D.5. <mutex> header

The <mutex> header provides facilities for ensuring mutual exclusion: mutex types, lock types and functions, and a mechanism for ensuring an operation is performed exactly once.

Header contents

```
namespace std
{
    class mutex;
    class recursive_mutex;
    class timed_mutex;
    class recursive_timed_mutex;
    struct adopt_lock_t;
    struct defer_lock_t;
    struct try_to_lock_t;

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

    template<typename LockableType>
    class lock_guard;

    template<typename LockableType>
    class unique_lock;

    template<typename LockableType1,typename... LockableType2>
    void lock(LockableType1& m1,LockableType2& m2...);

    template<typename LockableType1,typename... LockableType2>
    int try_lock(LockableType1& m1,LockableType2& m2...);

    struct once_flag;

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

D.5.1. std::mutex class

The `std::mutex` class provides a basic mutual exclusion and synchronization facility for threads that can be used to protect shared data. Prior to accessing the data protected by the mutex, the mutex must be *locked* by calling `lock()` or `try_lock()`. Only one thread may hold the lock at a time, so if another thread also tries to lock the mutex, it will fail (`try_lock()`) or block (`lock()`) as appropriate. Once a thread is done accessing the shared data, it then must call `unlock()` to release the lock and allow other threads to acquire it.

`std::mutex` meets the `Lockable` requirements.

Class definition

```
class mutex
{
public:
    mutex(mutex const&)=delete;
    mutex& operator=(mutex const&)=delete;

    constexpr mutex() noexcept;
    ~mutex();

    void lock();
    void unlock();
    bool try_lock();
};
```

Std::Mutex Default Constructor

Constructs a `std::mutex` object.

Declaration

```
constexpr mutex() noexcept;
```

Effects

Constructs a `std::mutex` instance.

Postconditions

The newly constructed `std::mutex` object is initially unlocked.

Throws

Nothing.

Std::Mutex Destructor

Destroys a `std::mutex` object.

Declaration

```
~mutex();
```

Preconditions

`*this` must not be locked.

Effects

Destroys `*this`.

Throws

Nothing.

Std::Mutex::Lock Member Function

Acquires a lock on a `std::mutex` object for the current thread.

Declaration

```
void lock();
```

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Blocks the current thread until a lock on `*this` can be obtained.

Postconditions

`*this` is locked by the calling thread.

Throws

An exception of type `std::system_error` if an error occurs.

Std::Mutex::Try_Lock Member Function

Attempts to acquire a lock on a `std::mutex` object for the current thread.

Declaration

```
bool try_lock();
```

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Attempts to acquire a lock on `*this` for the calling thread without blocking.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns `true`.

Throws

Nothing.

Note

The function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`.

Std::Mutex::Unlock Member Function

Releases a lock on a `std::mutex` object held by the current thread.

Declaration

```
void unlock();
```

Preconditions

The calling thread must hold a lock on `*this`.

Effects

Releases the lock on `*this` held by the current thread. If any threads are blocked waiting to acquire a lock on `*this`, unblocks one of them.

Postconditions

`*this` is not locked by the calling thread.

Throws

Nothing.

D.5.2. std::recursive_mutex class

The `std::recursive_mutex` class provides a basic mutual exclusion and synchronization facility for threads that can be used to protect shared data. Prior to accessing the data protected by the mutex, the mutex must be *locked* by calling `lock()` or `try_lock()`. Only one thread may hold the lock at a time, so if another thread also tries to lock the `recursive_mutex`, it will fail (`try_lock`) or block (`lock`) as appropriate. Once a thread is done accessing the shared data, it then must call `unlock()` to release the lock and allow other threads to acquire it.

This mutex is *recursive* so a thread that holds a lock on a particular `std::recursive_mutex` instance may make further calls `lock()` or `try_lock()` to increase the lock count. The mutex can't be locked by another thread until the thread that acquired the locks has called `unlock` once for each successful call to `lock()` or `try_lock()`. `std::recursive_mutex` meets the `Lockable` requirements.

Class definition

```
class recursive_timed_mutex
{
public:
    recursive_timed_mutex(recursive_timed_mutex const&)=delete;
    recursive_timed_mutex& operator=(recursive_timed_mutex const&)=delete;

    recursive_timed_mutex();
     ~recursive_timed_mutex();

    void lock();
    void unlock();
    bool try_lock() noexcept;
};
```

Std::Recursive_Mutex Default Constructor

Constructs a `std::recursive_mutex` object.

Declaration

```
recursive_mutex() noexcept;
```

Effects

Constructs a `std::recursive_mutex` instance.

Postconditions

The newly constructed `std::recursive_mutex` object is initially unlocked.

Throws

An exception of type `std::system_error` if unable to create a new `std::recursive_mutex` instance.

Std::Recursive_Mutex Destructor

Destroys a `std::recursive_mutex` object.

Declaration

```
~recursive_mutex();
```

Preconditions

`*this` must not be locked.

Effects

Destroys `*this`.

Throws

Nothing.

Std::Recursive_Mutex::Lock Member Function

Acquires a lock on a `std::recursive_mutex` object for the current thread.

Declaration

```
void lock();
```

Effects

Blocks the current thread until a lock on `*this` can be obtained.

Postconditions

`*this` is locked by the calling thread. If the calling thread already held a lock on `*this`, the lock count is increased by one.

Throws

An exception of type `std::system_error` if an error occurs.

Std::Recursive_Mutex::Try_Lock Member Function

Attempts to acquire a lock on a `std::recursive_mutex` object for the current thread.

Declaration

```
bool try_lock() noexcept;
```

Effects

Attempts to acquire a lock on `*this` for the calling thread without blocking.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

A new lock on `*this` has been obtained for the calling thread if the function returns `true`.

Throws

Nothing.

Note

If the calling thread already holds the lock on `*this`, the function returns `true` and the count of locks on `*this` held by the calling thread is increased by one. If the current thread doesn't already hold a lock on `*this`, the function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`.

Std::Recursive_Mutex::Unlock Member Function

Releases a lock on a `std::recursive_mutex` object held by the current thread.

Declaration

```
void unlock();
```

Preconditions

The calling thread must hold a lock on `*this`.

Effects

Releases a lock on `*this` held by the current thread. If this is the last lock on `*this` held by the calling thread, any threads are blocked waiting to acquire a lock on `*this`. Unblocks one of them.

Postconditions

The number of locks on `*this` held by the calling thread is reduced by one.

Throws

Nothing.

D.5.3. std::timed_mutex class

The `std::timed_mutex` class provides support for locks with timeouts on top of the basic mutual exclusion and synchronization facility provided by `std::mutex`. Prior to accessing the data protected by the mutex, the mutex must be *locked* by calling `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()`. If a lock is already held by another thread, an attempt to acquire the lock will fail (`try_lock()`), block until the lock can be acquired (`lock()`), or block until the lock can be acquired or the lock attempt times out (`try_lock_for()` or `try_lock_until()`). Once a lock has been acquired (whichever function was used to acquire it), it must be released by calling `unlock()` before another thread can acquire the lock on the mutex.

`std::timed_mutex` meets the `TimedLockable` requirements.

Class definition

```
c"ass time"_mutex
{
pub"ic:
    time"_mutex(time"_mutex const&)="e"ete;
    time"_mutex& operator=(time"_mutex const&)="e"ete;

    time"_mutex();
    ~time"_mutex();

    voi" "ock();
    voi" un"ock();
    boo" try_"ock();

    temp"ate&"t;typename Rep,typename Perio">
    boo" try_"ock_for(
        st"::chrono::"uration&"t;Rep,Perio"> const& re"ative_time);

    temp"ate&"t;typename C"ock,typename "uration>
    boo" try_"ock_unti"(
        st"::chrono::time_point&"t;C"ock,"uration> const& abso"ute_time);
};
```

Std::Timed_Mutex Default Constructor

Constructs a `std::timed_mutex` object.

Declaration

```
timed_mutex();
```

Effects

Constructs a `std::timed_mutex` instance.

Postconditions

The newly constructed `std::timed_mutex` object is initially unlocked.

Throws

An exception of type `std::system_error` if unable to create a new `std::timed_mutex` instance.

Std::Timed_Mutex Destructor

Destroys a `std::timed_mutex` object.

Declaration

```
~timed_mutex();
```

Preconditions

`*this` must not be locked.

Effects

Destroys `*this`.

Throws

Nothing.

Std::Timed_Mutex::Lock Member Function

Acquires a lock on a `std::timed_mutex` object for the current thread.

Declaration

```
void lock();
```

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Blocks the current thread until a lock on `*this` can be obtained.

Postconditions

`*this` is locked by the calling thread.

Throws

An exception of type `std::system_error` if an error occurs.

Std::Timed_Mutex::Try_Lock Member Function

Attempts to acquire a lock on a `std::timed_mutex` object for the current thread.

Declaration

```
bool try_lock();
```

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Attempts to acquire a lock on `*this` for the calling thread without blocking.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns `true`.

Throws Nothing.

Note

The function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`.

Std::Timed_Mutex::Try_Lock_For Member Function

Attempts to acquire a lock on a `std::timed_mutex` object for the current thread.

Declaration

```
temp"ate&"t;typename Rep,typename Perio">
boo" try_"ock_for(
    st"::chrono::"uration&"t;Rep,Perio"> const& re"ative_time);
```

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Attempts to acquire a lock on `*this` for the calling thread within the time specified by `relative_time`. If `relative_time.count()` is zero or negative, the call will return immediately, as if it was a call to `try_lock()`. Otherwise, the call blocks until either the lock has been acquired or the time period specified by `relative_time` has elapsed.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns `true`.

Throws

Nothing.

Note

The function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`. The thread may be blocked for longer than the specified duration. Where possible, the elapsed time is determined by a steady clock.

Std::Timed_Mutex::Try_Lock_Until Member Function

Attempts to acquire a lock on a `std::timed_mutex` object for the current thread.

Declaration

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

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Attempts to acquire a lock on `*this` for the calling thread before the time specified by `absolute_time`. If `absolute_time<=Clock::now()` on entry, the call will return immediately, as if it was a call to `try_lock()`. Otherwise, the call blocks until either the lock has been acquired or `Clock::now()` returns a time equal to or later than `absolute_time`.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns `true`.

Throws

Nothing.

Note

The function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`. There's no guarantee as to how long the calling thread will be blocked, only that if the function returns `false`, then `Clock::now()` returns a time equal to or later than `absolute_time` at the point at which the thread became unblocked.

Std::Timed_Mutex::Unlock Member Function

Releases a lock on a `std::timed_mutex` object held by the current thread.

Declaration

```
void unlock();
```

Preconditions

The calling thread must hold a lock on `*this`.

Effects

Releases the lock on `*this` held by the current thread. If any threads are blocked waiting to acquire a lock on `*this`, unblocks one of them.

Postconditions

`*this` is not locked by the calling thread.

Throws

Nothing.

D.5.4. std::recursive_timed_mutex class

The `std::recursive_timed_mutex` class provides support for locks with timeouts on top of the mutual exclusion and synchronization facility provided by `std::recursive_mutex`. Prior to accessing the data protected by the mutex, the mutex must be *locked* by calling `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()`. If a lock is already held by another thread, an attempt to acquire the lock will fail (`try_lock()`), block until the lock can be acquired (`lock()`), or block until the lock can be acquired or the lock attempt times out (`try_lock_for()` or `try_lock_until()`). Once a lock has been acquired (whichever function was used to acquire it) it must be released by calling `unlock()` before another thread can acquire the lock on the mutex.

This mutex is *recursive*, so a thread that holds a lock on a particular instance of `std::recursive_timed_mutex` may acquire additional locks on that instance through any of the lock functions. All of these locks must be released by a corresponding call to `unlock()` before another thread can acquire a lock on that instance.

`std::recursive_timed_mutex` meets the `TimedLockable` requirements.

Class definition

```
class recursive_timed_mutex
{
public:
    recursive_timed_mutex(recursive_timed_mutex const&)=delete;
    recursive_timed_mutex& operator=(recursive_timed_mutex const&)=delete;

    recursive_timed_mutex();
    ~recursive_timed_mutex();

    void lock();
    void unlock();
    bool try_lock() noexcept;

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

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

Std::Recursive_Timed_Mutex Default Constructor

Constructs a `std::recursive_timed_mutex` object.

Declaration

```
recursive_timed_mutex();
```

Effects

Constructs a `std::recursive_timed_mutex` instance.

Postconditions

The newly constructed `std::recursive_timed_mutex` object is initially unlocked.

Throws

An exception of type `std::system_error` if unable to create a new `std::recursive_timed_mutex` instance.

Std::Recursive_Timed_Mutex Destructor

Destroys a `std::recursive_timed_mutex` object.

Declaration

```
~recursive_timed_mutex();
```

Preconditions

`*this` must not be locked.

Effects

Destroys `*this`.

Throws

Nothing.

Std::Recursive_Timed_Mutex::Lock Member Function

Acquires a lock on a `std::recursive_timed_mutex` object for the current thread.

Declaration

```
void lock();
```

Preconditions

The calling thread must not hold a lock on `*this`.

Effects

Blocks the current thread until a lock on `*this` can be obtained.

Postconditions

`*this` is locked by the calling thread. If the calling thread already held a lock on `*this`, the lock count is increased by one.

Throws

An exception of type `std::system_error` if an error occurs.

Std::Recursive_Timed_Mutex::Try_Lock Member Function

Attempts to acquire a lock on a `std::recursive_timed_mutex` object for the current thread.

Declaration

```
bool try_lock() noexcept;
```

Effects

Attempts to acquire a lock on `*this` for the calling thread without blocking.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns `true`.

Throws

Nothing.

Note

If the calling thread already holds the lock on `*this`, the function returns `true` and the count of locks on `*this` held by the calling thread is increased by one. If the current thread doesn't already hold a lock on `*this`, the function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`.

Std::Recursive_Timed_Mutex::Try_Lock_For Member Function

Attempts to acquire a lock on a `std::recursive_timed_mutex` object for the current thread.

Declaration

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

Effects

Attempts to acquire a lock on `*this` for the calling thread within the time specified by `relative_time`. If `relative_time.count()` is zero or negative, the call will return immediately, as if it was a call to `try_lock()`. Otherwise, the call blocks until either the lock has been acquired or the time period specified by `relative_time` has elapsed.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns

`true`.

Throws

Nothing.

Note

If the calling thread already holds the lock on `*this`, the function returns `true` and the count of locks on `*this` held by the calling thread is increased by one. If the current thread doesn't already hold a lock on `*this`, the function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`. The thread may be blocked for longer than the specified duration. Where possible, the elapsed time is determined by a steady clock.

Std::Recursive_Timed_Mutex::Try_Lock_Until Member Function

Attempts to acquire a lock on a `std::recursive_timed_mutex` object for the current thread.

Declaration

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

Effects

Attempts to acquire a lock on `*this` for the calling thread before the time specified by `absolute_time`. If `absolute_time<=Clock::now()` on entry, the call will return immediately, as if it was a call to `try_lock()`. Otherwise, the call blocks until either the lock has been acquired or `Clock::now()` returns a time equal to or later than `absolute_time`.

Returns

`true` if a lock was obtained for the calling thread, `false` otherwise.

Postconditions

`*this` is locked by the calling thread if the function returns `true`.

Throws

Nothing.

Note

If the calling thread already holds the lock on `*this`, the function returns `true` and the count of locks on `*this` held by the calling thread is increased by one. If the current thread doesn't already hold a lock on `*this`, the function may fail to acquire the lock (and return `false`) even if no other thread holds a lock on `*this`. There's no guarantee as to how long the calling thread will be blocked, only that if the function returns `false`, then `Clock::now()` returns a time equal to or later than `absolute_time` at the point at which the thread became unblocked.

Std::Recursive_Timed_Mutex::Unlock Member Function

Releases a lock on a `std::recursive_timed_mutex` object held by the current thread.

Declaration

```
void unlock();
```

Preconditions

The calling thread must hold a lock on `*this`.

Effects

Releases a lock on `*this` held by the current thread. If this is the last lock on `*this` held by the calling thread, any threads are blocked waiting to acquire a lock on `*this`. Unblocks one of them.

Postconditions

The number of locks on `*this` held by the calling thread is reduced by one.

Throws

Nothing.

D.5.5. std::lock_guard class template

The `std::lock_guard` class template provides a basic lock ownership wrapper. The type of mutex being locked is specified by template parameter `Mutex` and must meet the `Lockable` requirements. The specified mutex is locked in the constructor and unlocked in the destructor. This provides a simple means of locking a mutex for a block of code and ensuring that the mutex is unlocked when the block is left, whether that's by running off the end, by the use of a control flow statement such as `break` or `return`, or by throwing an exception.

Instances of `std::lock_guard` are not `MoveConstructible, CopyConstructible`, or `CopyAssignable`.

Class definition

```
template <class Mutex>
class lock_guard
{
public:
    typedef Mutex mutex_type;

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

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

Std::Lock_Guard Locking Constructor

Constructs a `std::lock_guard` instance that locks the supplied mutex.

Declaration

```
explicit lock_guard(mutex_type& m);
```

Effects

Constructs a `std::lock_guard` instance that references the supplied mutex. Calls `m.lock()`.

Throws

Any exceptions thrown by `m.lock()`.

Postconditions

`*this` owns a lock on `m`.

Std::Lock_Guard Lock-Adopting Constructor

Constructs a `std::lock_guard` instance that owns the lock on the supplied mutex.

Declaration

```
lock_guard(mutex_type& m,std::adopt_lock_t);
```

Preconditions

The calling thread must own a lock on `m`.

Effects

Constructs a `std::lock_guard` instance that references the supplied mutex and takes ownership of the lock on `m` held by the calling thread.

Throws

Nothing.

Postconditions

`*this` owns the lock on `m` held by the calling thread.

Std::Lock_Guard Destructor

Destroys a `std::lock_guard` instance and unlocks the corresponding mutex.

Declaration

```
~lock_guard();
```

Effects

Calls `m.unlock()` for the mutex instance `m` supplied when `*this` was constructed.

Throws

Nothing.

D.5.6. std::unique_lock class template

The `std::unique_lock` class template provides a more general lock ownership wrapper than `std::lock_guard`. The type of mutex being locked is specified by the template parameter `Mutex`, which must meet the `BasicLockable` requirements. In general, the specified mutex is locked in the constructor and unlocked in the destructor, although additional constructors and member

functions are provided to allow other possibilities. This provides a means of locking a mutex for a block of code and ensuring that the mutex is unlocked when the block is left, whether that's by running off the end, by the use of a control flow statement such as `break` or `return`, or by throwing an exception. The wait functions of `std::condition_variable` require an instance of `std::unique_lock<std::mutex>`, and all instantiations of `std::unique_lock` are suitable for use with the `Lockable` parameter for the `std::condition_variable_any` wait functions.

If the supplied `Mutex` type meets the `Lockable` requirements, then `std::unique_lock<Mutex>` also meets the `Lockable` requirements. If, in addition, the supplied `Mutex` type meets the `TimedLockable` requirements, then `std::unique_lock<Mutex>` also meets the `TimedLockable` requirements.

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

Class definition

```
template <class Mutex>
class unique_lock
{
public:
    typedef Mutex mutex_type;

    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, adopt_lock_t);
    unique_lock(mutex_type& m, defer_lock_t) noexcept;
    unique_lock(mutex_type& m, try_to_lock_t);
    template<typename Clock,typename Duration>
    unique_lock(
        mutex_type& m,
        std::chrono::time_point<Clock,Duration> const& absolute_time);

    template<typename Rep,typename Period>
    unique_lock(
        mutex_type& m,
        std::chrono::duration<Rep,Period> const& relative_time);

    ~unique_lock();

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

    unique_lock(unique_lock&& );
    unique_lock& operator=(unique_lock&& );

    void swap(unique_lock& other) noexcept;

    void lock();
    bool try_lock();
    template<typename Rep, typename Period>
    bool try_lock_for(
        std::chrono::duration<Rep,Period> const& relative_time);
    template<typename Clock, typename Duration>
    bool try_lock_until(
```

```
            std::chrono::time_point<Clock,Duration> const& absolute_time);
        void unlock();

        explicit operator bool() const noexcept;
        bool owns_lock() const noexcept;
        Mutex* mutex() const noexcept;
        Mutex* release() noexcept;
    };
```

Std::Unique_Lock Default Constructor

Constructs a `std::unique_lock` instance with no associated mutex.

Declaration

```
unique_lock() noexcept;
```

Effects

Constructs a `std::unique_lock` instance that has no associated mutex.

Postconditions

```
this->mutex()==NULL, this->owns_lock()==false.
```

Std::Unique_Lock Locking Constructor

Constructs a `std::unique_lock` instance that locks the supplied mutex.

Declaration

```
explicit unique_lock(mutex_type& m);
```

Effects

Constructs a `std::unique_lock` instance that references the supplied mutex. Calls `m.lock()`.

Throws

Any exceptions thrown by `m.lock()`.

Postconditions

```
this->owns_lock()==true, this->mutex()==&m.
```

Std::Unique_Lock Lock-Adopting Constructor

Constructs a `std::unique_lock` instance that owns the lock on the supplied mutex.

Declaration

```
unique_lock(mutex_type& m,std::adopt_lock_t);
```

Preconditions

The calling thread must own a lock on `m`.

Effects

Constructs a `std::unique_lock` instance that references the supplied mutex and takes ownership of the lock on `m` held by the calling thread.

Throws

Nothing.

Postconditions

```
this->owns_lock()==true, this->mutex()==&m.
```

Std::Unique_Lock Deferred-Lock Constructor

Constructs a `std::unique_lock` instance that doesn't own the lock on the supplied mutex.

Declaration

```
unique_lock(mutex_type& m,std::defer_lock_t) noexcept;
```

Effects

Constructs a `std::unique_lock` instance that references the supplied mutex.

Throws

Nothing.

Postconditions

```
this->owns_lock()==false, this->mutex()==&m.
```

Std::Unique_Lock Try-to-Lock Constructor

Constructs a `std::unique_lock` instance associated with the supplied mutex and tries to acquire a lock on that mutex.

Declaration

```
unique_lock(mutex_type& m,std::try_to_lock_t);
```

Preconditions

The `Mutex` type used to instantiate `std::unique_lock` must meet the `Lockable` requirements.

Effects

Constructs a `std::unique_lock` instance that references the supplied mutex. Calls `m.try_lock()`.

Throws

Nothing.

Postconditions

`this->owns_lock()` returns the result of the `m.try_lock()` call, `this->mutex()==&m`.

Std::Unique_Lock Try-to-Lock Constructor With a Duration Timeout

Constructs a `std::unique_lock` instance associated with the supplied mutex and tries to acquire a lock on that mutex.

Declaration

```
template<typename Rep,typename Period>
unique_lock(
    mutex_type& m,
    std::chrono::duration<Rep,Period> const& relative_time);
```

Preconditions

The `Mutex` type used to instantiate `std::unique_lock` must meet the `Timed-Lockable` requirements.

Effects

Constructs a `std::unique_lock` instance that references the supplied mutex. Calls `m.try_lock_for(relative_time)`.

Throws

Nothing.

Postconditions

`this->owns_lock()` returns the result of the `m.try_lock_for()` call, `this->mutex()==&m`.

Std::Unique_Lock Try-to-Lock Constructor With a Time_Point Timeout

Constructs a `std::unique_lock` instance associated with the supplied mutex and tries to acquire a lock on that mutex.

Declaration

```
template<typename Clock,typename Duration>
unique_lock(
    mutex_type& m,
    std::chrono::time_point<Clock,Duration> const& absolute_time);
```

Preconditions

The `Mutex` type used to instantiate `std::unique_lock` must meet the `Timed-Lockable` requirements.

Effects

Constructs a `std::unique_lock` instance that references the supplied mutex. Calls `m.try_lock_until(absolute_time)`.

Throws

Nothing.

Postconditions

`this->owns_lock()` returns the result of the `m.try_lock_until()` call, `this->mutex()==&m`.

Std::Unique_Lock Move-Constructor

Transfers ownership of a lock from one `std::unique_lock` object to a newly created `std::unique_lock` object.

Declaration

```
unique_lock(unique_lock&& other) noexcept;
```

Effects

Constructs a `std::unique_lock` instance. If `other` owned a lock on a mutex prior to the constructor invocation, that lock is now owned by the newly created `std::unique_lock` object.

Postconditions

For a newly constructed `std::unique_lock` object `x, x.mutex()` is equal to the value of `other.mutex()` prior to the constructor invocation, and `x.owns_lock()` is equal to the value of `other.owns_lock()` prior to the constructor invocation. `other.mutex()==NULL, other.owns_lock()==false`.

Throws

Nothing.

Note

`std::unique_lock` objects are *not* `CopyConstructible`, so there's no copy constructor, only this move constructor.

Std::Unique_Lock Move-Assignment Operator

Transfers ownership of a lock from one `std::unique_lock` object to another `std::unique_lock` object.

Declaration

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

Effects

If `this->owns_lock()` returns `true` prior to the call, calls `this->unlock()`. If `other` owned a lock on a mutex prior to the assignment, that lock is now owned by `*this`.

Postconditions

`this->mutex()` is equal to the value of `other.mutex()` prior to the assignment, and `this->owns_lock()` is equal to the value of `other.owns_lock()` prior to the assignment. `other.mutex()==NULL, other.owns_lock()==false`.

Throws

Nothing.

Note

`std::unique_lock` objects are *not* `CopyAssignable`, so there's no copy-assignment operator, only this move-assignment operator.

Std::Unique_Lock Destructor

Destroys a `std::unique_lock` instance and unlocks the corresponding mutex if it's owned by the destroyed instance.

Declaration

```
~unique_lock();
```

Effects

If `this->owns_lock()` returns `true`, calls `this->mutex()->unlock()`.

Throws

Nothing.

Std::Unique_Lock::Swap Member Function

Exchanges ownership of their associated `unique_locks` of execution between two `std::unique_lock` objects.

Declaration

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

Effects

If `other` owns a lock on a mutex prior to the call, that lock is now owned by `*this`. If `*this` owns a lock on a mutex prior to the call, that lock is now owned by `other`.

Postconditions

`this->mutex()` is equal to the value of `other.mutex()` prior to the call. `other.mutex()` is equal to the value of `this->mutex()` prior to the call. `this->owns_lock()` is equal to the value of `other.owns_lock()` prior to the call. `other.owns_lock()` is equal to the value of `this->owns_lock()` prior to the call.

Throws

Nothing.

Swap Nonmember Function for Std::Unique_Lock

Exchanges ownership of their associated mutex locks between two `std::unique_lock` objects.

Declaration

```
void swap(unique_lock& lhs,unique_lock& rhs) noexcept;
```

Effects

```
lhs.swap(rhs)
```

Throws

Nothing.

Std::Unique_Lock::Lock Member Function

Acquires a lock on the mutex associated with `*this`.

Declaration

```
void lock();
```

Preconditions

```
this->mutex()!=NULL, this->owns_lock()==false.
```

Effects

Calls `this->mutex()->lock()`.

Throws

Any exceptions thrown by `this->mutex()->lock()`. `std::system_error` with an error code of `std::errc::operation_not_permitted` if `this->mutex()==NULL`. `std::system_error` with an error code of `std::errc::resource_deadlock_would_occur` if `this->owns_lock()==true` on entry.

Postconditions

```
this->owns_lock()==true.
```

Std::Unique_Lock::Try_Lock Member Function

Attempts to acquire a lock on the mutex associated with `*this`.

Declaration

```
bool try_lock();
```

Preconditions

The `Mutex` type used to instantiate `std::unique_lock` must meet the `Lockable` requirements. `this->mutex()!=NULL, this->owns_lock()==false`.

Effects

Calls `this->mutex()->try_lock()`.

Returns

`true` if the call to `this->mutex()->try_lock()` returned `true, false` otherwise.

Throws

Any exceptions thrown by `this->mutex()->try_lock()`. `std::system_error` with an error code of `std::errc::operation_not_permitted` if `this->mutex()==NULL`. `std::system_error` with an error code of `std::errc::resource_deadlock_would_occur` if `this->owns_lock()==true` on entry.

Postconditions

If the function returns `true`, `this->owns_lock()==true`, otherwise `this->owns_lock()==false`.

Std::Unique_Lock::Unlock Member Function

Releases a lock on the mutex associated with `*this`.

Declaration

```
void unlock();
```

Preconditions

`this->mutex()!=NULL, this->owns_lock()==true.`

Effects

Calls `this->mutex()->unlock()`.

Throws

Any exceptions thrown by `this->mutex()->unlock()`. `std::system_error` with an error code of `std::errc::operation_not_permitted` if `this->owns_lock()==false` on entry.

Postconditions

`this->owns_lock()==false.`

Std::Unique_Lock::Try_Lock_for Member Function

Attempts to acquire a lock on the mutex associated with `*this` within the time specified.

Declaration

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

Preconditions

The `Mutex` type used to instantiate `std::unique_lock` must meet the `TimedLockable` requirements. `this->mutex()!=NULL, this->owns_lock()==false`.

Effects

Calls `this->mutex()->try_lock_for(relative_time)`.

Returns

`true` if the call to `this->mutex()->try_lock_for()` returned `true`, `false` otherwise.

Throws

Any exceptions thrown by `this->mutex()->try_lock_for()`. `std::system_error` with an error code of `std::errc::operation_not_permitted` if `this->mutex()==NULL`. `std::system_error` with an error code of `std::errc::resource_deadlock_would_occur` if `this->owns_lock()==true` on entry.

Postconditions

If the function returns `true`, `this->owns_lock()==true`, otherwise `this->owns_lock()==false`.

Std::Unique_Lock::Try_Lock_Until Member Function

Attempts to acquire a lock on the mutex associated with `*this` within the time specified.

Declaration

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

Preconditions

The `Mutex` type used to instantiate `std::unique_lock` must meet the `Timed-Lockable` requirements. `this->mutex()!=NULL`, `this->owns_lock()==false`.

Effects

Calls `this->mutex()->try_lock_until(absolute_time)`.

Returns

`true` if the call to `this->mutex()->try_lock_until()` returned `true`, `false` otherwise.

Throws

Any exceptions thrown by `this->mutex()->try_lock_until()`. `std::system_error` with an error code of `std::errc::operation_not_permitted` if `this->mutex()==NULL`. `std::system_error` with an error code of `std::errc::resource_deadlock_would_occur` if `this->owns_lock()==true` on entry.

Postcondition

If the function returns `true`, `this->owns_lock()==true`, otherwise `this->owns_lock()==false`.

Std::Unique_Lock::Operator Bool Member Function

Checks whether or not `*this` owns a lock on a mutex.

Declaration

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

Returns

`this->owns_lock()`.

Throws

Nothing.

Note

This is an `explicit` conversion operator, so it's only implicitly called in contexts where the result is used as a Boolean and not where the result would be treated as an integer value 0 or 1.

Std::Unique_Lock::Owns_Lock Member Function

Checks whether or not `*this` owns a lock on a mutex.

Declaration

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

Returns

`true` if `*this` owns a lock on a mutex, `false` otherwise.

Throws

Nothing.

Std::Unique_Lock::Mutex Member Function

Returns the mutex associated with `*this` if any.

Declaration

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

Returns

A pointer to the mutex associated with `*this` if any, `NULL` otherwise.

Throws

Nothing.

Std::Unique_Lock::Release Member Function

Returns the mutex associated with `*this` if any, and releases that association.

Declaration

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

Effects

Breaks the association of the mutex with `*this` without unlocking any locks held.

Returns

A pointer to the mutex associated with `*this` prior to the call if any, `NULL` otherwise.

Postconditions

`this->mutex()==NULL, this->owns_lock()==false.`

Throws

Nothing.

Note

If `this->owns_lock()` would have returned `true` prior to the call, the caller would now be responsible for unlocking the mutex.

D.5.7. std::lock function template

The `std::lock` function template provides a means of locking more than one mutex at the same time, without risk of deadlock resulting from inconsistent lock orders.

Declaration

```
template<typename LockableType1,typename... LockableType2>
void lock(LockableType1& m1,LockableType2& m2...);
```

Preconditions

The types of the supplied lockable objects `LockableType1, LockableType2`,... shall conform to the `Lockable` requirements.

Effects

Acquires a lock on each of the supplied lockable objects `m1, m2`,... by an unspecified sequence of calls to the `lock(), try_lock()`, and `unlock()` members of those types that avoid deadlock.

Postconditions

The current thread owns a lock on each of the supplied lockable objects.

Throws

Any exceptions thrown by the calls to `lock(), try_lock()`, and `unlock()`.

Note

If an exception propagates out of the call to `std::lock`, then `unlock()` shall have been called for any of the objects `m1, m2`,... for which a lock has been acquired in the function by a call to `lock()` or `try_lock()`.

D.5.8. std::try_lock function template

The `std::try_lock` function template allows you to try to lock a set of lockable objects in one go, so either they are all locked or none are locked.

Declaration

```
template<typename LockableType1,typename... LockableType2>
```

```
int try_lock(LockableType1& m1,LockableType2& m2...);
```

Preconditions

The types of the supplied lockable objects `LockableType1, LockableType2`,... shall conform to the `Lockable` requirements.

Effects

Tries to acquires a lock on each of the supplied lockable objects `m1, m2`,... by calling `try_lock()` on each in turn. If a call to `try_lock()` returns `false` or throws an exception, locks already acquired are released by calling `unlock()` on the corresponding lockable object.

Returns

`-1` if all locks were acquired (each call to `try_lock()` returned `true`), otherwise the zero-based index of the object for which the call to `try_lock()` returned `false`.

Postconditions

If the function returns `-1`, the current thread owns a lock on each of the supplied lockable objects. Otherwise, any locks acquired by this call have been released.

Throws

Any exceptions thrown by the calls to `try_lock()`.

Note

If an exception propagates out of the call to `std::try_lock`, then `unlock()` shall have been called for any of the objects `m1, m2`,... for which a lock has been acquired in the function by a call to `try_lock()`.

D.5.9. std::once_flag class

Instances of `std::once_flag` are used with `std::call_once` to ensure that a particular function is called exactly once, even if multiple threads invoke the call concurrently.

Instances of `std::once_flag` are not `CopyConstructible, CopyAssignable, Move-Constructible`, or `MoveAssignable`.

Class definition

```
struct once_flag
{
    constexpr once_flag() noexcept;

    once_flag(once_flag const& ) = delete;
    once_flag& operator=(once_flag const& ) = delete;
};
```

Std::Once_Flag Default Constructor

The `std::once_flag` default constructor creates a new `std::once_flag` instance in a state, which

indicates that the associated function hasn't been called.

Declaration

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

Effects

Constructs a new `std::once_flag` instance in a state, which indicates that the associated function hasn't been called. Because this is a `constexpr` constructor, an instance with static storage duration is constructed as part of the static initialization phase, which avoids race conditions and order-of-initialization problems.

D.5.10. std::call_once function template

`std::call_once` is used with an instance of `std::once_flag` to ensure that a particular function is called exactly once, even if multiple threads invoke the call concurrently.

Declaration

```
template<typename Callable,typename... Args>
void call_once(std::once_flag& flag,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

Invocations of `std::call_once` on the same `std::once_flag` object are serialized. If there has been no prior effective `std::call_once` invocation on the same `std::once_flag` object, the argument `func` (or a copy thereof) is called as-if by *INVOKE(*`func,args`*)*, and the invocation of `std::call_once` is effective if and only if the invocation of `func` returns without throwing an exception. If an exception is thrown, the exception is propagated to the caller. If there has been a prior effective `std::call_once` on the same `std::once_flag` object, the invocation of `std::call_once` returns without invoking `func`.

Synchronization

The completion of an effective `std::call_once` invocation on a `std::once_flag` object happens-before all subsequent `std::call_once` invocations on the same `std::once_flag` object.

Throws

`std::system_error` when the effects can't be achieved or for any exception propagated from the invocation of `func`.