchronous result already has a stored value or exception. An exception of type `std::future_error` with an error code of `std::future_errc::no_state` if `*this` has no associated asynchronous state.

Synchronization

The completion of the thread that made a successful call to `make_ready_at_thread_exit()` synchronizes-with a call to `std::future<ResultType>::get()` or `std::shared_future<ResultType>::get()`, which retrieves the value or exception stored.

D.4.4. std::promise class template

The `std::promise` class template provides a means of setting an asynchronous result, which may be retrieved from another thread through an instance of `std::future`.

The `ResultType` template parameter is the type of the value that can be stored in the asynchronous result.

A `std::future` associated with the asynchronous result of a particular `std::promise` instance can be obtained by calling the `get_future()` member function. The asynchronous result is set either to a value of type `ResultType` with the `set_value()` member function or to an exception with the `set_exception()` member function.

Instances of `std::promise` are `MoveConstructible` and `MoveAssignable` but not `CopyConstructible` or `CopyAssignable`.

Class definition

```
template<typename ResultType>
class promise
{
public:
    promise();
    promise(promise&&) noexcept;
    ~promise();
    promise& operator=(promise&&) noexcept;

    template<typename Allocator>
    promise(std::allocator_arg_t, Allocator const&);

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

    void swap(promise& ) noexcept;

    std::future<ResultType> get_future();

    void set_value(see description);
    void set_exception(std::exception_ptr p);
};
```

Std::Promise Default Constructor

Constructs a `std::promise` object.

Declaration

```
promise();
```

Effects

Constructs a `std::promise` instance with an associated asynchronous result of type `ResultType` that's not *ready*.

Throws

An exception of type `std::bad_alloc` if the constructor is unable to allocate memory for the

asynchronous result.

Std::Promise Allocator Constructor

Constructs a `std::promise` object, using the supplied allocator to allocate memory for the associated asynchronous result.

Declaration

```
template<typename Allocator>
promise(std::allocator_arg_t, Allocator const& alloc);
```

Effects

Constructs a `std::promise` instance with an associated asynchronous result of type `ResultType` that isn't *ready*. The memory for the asynchronous result is allocated through the allocator `alloc`.

Throws

Any exception thrown by the allocator when attempting to allocate memory for the asynchronous result.

Std::Promise Move Constructor

Constructs one `std::promise` object from another, transferring ownership of the asynchronous result associated with the other `std::promise` object to the newly constructed instance.

Declaration

```
promise(promise&& other) noexcept;
```

Effects

Constructs a new `std::promise` instance.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the constructor is associated with the newly constructed `std::promise` object. `other` has no associated asynchronous result.

Throws

Nothing.

Std::Promise Move-Assignment Operator

Transfers ownership of the asynchronous result associated with one `std::promise` object to another.

Declaration

```
promise& operator=(promise&& other) noexcept;
```

Effects

Transfers ownership of the asynchronous result associated with `other` to `*this`. If `*this` already had

an associated asynchronous result, that asynchronous result is made *ready* with an exception of type `std::future_error` and an error code of `std::future_errc::broken_promise`.

Postconditions

The asynchronous result associated with `other` prior to the invocation of the move-assignment operator is associated with the `*this`. `other` has no associated asynchronous result.

Returns

```
*this
```

Throws

Nothing.

Std::Promise::Swap Member Function

Exchanges ownership of the asynchronous results associated with two `std::promise` objects.

Declaration

```
void swap(promise& other);
```

Effects

Exchanges ownership of the asynchronous results associated with `other` and `*this`.

Postconditions

The asynchronous result associated with `other` prior to the invocation of `swap` (if any) is associated with `*this`. The asynchronous result associated with `*this` prior to the invocation of `swap` (if any) is associated with `other`.

Throws

Nothing.

Std::Promise Destructor

Destroys a `std::promise` object.

Declaration

```
~promise();
```

Effects

Destroys `*this`. If `*this` has an associated asynchronous result, and that result doesn't have a stored value or exception, that result becomes *ready* with a `std::future_error` exception with an error code of `std::future_errc::broken_promise`.

Throws

Nothing.

Std::Promise::Get_Future Member Function

Retrieves a `std::future` instance for the asynchronous result associated with `*this`.

Declaration

```
std::future<ResultType> get_future();
```

Preconditions

`*this` has an associated asynchronous result.

Returns

A `std::future` instance for the asynchronous result associated with `*this`.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::future_already_retrieved` if a `std::future` has already been obtained for this asynchronous result through a prior call to `get_future()`.

Std::Promise::Set_Value Member Function

Stores a value in the asynchronous result associated with `*this`.

Declaration

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

Preconditions

`*this` has an associated asynchronous result.

Effects

Stores `r` in the asynchronous result associated with `*this` if `ResultType` isn't `void`.

Postconditions

The asynchronous result associated with `*this` is *ready* with a stored value. Any threads blocked waiting for the asynchronous result are unblocked.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::promise_already_satisfied` if the asynchronous result already has a stored value or exception. Any exceptions thrown by the copy-constructor or move-constructor of `r`.

Synchronization

Multiple concurrent calls to `set_value()`, `set_value_at_thread_exit()`, `set_exception()`, and `set_exception_at_thread_exit()` are serialized. A successful call to `set_value()` happens-before a call to `std::future<ResultType>::get()` or `std::shared_future<ResultType>::get()`, which retrieves the value stored.

Std::Promise::Set_Value_At_Thread_Exit Member Function

Stores a value in the asynchronous result associated with `*this` without making that result *ready* until the current thread exits.

Declaration

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

Preconditions

`*this` has an associated asynchronous result.

Effects

Stores `r` in the asynchronous result associated with `*this` if `ResultType` isn't `void`. Marks the asynchronous result as having a stored value. Schedules the associated asynchronous result to be made *ready* when the current thread exits.

Postconditions

The asynchronous result associated with `*this` has a stored value but isn't *ready* until the current thread exits. Threads blocked waiting for the asynchronous result will be unblocked when the current thread exits.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::promise_already_satisfied` if the asynchronous result already has a stored value or exception. Any exceptions thrown by the copy-constructor or move-constructor of `r`.

Synchronization

Multiple concurrent calls to `set_value()`, `set_value_at_thread_exit()`, `set_exception()`, and `set_exception_at_thread_exit()` are serialized. The completion of the thread that made a successful call to `set_value_at_thread_exit()` happens-before a call to `std::future<ResultType>::get()` or `std::shared_future<ResultType>::get()`, which retrieves the exception stored.

Std::Promise::Set_Exception Member Function

Stores an exception in the asynchronous result associated with `*this`.

Declaration

```
void set_exception(std::exception_ptr e);
```

Preconditions

`*this` has an associated asynchronous result. `(bool)e` is `true`.

Effects

Stores `e` in the asynchronous result associated with `*this`.

Postconditions

The asynchronous result associated with `*this` is *ready* with a stored exception. Any threads blocked waiting for the asynchronous result are unblocked.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::promise_already_satisfied` if the asynchronous result already has a stored value or exception.

Synchronization

Multiple concurrent calls to `set_value()` and `set_exception()` are serialized. A successful call to `set_exception()` happens-before a call to `std::future<Result-Type>::get()` or `std::shared_future<ResultType>::get()`, which retrieves the exception stored.

Std::Promise::Set_Exception_At_Thread_Exit Member Function

Stores an exception in the asynchronous result associated with `*this` without making that result *ready* until the current thread exits.

Declaration

```
void set_exception_at_thread_exit(std::exception_ptr e);
```

Preconditions

`*this` has an associated asynchronous result. `(bool)e` is `true`.

Effects

Stores `e` in the asynchronous result associated with `*this`. Schedules the associated asynchronous result to be made *ready* when the current thread exits.

Postconditions

The asynchronous result associated with `*this` has a stored exception but isn't *ready* until the current thread exits. Threads blocked waiting for the asynchronous result will be unblocked when the current thread exits.

Throws

An exception of type `std::future_error` with an error code of `std::future_errc::promise_already_satisfied` if the asynchronous result already has a stored value or exception.

Synchronization

Multiple concurrent calls to `set_value()`, `set_value_at_thread_exit()`, `set_exception()`, and `set_exception_at_thread_exit()` are serialized. The completion of the thread that made a successful call to `set_exception_at_thread_exit()` happens-before a call to `std::future<ResultType>::get()` or `std::shared_future<ResultType>::get()`, which retrieves the exception stored.

D.4.5. std::async function template

`std::async` is a simple way of running self-contained asynchronous tasks to make use of the available hardware concurrency. A call to `std::async` returns a `std::future` that will contain the result of the task. Depending on the launch policy, the task is either run asynchronously on its own thread or synchronously on whichever thread calls the `wait()` or `get()` member functions on that future.

Declaration

```
enum class launch
{
    async,deferred
};

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(Callable&& func,Args&& ... args);

template<typename Callable,typename ... Args>
future<result_of<Callable(Args...)>::type>
async(launch policy,Callable&& func,Args&& ... args);
```

Preconditions

The expression `INVOKE(func,args)` is valid for the supplied values of `func` and `args`. `Callable` and every member of `Args` are `MoveConstructible`.

Effects

Constructs copies of `func` and `args...` in internal storage (denoted by `fff` and `xyz...` respectively).

If `policy` is `std::launch::async`, runs `INVOKE(fff,xyz...)` on its own thread. The returned `std::future` will become *ready* when this thread is complete and will hold either the return value or exception thrown by the function invocation. The destructor of the last future object associated with the asynchronous state of the returned `std::future` blocks until the future is *ready*.

If `policy` is `std::launch::deferred, fff` and `xyz...` are stored in the returned `std::future` as a deferred function call. The first call to the `wait()` or `get()` member functions on a future that shares the same associated state will execute `INVOKE(fff,xyz...)` synchronously on the thread that called `wait()` or `get()`.

The value returned or exception thrown by the execution of `INVOKE(fff,xyz...)` will be returned from a call to `get()` on that `std::future`.

If `policy` is `std::launch::async | std::launch::deferred` or the `policy` argument is omitted, the behavior is as-if either `std::launch::async` or `std::launch::deferred` had been specified. The implementation will choose the behavior on a call-by-call basis in order to take advantage of the available hardware concurrency without excessive oversubscription.

In all cases, the `std::async` call returns immediately.

Synchronization

The completion of the function invocation happens-before a successful return from a call to `wait(), get(), wait_for()`, or `wait_until()` on any `std::future` or `std::shared_future` instance that references the same associated state as the `std::future` object returned from the `std::async` call. In

the case of a `policy` of `std::launch::async`, the completion of the thread on which the function invocation occurs also happens-before the successful return from these calls.

Throws

`std::bad_alloc` if the required internal storage can't be allocated, otherwise `std::future_error` when the effects can't be achieved, or any exception thrown during the construction of `fff` or `xyz...`.