
Thread 4.3.0

Anthony Williams

Vicente J. Botet Escriba

Copyright © 2007 -11 Anthony Williams

Copyright © 2011 -14 Vicente J. Botet Escriba

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Overview	3
Using and building the library	4
Configuration	5
Limitations	13
History	15
Future	27
Thread Management	28
Synopsis	28
Tutorial	28
Class <code>thread</code>	35
Namespace <code>this_thread</code>	47
Class <code>thread_group</code> EXTENSION	52
Scoped Threads	55
Motivation	55
Tutorial	55
Free Thread Functors	55
Class <code>strict_scoped_thread</code>	56
Class <code>scoped_thread</code>	58
Non-member function <code>swap(scoped_thread&,scoped_thread&)</code>	61
Synchronization	62
Tutorial	62
Mutex Concepts	75
Lock Options	89
Lock Guard	90
With Lock Guard	91
Lock Concepts	92
Lock Types	93
Other Lock Types - EXTENSION	105
Lock functions	118
Lock Factories - EXTENSION	120
Mutex Types	121
Condition Variables	128
One-time Initialization	139
Barriers -- EXTENSION	141
Latches -- EXPERIMENTAL	142
Executors and Schedulers -- EXPERIMENTAL	146
Futures	159
Thread Local Storage	198
Class <code>thread_specific_ptr</code>	199
Synchronized Data Structures	201
Synchronized Values - EXPERIMENTAL	201

Synchronized Queues -- EXPERIMENTAL	209
Time Requirements	222
Deprecated	222
Emulations	224
=delete emulation	224
Move semantics	224
Bool explicit conversion	229
Scoped Enums	229
Acknowledgments	231
Conformance and Extension	232
C++11 standard Thread library	232
C++14 standard Thread library - accepted changes	235
C++1y TS Concurrency - On going proposals	236

Overview

Boost.Thread enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with others for synchronizing data between the threads or providing separate copies of data specific to individual threads.

The **Boost.Thread** library was originally written and designed by William E. Kempf (version 1).

Anthony Williams version (version 2) was a major rewrite designed to closely follow the proposals presented to the C++ Standards Committee, in particular [N2497](#), [N2320](#), [N2184](#), [N2139](#), and [N2094](#)

Vicente J. Botet Escriba started (version 3) the adaptation to comply with the accepted Thread C++11 library (Make use of Boost.Chrono and Boost.Move) and the [Shared Locking](#) Howard Hinnant proposal except for the upward conversions. Some minor non-standard features have been added also as thread attributes, `reverse_lock`, `shared_lock_guard`.

In order to use the classes and functions described here, you can either include the specific headers specified by the descriptions of each class or function, or include the master thread library header:

```
#include <boost/thread.hpp>
```

which includes all the other headers in turn.

Using and building the library

Boost.Thread is configured following the conventions used to build [libraries with separate source code](#). Boost.Thread will import/export the code only if the user has specifically asked for it, by defining either `BOOST_ALL_DYN_LINK` if they want all boost libraries to be dynamically linked, or `BOOST_THREAD_DYN_LINK` if they want just this one to be dynamically linked.

The definition of these macros determines whether `BOOST_THREAD_USE_DLL` is defined. If `BOOST_THREAD_USE_DLL` is not defined, the library will define `BOOST_THREAD_USE_DLL` or `BOOST_THREAD_USE_LIB` depending on whether the platform. On non windows platforms `BOOST_THREAD_USE_LIB` is defined if is not defined. In windows platforms, `BOOST_THREAD_USE_LIB` is defined if `BOOST_THREAD_USE_DLL` and the compiler supports auto-tss cleanup with Boost.Threads (for the time been Msvc and Intel)

The source code compiled when building the library defines a macros `BOOST_THREAD_SOURCE` that is used to import or export it. The user must not define this macro in any case.

Boost.Thread depends on some non header-only libraries.

- Boost.System: This dependency is mandatory and you will need to link with the library.
- Boost.Chrono: This dependency is optional (see below how to configure) and you will need to link with the library if you use some of the time related interfaces.
- Boost.DateTime: This dependency is mandatory, but even if Boost.DateTime is a non header-only library Boost.Thread uses only parts that are header-only, so in principle you should not need to link with the library.

It seems that there are some IDE (as e.g. Visual Studio) that deduce the libraries that a program needs to link to inspecting the sources. Such IDE could force to link to Boost.DateTime and/or Boost.Chrono.

As the single mandatory dependency is to Boost.System, the following

```
bjam toolset=msvc-11.0 --build-type=complete --with-thread
```

will install only `boost_thread` and `boost_system`.

Users of such IDE should force the Boost.Chrono and Boost.DateTime build using

```
bjam toolset=msvc-11.0 --build-type=complete --with-thread --with-chrono --with-date_time
```

The following section describes all the macros used to configure Boost.Thread.

Configuration

Table 1. Default Values for Configurable Features

Feature	Anti-Feature	V2	V3	V4
USES_CHRONO	DONT_USE_CHRONO	YES/NO	YES/NO	YES/NO
PROVIDES_INTERRUPTIONS	DONT_PROVIDE_INTERRUPTIONS	YES	YES	YES
THROW_IF_PRECONDITION_NOT_SATISFIED	-	NO	NO	NO
PROVIDES_PROMISE_LAZY	DONT_PROVIDE_PROMISE_LAZY	YES	NO	NO
PROVIDES_BASIC_THREAD_ID	DONT_PROVIDE_BASIC_THREAD_ID	NO	YES	YES
PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN	DONT_PROVIDE_GENERIC_SHARED_MUTEX_ON_WIN	NO	YES	YES
PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION	DONT_PROVIDE_SHARED_MUTEX_UPWARDS_CONVERSION	NO	YES	YES
PROVIDES_EXPLICIT_LOCK_CONVERSION	DONT_PROVIDE_EXPLICIT_LOCK_CONVERSION	NO	YES	YES
PROVIDES_FUTURE	DONT_PROVIDE_FUTURE	NO	YES	YES
PROVIDES_FUTURE_CTOR_ALLOCATORS	DONT_PROVIDE_FUTURE_CTOR_ALLOCATORS	NO	YES	YES
PROVIDES_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE	DONT_PROVIDE_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE	NO	YES	YES
PROVIDES_THREAD_MOVE_ASIGN_CALLS_TERMINATE_IF_JOINABLE	DONT_PROVIDE_THREAD_MOVE_ASIGN_CALLS_TERMINATE_IF_JOINABLE	NO	YES	YES
PROVIDES_ONCE_CXX11	DONT_PROVIDE_ONCE_CXX11	NO	YES	YES
USES_MOVE	DONT_USE_MOVE	NO	YES	YES
USES_DATETIME	DONT_USE_DATETIME	YES/NO	YES/NO	YES/NO
PROVIDES_THREAD_EQ	DONT_PROVIDE_THREAD_EQ	YES	YES	NO

Feature	Anti-Feature	V2	V3	V4
PROVIDES_CONDITION	DONT_PROVIDE_CONDITION	YES	YES	NO
PROVIDES_NESTED_LOCKS	DONT_PROVIDE_NESTED_LOCKS	YES	YES	NO
PROVIDES_SIGNATURE_PACKAGE_TASK	DONT_PROVIDE_SIGNATURE_PACKAGE_TASK	NO	NO	YES
PROVIDES_FUTURE_INVALID_ID_AFTER_GET	DONT_PROVIDE_FUTURE_INVALID_ID_AFTER_GET	NO	NO	YES
PROVIDES_VARIADIC_THREAD	DONT_PROVIDE_VARIADIC_THREAD	NO	NO	C++11

Boost.Chrono

Boost.Thread uses by default Boost.Chrono for the time related functions and define `BOOST_THREAD_USES_CHRONO` if `BOOST_THREAD_DONT_USE_CHRONO` is not defined. The user should define `BOOST_THREAD_DONT_USE_CHRONO` for compilers that don't work well with Boost.Chrono.



Warning

When defined `BOOST_THREAD_PLATFORM_WIN32` `BOOST_THREAD_USES_CHRONO` is defined independently of user settings.

Boost.Move

Boost.Thread uses by default an internal move semantic implementation. Since version 3.0.0 you can use the move emulation emulation provided by Boost.Move.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_USES_MOVE` if you want to use Boost.Move interface. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_USE_MOVE` if you don't want to use Boost.Move interface.

Boost.DateTime

The Boost.DateTime time related functions introduced in Boost 1.35.0, using the [Boost.Date_Time](#) library are deprecated. These include (but are not limited to):

- `boost::this_thread::sleep()`
- `timed_join()`
- `timed_wait()`
- `timed_lock()`

When `BOOST_THREAD_VERSION<=3` && defined `BOOST_THREAD_PLATFORM_PTHREAD` define `BOOST_THREAD_DONT_USE_DATETIME` if you don't want to use Boost.DateTime related interfaces. When `BOOST_THREAD_VERSION>3` && defined `BOOST_THREAD_PLATFORM_PTHREAD` define `BOOST_THREAD_USES_DATETIME` if you want to use Boost.DateTime related interfaces.



Warning

When defined `BOOST_THREAD_PLATFORM_WIN32` `BOOST_THREAD_USES_DATETIME` is defined independently of user settings.

Boost.Atomic

Boost.Thread uses by default an Boost.Atomic in POSIX platforms to implement `call_once..`

Define `BOOST_THREAD_USES_ATOMIC` if you want to use Boost.Atomic. Define `BOOST_THREAD_DONT_USE_ATOMIC` if you don't want to use Boost.Atomic or if it is not supported in your platform.

`boost::thread::operator==` deprecated

The following operators are deprecated:

- `boost::thread::operator==`
- `boost::thread::operator!=`

When `BOOST_THREAD_PROVIDES_THREAD_EQ` is defined Boost.Thread provides these deprecated feature.

Use instead

- `boost::thread::id::operator==`
- `boost::thread::id::operator!=`



Warning

This is a breaking change respect to version 1.x.

When `BOOST_THREAD_VERSION>=4` define `BOOST_THREAD_PROVIDES_THREAD_EQ` if you want this feature. When `BOOST_THREAD_VERSION<4` define `BOOST_THREAD_DONT_PROVIDE_THREAD_EQ` if you don't want this feature.

`boost::condition` deprecated

`boost::condition` is deprecated. When `BOOST_THREAD_PROVIDES_CONDITION` is defined Boost.Thread provides this deprecated feature.

Use instead `boost::condition_variable_any`.



Warning

This is a breaking change respect to version 1.x.

When `BOOST_THREAD_VERSION>3` define `BOOST_THREAD_PROVIDES_CONDITION` if you want this feature. When `BOOST_THREAD_VERSION<=3` define `BOOST_THREAD_DONT_PROVIDE_CONDITION` if you don't want this feature.

Mutex nested lock types deprecated

The following nested typedefs are deprecated:

- `boost::mutex::scoped_lock`,

- `boost::mutex::scoped_try_lock`,
- `boost::timed_mutex::scoped_lock`
- `boost::timed_mutex::scoped_try_lock`
- `boost::timed_mutex::timed_scoped_timed_lock`
- `boost::recursive_mutex::scoped_lock`,
- `boost::recursive_mutex::scoped_try_lock`,
- `boost::recursive_timed_mutex::scoped_lock`
- `boost::recursive_timed_mutex::scoped_try_lock`
- `boost::recursive_timed_mutex::timed_scoped_timed_lock`

When `BOOST_THREAD_PROVIDES_NESTED_LOCKS` is defined Boost.Thread provides these deprecated feature.

Use instead `* boost::unique_lock<boost::mutex>`, `* boost::unique_lock<boost::mutex>` with the `try_to_lock_t` constructor, `* boost::unique_lock<boost::timed_mutex>` `* boost::unique_lock<boost::timed_mutex>` with the `try_to_lock_t` constructor `* boost::unique_lock<boost::timed_mutex>` `* boost::unique_lock<boost::recursive_mutex>`, `* boost::unique_lock<boost::recursive_mutex>` with the `try_to_lock_t` constructor, `* boost::unique_lock<boost::recursive_timed_mutex>` `* boost::unique_lock<boost::recursive_timed_mutex>` with the `try_to_lock_t` constructor `* boost::unique_lock<boost::recursive_timed_mutex>`



Warning

This is a breaking change respect to version 1.x.

When `BOOST_THREAD_VERSION`>=4 define `BOOST_THREAD_PROVIDES_NESTED_LOCKS` if you want these features. When `BOOST_THREAD_VERSION`<4 define `BOOST_THREAD_DONT_PROVIDE_NESTED_LOCKS` if you don't want thes features.

thread::id

Boost.Thread uses by default a `thread::id` on Posix based on the `pthread_t` type (`BOOST_THREAD_PROVIDES_BASIC_THREAD_ID`). For backward compatibility and also for compilers that don't work well with this modification the user can define `BOOST_THREAD_DONT_PROVIDE_BASIC_THREAD_ID`.

Define `BOOST_THREAD_DONT_PROVIDE_BASIC_THREAD_ID` if you don't want these features.

Shared Locking Generic

The shared mutex implementation on Windows platform provides currently less functionality than the generic one that is used for PThreads based platforms. In order to have access to these functions, the user needs to define `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` to use the generic implementation, that while could be less efficient, provides all the functions.

When `BOOST_THREAD_VERSION`==2 define `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` if you want these features. When `BOOST_THREAD_VERSION`>=3 define `BOOST_THREAD_DONT_PROVIDE_GENERIC_SHARED_MUTEX_ON_WIN` if you don't want these features.

Shared Locking Upwards Conversion

Boost.Threads includes in version 3 the Shared Locking Upwards Conversion as defined in [Shared Locking](#). These conversions need to be used carefully to avoid deadlock or livelock. The user need to define explicitly `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` to get these upwards conversions.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` if you want these features. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_PROVIDE_SHARED_MUTEX_UPWARDS_CONVERSION` if you don't want these features.

Explicit Lock Conversion

In [Shared Locking](#) the lock conversions are explicit. As this explicit conversion breaks the lock interfaces, it is provided only if the `BOOST_THREAD_PROVIDES_EXPLICIT_LOCK_CONVERSION` is defined.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_EXPLICIT_LOCK_CONVERSION` if you want these features. When `BOOST_THREAD_VERSION==3` define `BOOST_THREAD_DONT_PROVIDE_EXPLICIT_LOCK_CONVERSION` if you don't want these features.

unique_future versus future

C++11 uses `std::future`. Versions of Boost.Thread previous to version 3.0.0 uses `boost::unique_future`. Since version 3.0.0 `boost::future` replaces `boost::unique_future` when `BOOST_THREAD_PROVIDES_FUTURE` is defined. The documentation doesn't contains anymore however `boost::unique_future`.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_FUTURE` if you want to use `boost::future`. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_PROVIDE_FUTURE` if you want to use `boost::unique_future`.

promise lazy initialization

C++11 promise initialize the associated state at construction time. Versions of Boost.Thread previous to version 3.0.0 initialize it lazily at any point in time in which this associated state is needed.

Since version 3.0.0 this difference in behavior can be configured. When `BOOST_THREAD_PROVIDES_PROMISE_LAZY` is defined the backward compatible behavior is provided.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_DONT_PROVIDE_PROMISE_LAZY` if you want to use `boost::future`. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_PROVIDES_PROMISE_LAZY` if you want to use `boost::unique_future`.

promise Allocator constructor

C++11 `std::promise` provides constructors with allocators.

```
template <typename R>
class promise
{
public:
    template <class Allocator>
        explicit promise(allocator_arg_t, Allocator a);
    // ...
};
template <class R, class Alloc> struct uses_allocator<promise<R>,Alloc>: true_type {};
```

where

```
struct allocator_arg_t { };
constexpr allocator_arg_t allocator_arg = allocator_arg_t();

template <class T, class Alloc> struct uses_allocator;
```

Since version 3.0.0 Boost.Thread implements this constructor using the following interface

```

namespace boost
{
    typedef container::allocator_arg_t allocator_arg_t;
    constexpr allocator_arg_t allocator_arg = {};

    namespace container
    {
        template <class R, class Alloc>
        struct uses_allocator<promise<R>, Alloc>: true_type {};
    }
    template <class T, class Alloc>
    struct uses_allocator : public container::uses_allocator<T, Alloc> {};
}

```

which introduces a dependency on Boost.Container. This feature is provided only if `BOOST_THREAD_PROVIDES_FUTURE_CTOR_ALLOCATORS` is defined.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_FUTURE_CTOR_ALLOCATORS` if you want these features. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_PROVIDE_FUTURE_CTOR_ALLOCATORS` if you don't want these features.

Call to terminate if joinable

C++11 has a different semantic for the thread destructor and the move assignment. Instead of detaching the thread, calls to `terminate()` if the thread was joinable. When `BOOST_THREAD_PROVIDES_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE` and `BOOST_THREAD_PROVIDES_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE` is defined Boost.Thread provides the C++ semantic.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE` if you want these features. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_PROVIDE_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE` if you don't want these features.

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE` if you want these features. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_PROVIDE_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE` if you don't want these features.

once_flag

C++11 defines a default constructor for `once_flag`. When `BOOST_THREAD_PROVIDES_ONCE_CXX11` is defined Boost.Thread provides this C++ semantics. In this case, the previous aggregate syntax is not supported.

```
boost::once_flag once = BOOST_ONCE_INIT;
```

You should now just do

```
boost::once_flag once;
```

When `BOOST_THREAD_VERSION==2` define `BOOST_THREAD_PROVIDES_ONCE_CXX11` if you want these features. When `BOOST_THREAD_VERSION>=3` define `BOOST_THREAD_DONT_PROVIDE_ONCE_CXX11` if you don't want these features.

Signature parameter for packaged_task

C++11 packaged task class has a Signature template parameter. When `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` is defined Boost.Thread provides this C++ feature.



Warning

This is a breaking change respect to version 3.x.

When `BOOST_THREAD_VERSION < 4` define `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` if you want this feature. When `BOOST_THREAD_VERSION >= 4` define `BOOST_THREAD_DONT_PROVIDE_SIGNATURE_PACKAGED_TASK` if you don't want this feature.

-var thread constructor with variadic rvalue parameters

C++11 thread constructor accept a variable number of rvalue arguments. When `BOOST_THREAD_PROVIDES_VARIADIC_THREAD` is defined `Boost.Thread` provides this C++ feature if the following are not defined

- `BOOST_NO_SFINAE_EXPR`
- `BOOST_NO_CXX11_VARIADIC_TEMPLATES`
- `BOOST_NO_CXX11_DECLTYPE`
- `BOOST_NO_CXX11_DECLTYPE_N3276`
- `BOOST_NO_CXX11_RVALUE_REFERENCES`
- `BOOST_NO_CXX11_TRAILING_RESULT_TYPES`
- `BOOST_NO_CXX11_RVALUE_REFERENCES`
- `BOOST_NO_CXX11_HDR_TUPLE`

When `BOOST_THREAD_VERSION > 4` define `BOOST_THREAD_DONT_PROVIDE_VARIADIC_THREAD` if you don't want this feature.

future<>::get() invalidates the future

C++11 `future<>::get()` invalidates the future once its value has been obtained. When `BOOST_THREAD_PROVIDES_FUTURE_INVALID_ID_AFTER_GET` is defined `Boost.Thread` provides this C++ feature.



Warning

This is a breaking change respect to version 3.x.

When `BOOST_THREAD_VERSION < 4` define `BOOST_THREAD_PROVIDES_FUTURE_INVALID_AFTER_GET` if you want this feature. When `BOOST_THREAD_VERSION >= 4` define `BOOST_THREAD_DONT_PROVIDE_FUTURE_INVALID_AFTER_GET` if you don't want this feature.

Interruptions

Thread interruption, while useful, makes any interruption point less efficient than if the thread were not interruptible.

When `BOOST_THREAD_PROVIDES_INTERRUPTIONS` is defined `Boost.Thread` provides interruptions. When `BOOST_THREAD_DONT_PROVIDE_INTERRUPTIONS` is defined `Boost.Thread` don't provide interruption.

`Boost.Thread` defines `BOOST_THREAD_PROVIDES_INTERRUPTIONS` if neither `BOOST_THREAD_PROVIDES_INTERRUPTIONS` nor `BOOST_THREAD_DONT_PROVIDE_INTERRUPTIONS` are defined, so that there is no compatibility break.

Version

`BOOST_THREAD_VERSION` defines the Boost.Thread version. The default version is 2. In this case the following breaking or extending macros are defined if the opposite is not requested:

- `BOOST_THREAD_PROVIDES_PROMISE_LAZY`

The user can request the version 3 by defining `BOOST_THREAD_VERSION` to 3. In this case the following breaking or extending macros are defined if the opposite is not requested:

- Breaking change `BOOST_THREAD_PROVIDES_EXPLICIT_LOCK_CONVERSION`
- Conformity & Breaking change `BOOST_THREAD_PROVIDES_FUTURE`
- Uniformity `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN`
- Extension `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION`
- Conformity `BOOST_THREAD_PROVIDES_FUTURE_CTOR_ALLOCATORS`
- Conformity & Breaking change `BOOST_THREAD_PROVIDES_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE`
- Conformity & Breaking change `BOOST_THREAD_PROVIDES_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE`
- Conformity & Breaking change `BOOST_THREAD_PROVIDES_ONCE_CXX11`
- Breaking change `BOOST_THREAD_DONT_PROVIDE_PROMISE_LAZY`

The user can request the version 4 by defining `BOOST_THREAD_VERSION` to 4. In this case the following breaking or extending macros are defined if the opposite is not requested:

- Conformity & Breaking change `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK`
- Conformity & Breaking change `BOOST_THREAD_PROVIDES_FUTURE_INVALID_AFTER_GET`
- Conformity `BOOST_THREAD_PROVIDES_VARIADIC_THREAD`
- Breaking change `BOOST_THREAD_DONT_PROVIDE_THREAD_EQ`
- Breaking change `BOOST_THREAD_DONT_USE_DATETIME`

Limitations

Some compilers don't work correctly with some of the added features.

SunPro

If `__SUNPRO_CC < 0x5100` the library defines

- `BOOST_THREAD_DONT_USE_MOVE`

If `__SUNPRO_CC < 0x5100` the library defines

- `BOOST_THREAD_DONT_PROVIDE_FUTURE_CTOR_ALLOCATORS`

VACPP

If `__IBMCPP__ < 1100` the library defines

- BOOST_THREAD_DONT_USE_CHRONO
- BOOST_THREAD_USES_DATE

And Boost.Thread doesn't links with Boost.Chrono.

WCE

If _WIN32_WCE && _WIN32_WCE==0x501 the library defines

- BOOST_THREAD_DONT_PROVIDE_FUTURE_CTOR_ALLOCATORS

History

Version 4.3.0 - boost 1.56

Know Bugs:

- [#2442](#) Application statically linked with Boost.Thread crashes when Google Desktop is installed (Windows XP)
- [#3926](#) thread_specific_ptr + dlopen library causes a SIGSEGV.
- [#4833](#) MinGW/test_tss_lib: Support of automatic tss cleanup for native threading API not available
- [#6782](#) call_once uses incorrect barrier intrinsic on Visual Studio
- [#7319](#) Take care of c++std-lib-32966 issue
- [#8600](#) wait_for_any hangs, if called with multiple copies of shared_future referencing same task
- [#9307](#) future::fallback_to assert with ERRORRRRRR boost: mutex lock failed in pthread_mutex_lock: Invalid argument
- [#9308](#) future::async fails with terminate called throwing an exception when called with a lambda - clang-darwin-asan11
- [#9310](#) test_4648_lib fails on clang-darwin-asan11
- [#9311](#) ex_lambda_future fails on msvc-11.0
- [#9425](#) Boost promise & future does not use supplied allocator for value storage
- [#9558](#) future continuations unit test hangs in get()/pthread_cond_wait() on Mac 10.7/32-bit/x86/darwin-4.2.1

Please take a look at [thread Know Bugs](#) to see the current state.

Please take a look at [thread trunk regression test](#) to see the last snapshot.

Sever limitations:

There are some severe bugs that prevent the use of the library on concrete contexts, in particular:

- on thread specific storage that prevent the library to be used with dynamic libraries,
- The experimental features of boost::future have some severe holes that make the program crash unexpectedly.

New Experimental Features:

- [#7446](#) Async: Add when_any.
- [#7447](#) Async: Add when_all.
- [#7448](#) Async: Add async taking a scheduler parameter.
- [#8274](#) Synchro: Add concurrent queue
- [#8518](#) Synchro: Add a latch class.
- [#8513](#) Async: Add a basic thread_pool executor.
- [#8516](#) Async: Add future/shared_future::then taking a scheduler as parameter.

Fixed Bugs:

- [#8070](#) prefer GetTickCount64 over GetTickCount

- [#9333](#) `ex_scoped_thread` compile fails on msvc-12.0
- [#9366](#) `async(Executor, ...)` fails to compile with msvc-10,11,12
- [#9402](#) `test_excutor` regression on msvc-10,11,12
- [#9404](#) `ex_make_future` regression error
- [#9471](#) Synchronization documentation nits
- [#9535](#) Missing exception safety might result in crash
- [#9618](#) `try_join_for` problem: program is not terminate.
- [#9673](#) thread compilation with MingW/gcc on Windows gives errors
- [#9708](#) `boost::condition_variable::timed_wait` unexpectedly wakes up while should wait infinite
- [#9711](#) future continuation called twice

Version 4.2.0 - boost 1.55

Know Bugs:

- [#2442](#) Application statically linked with Boost.Thread crashes when Google Desktop is installed (Windows XP)
- [#3926](#) `thread_specific_ptr` + `dlopen` library causes a SIGSEGV.
- [#4833](#) MinGW/test_tss_lib: Support of automatic tss cleanup for native threading API not available
- [#6782](#) `call_once` uses incorrect barrier intrinsic on Visual Studio
- [#7319](#) Take care of c++std-lib-32966 issue
- [#8600](#) `wait_for_any` hangs, if called with multiple copies of `shared_future` referencing same task
- [#9307](#) `future::fallback_to` assert with ERRORRRRRR boost: mutex lock failed in `pthread_mutex_lock`: Invalid argument
- [#9308](#) `future::async` fails with terminate called throwing an exception when called with a lambda - clang-darwin-asan11
- [#9311](#) `ex_lambda_future` fails on msvc-11.0
- [#9310](#) `test_4648_lib` fails on clang-darwin-asan11

Please take a look at [thread trunk regression test](#) to see the current state.

Sever limitations:

There are some severe bugs that prevent the use of the library on concrete contexts, in particular:

- on thread specific storage that prevent the library to be used with dynamic libraries,
- The experimental features of `boost::future` have some severe holes that make the program crash unexpectedly.

New Features:

- [#8519](#) Synchro: Update class barrier with a completion function.
- [#8515](#) Async: Add `shared_future::then`.
- [#8519](#) Synchro: Update class barrier with a completion function

- [#8615](#) Async: Replace `make_future/make_shared_future` by `make_ready_future`.
- [#8627](#) Async: Add `future<>::unwrap` and `unwrapping` constructor.
- [#8677](#) Async: Add `future<>::get_or`.
- [#8678](#) Async: Add `future<>::fallback_to`.
- [#8891](#) `upgrade_to_unique_lock`: missing `mutex()` function.
- [#8955](#) Request for more efficient way to get `exception_ptr` from future.

Fixed Bugs:

- [#7461](#) `detail::win32::ReleaseSemaphore` may be called with `count_to_release` equal to 0
- [#8070](#) prefer `GetTickCount64` over `GetTickCount`
- [#8768](#) `win32 condition_variable::wait_until` infinite wait in rare cases.
- [#8817](#) Boost Thread Windows CE `_createthreadex` handling breaks mingw w64.
- [#8943](#) Failed to compile code using `boost::call_once` with Intel C++ Composer XE 2013 on Windows.
- [#8931](#) Typos in `external_locking` reference.
- [#9029](#) Misprint in documentation.
- [#9037](#) `gcc -Wshadow` gives warnings in `condition_variable{,_fwd}.hpp`.
- [#9041](#) `Boost.Thread` DSO's may need to link with `Boost.Atomic`.
- [#9048](#) `boost::scoped_thread` useless ctor with variadic template arguments.
- [#9079](#) Condition variable will wait forever for some timepoint values (Win).

Version 4.1.0 - boost 1.54

New Features:

- [#7285](#) C++11 compliance: Allow to pass movable arguments for `call_once`.
- [#7445](#) Async: Add `future<>.then`
- [#7449](#) Synchro: Add a synchronized value class

Fixed Bugs:

- [#4878](#) MinGW 4.5.0 undefined reference to `bool interruptible_wait(detail::win32::handle handle_to_wait_for, detail::timeout target_time)`.
- [#4882](#) Win32 `shared_mutex` does not handle timeouts correctly.
- [#5752](#) `boost::call_once()` is unreliable on some platforms
- [#6652](#) `Boost.Thread shared_mutex.hpp:50:99`: warning: dereferencing type-punned pointer will break strict-aliasing rules ~~Wstrict-aliasing~~
- [#6843](#) [Intel C++] Compile Errors with `'#include <atomic>'`
- [#6966](#) future `boost::future_category` inconsistent dll linkage

- [#7720](#) exception lock_error while intensive locking/unlocking of mutex
- [#7755](#) Thread: deadlock with shared_mutex on Windows
- [#7980](#) Build error: msvc-11.0 and BOOST_THREAD_DONT_USE_DATETIME
- [#7982](#) pthread_delay_np() parm compile error on AIX
- [#8027](#) thread library fails to compile with Visual Studio 2003
- [#8070](#) prefer GetTickCount64 over GetTickCount
- [#8136](#) boost::this_thread::sleep_for() sleeps longer than it should in Windows
- [#8212](#) Boost thread compilation error on Solaris 10
- [#8237](#) fix documentation for 'thread_group'
- [#8239](#) barrier::wait() not marked as interruption_point
- [#8323](#) boost::thread::try_join_for/try_join_until may block indefinitely due to a combination of problems in Boost.Thread and Boost.Chrono
- [#8337](#) The internal representation of "std::string(this->code()->message())" escapes, but is destroyed when it exits scope.
- [#8371](#) C++11 once_flag enabled when constexpr is not available
- [#8422](#) Assertion in win32::WaitForSingleObject()
- [#8443](#) Header file inclusion order may cause crashes
- [#8451](#) Missing documented function 'boost::scoped_thread::joinable'
- [#8458](#) -DBOOST_THREAD_DONT_USE_CHRONO in thread.obj.rsp but not explicitly set
- [#8530](#) [Coverity] Unused variable thread_handle, uninitialized variable cond_mutex in thread/pthread/thread_data.hpp
- [#8550](#) static linking of Boost.Thread with an MFC-Dll
- [#8576](#) "sur parole" should be "sur parole".
- [#8596](#) With C++0x enabled, boost::packaged_task stores a reference to function objects, instead of a copy.
- [#8626](#) Reintroduce BOOST_VERIFY on pthread_mutex_destroy return type
- [#8645](#) Typo in Strict lock definition
- [#8671](#) promise: set_..._at_thread_exit
- [#8672](#) future<>::then(void()) doesn't work
- [#8674](#) Futures as local named objects can't be returned with implicit move.

Version 4.0.0 - boost 1.53

Deprecated features:



Warning

Deprecated features since boost 1.53 will be available only until boost 1.58.

- C++11 compliance: `packaged_task<R>` is deprecated, use instead `packaged_task<R()>`. See `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` and `BOOST_THREAD_DONT_PROVIDE_SIGNATURE_PACKAGED_TASK`
- #7537 deprecate `Mutex::scoped_lock` and `scoped_try_lock` and `boost::condition`

New Features:

- #6270 C++11 compliance: Add thread constructor from movable callable and movable arguments Provided when `BOOST_THREAD_PROVIDES_VARIADIC_THREAD` is defined (Default value from Boost 1.55): See `BOOST_THREAD_PROVIDES_VARIADIC_THREAD` and `BOOST_THREAD_DONT_PROVIDE_VARIADIC_THREAD`.
- #7279 C++11 compliance: Add `noexcept` in system related functions
- #7280 C++11 compliance: Add `promise::...at_thread_exit` functions
- #7281 C++11 compliance: Add `ArgTypes` to `packaged_task` template. Provided when `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` is defined (Default value from Boost 1.55). See `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` and `BOOST_THREAD_DONT_PROVIDE_SIGNATURE_PACKAGED_TASK`.
- #7282 C++11 compliance: Add `packaged_task::make_ready_at_thread_exit` function
- #7285 C++11 compliance: Allow to pass movable arguments for `call_once`
- #7412 C++11 compliance: Add `async` from movable callable and movable arguments Provided when `BOOST_THREAD_PROVIDES_VARIADIC_THREAD` and `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` are defined (Default value from Boost 1.55): See `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` and `BOOST_THREAD_DONT_PROVIDE_SIGNATURE_PACKAGED_TASK`, `BOOST_THREAD_PROVIDES_VARIADIC_THREAD` and `BOOST_THREAD_DONT_PROVIDE_VARIADIC_THREAD`.
- #7413 C++11 compliance: Add `async` when the launch policy is deferred.
- #7414 C++11 compliance: `future::get` post-condition should be `valid()==false`.
- #7422 Provide a condition variable with zero-overhead performance penalty.
- #7444 Async: Add `make_future/make_shared_future`.
- #7540 Threads: Add a helper class that join a thread on destruction.
- #7541 Threads: Add a thread wrapper class that joins on destruction.
- #7575 C++11 compliance: A future created by `async` should "join" in the destructor.
- #7587 Synchro: Add `strict_lock` and `nested_strict_lock`.
- #7588 Synchro: Split the `locks.hpp` in several files to limit dependencies.
- #7590 Synchro: Add lockable concept checkers based on `Boost::ConceptCheck`.
- #7591 Add lockable traits that can be used with `enable_if`.
- #7592 Synchro: Add a `null_mutex` that is a no-op and that is a model of `UpgradeLockable`.
- #7593 Synchro: Add a `externally_locked` class.
- #7594 Threads: Allow to disable thread interruptions.

Fixed Bugs:

- #5752 `boost::call_once()` is unreliable on some platforms

- [#7464](#) BOOST_TEST(n_alive == 1); fails due to race condition in a regression test tool.
- [#7657](#) Serious performance and memory consumption hit if condition_variable methods condition_notify_one or notify_all is used repeatedly.
- [#7665](#) this_thread::sleep_for no longer uses steady_clock in thread.
- [#7668](#) thread_group::join_all() should check whether its threads are joinable.
- [#7669](#) thread_group::join_all() should catch resource_deadlock_would_occur.
- [#7671](#) Error including boost/thread.hpp header on iOS.
- [#7672](#) lockable_traits.hpp syntax error: "defined" token misspelled.
- [#7798](#) boost::future set_wait_callback thread safety issues.
- [#7808](#) Incorrect description of effects for this_thread::sleep_for and this_thread::sleep_until.
- [#7812](#) Returns: cv_status::no_timeout if the call is returning because the time period specified by rel_time has elapsed, cv_status::timeout otherwise.
- [#7874](#) compile warning: thread.hpp:342: warning: type attributes are honored only at type definition.
- [#7875](#) BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED should not be enabled by default.
- [#7882](#) wrong exception text from condition_variable::wait(unique_lock<mutex>&).
- [#7890](#) thread::do_try_join_until() is missing a return type.

Version 3.1.0 - boost 1.52

Deprecated Features:

Deprecated features since boost 1.50 available only until boost 1.55:

These deprecated features will be provided by default up to boost 1.52. If you don't want to include the deprecated features you could define BOOST_THREAD_DONT_PROVIDE_DEPRECATED_FEATURES_SINCE_V3_0_0. Since 1.53 these features will not be included any more by default. Since this version, if you want to include the deprecated features yet you could define BOOST_THREAD_PROVIDE_DEPRECATED_FEATURES_SINCE_V3_0_0. These deprecated features will be only available until boost 1.55, that is you have yet 1 year to move to the new features.

- Time related functions don't using the Boost.Chrono library, use the chrono overloads instead.

Breaking changes when BOOST_THREAD_VERSION==3 (Default value since Boost 1.53):

There are some new features which share the same interface but with different behavior. These breaking features are provided by default when BOOST_THREAD_VERSION is 3, but the user can however choose the version 2 behavior by defining the corresponding macro. As for the deprecated features, these broken features will be only available until boost 1.55.

- [#6229](#) Rename the unique_future to future following the c++11.
- [#6266](#) Breaking change: thread destructor should call terminate if joinable.
- [#6269](#) Breaking change: thread move assignment should call terminate if joinable.

New Features:

- [#2361](#) thread_specific_ptr: document nature of the key, complexity and rationale.
- [#4710](#) C++11 compliance: Missing async().

- [#7283](#) C++11 compliance: Add `notify_all_at_thread_exit`.
- [#7345](#) C++11 compliance: Add `noexcept` to recursive mutex `try_lock`.

Fixed Bugs:

- [#2797](#) Two problems with `thread_specific_ptr`.
- [#5274](#) failed to compile `future.hpp` with `stlport 5.1.5` under `msvc8.1`, because of undefined class.
- [#5431](#) compile error in Windows CE 6.0(interlocked).
- [#5696](#) `win32_detail::set_tss_data` does nothing when `tss_cleanup_function` is `NULL`.
- [#6931](#) mutex waits forever with Intel C++ Compiler XE 12.1.5.344 Build 20120612
- [#7045](#) Thread library does not automatically compile `date_time`.
- [#7173](#) wrong function name `interrupt_point()`.
- [#7200](#) Unable to build `boost.thread` modularized.
- [#7220](#) gcc 4.6.2 warns about `inline+dllimport` functions.
- [#7238](#) `this_thread::sleep_for()` does not respond to `interrupt()`.
- [#7245](#) Minor typos on documentation related to version 3.
- [#7272](#) `win32/thread_primitives.hpp`: (Unnecessary) Warning.
- [#7284](#) Clarify that there is no access priority between `lock` and `shared_lock` on shared mutex.
- [#7329](#) `boost/thread/future.hpp` does not compile on HP-UX.
- [#7336](#) `BOOST_THREAD_DONT_USE_SYSTEM` doesn't work.
- [#7349](#) `packaged_task` holds reference to temporary.
- [#7350](#) `allocator_destructor` does not destroy object
- [#7360](#) Memory leak in pthread implementation of `boost::thread_specific_ptr`
- [#7370](#) Boost.Thread documentation
- [#7438](#) Segmentation fault in `test_once` regression test in `group.join_all()`;
- [#7461](#) `detail::win32::ReleaseSemaphore` may be called with `count_to_release` equal to 0
- [#7499](#) `call_once` doesn't call even once

Version 3.0.1 - boost 1.51

Deprecated Features:

Deprecated features since boost 1.50 available only until boost 1.55:

These deprecated features will be provided by default up to boost 1.52. If you don't want to include the deprecated features you could define `BOOST_THREAD_DONT_PROVIDE_DEPRECATED_FEATURES_SINCE_V3_0_0`. Since 1.53 these features will not be included any more by default. Since this version, if you want to include the deprecated features yet you could define `BOOST_THREAD_PROVIDE_DEPRECATED_FEATURES_SINCE_V3_0_0`. These deprecated features will be only available until boost 1.55, that is you have 1 year and a half to move to the new features.

- Time related functions don't using the Boost.Chrono library, use the chrono overloads instead.

Breaking changes when BOOST_THREAD_VERSION==3:

There are some new features which share the same interface but with different behavior. These breaking features are provided by default when BOOST_THREAD_VERSION is 3, but the user can however choose the version 2 behavior by defining the corresponding macro. As for the deprecated features, these broken features will be only available until boost 1.55.

- [#6229](#) Rename the unique_future to future following the c++11.
- [#6266](#) Breaking change: thread destructor should call terminate if joinable.
- [#6269](#) Breaking change: thread move assignment should call terminate if joinable.

Fixed Bugs:

- [#4258](#) Linking with boost thread does not work on mingw/gcc 4.5.
- [#4885](#) Access violation in set_tss_data at process exit due to invalid assumption about TlsAlloc.
- [#6931](#) mutex waits forever with Intel Compiler and /debug:parallel
- [#7044](#) boost 1.50.0 header missing.
- [#7052](#) Thread: BOOST_THREAD_PROVIDES_DEPRECATED_FEATURES_SINCE_V3_0_0 only masks thread::operator==, thread::operator!= forward declarations, not definitions.
- [#7066](#) An attempt to fix current_thread_tls_key static initialization order.
- [#7074](#) Multiply defined symbol boost::allocator_arg.
- [#7078](#) Trivial 64-bit warning fix on Windows for thread attribute stack size
- [#7089](#) BOOST_THREAD_WAIT_BUG limits functionality without solving anything

Version 3.0.0 - boost 1.50

Breaking changes when BOOST_THREAD_VERSION==3:

- [#6229](#) Breaking change: Rename the unique_future to future following the c++11.
- [#6266](#) Breaking change: thread destructor should call terminate if joinable.
- [#6269](#) Breaking change: thread move assignment should call terminate if joinable.

New Features:

- [#1850](#) Request for unlock_guard to compliment lock_guard.
- [#2637](#) Request for shared_mutex duration timed_lock and timed_lock_shared.
- [#2741](#) Proposal to manage portable and non portable thread attributes.
- [#3567](#) Request for shared_lock_guard.
- [#6194](#) Adapt to Boost.Move.
- [#6195](#) c++11 compliance: Provide the standard time related interface using Boost.Chrono.
- [#6217](#) Enhance Boost.Thread shared mutex interface following Howard Hinnant proposal.
- [#6224](#) c++11 compliance: Add the use of standard noexcept on compilers supporting them.

- [#6225](#) Add the use of standard =delete defaulted operations on compilers supporting them.
- [#6226](#) c++11 compliance: Add explicit bool conversion from locks.
- [#6228](#) Add promise constructor with allocator following the standard c++11.
- [#6230](#) c++11 compliance: Follows the exception reporting mechanism as defined in the c++11.
- [#6231](#) Add BasicLockable requirements in the documentation to follow c++11.
- [#6272](#) c++11 compliance: Add thread::id hash specialization.
- [#6273](#) c++11 compliance: Add cv_status enum class and use it on the conditions wait functions.
- [#6342](#) c++11 compliance: Adapt the one_flag to the c++11 interface.
- [#6671](#) upgrade_lock: missing mutex and release functions.
- [#6672](#) upgrade_lock:: missing constructors from time related types.
- [#6675](#) upgrade_lock:: missing non-member swap.
- [#6676](#) lock conversion should be explicit.
- Added missing packaged_task::result_type and packaged_task:: constructor with allocator.
- Added packaged_task::reset()

Fixed Bugs:

- [#2380](#) boost::move from lvalue does not work with gcc.
- [#2430](#) shared_mutex for win32 doesn't have timed_lock_upgrade.
- [#2575](#) Bug- Boost 1.36.0 on Itanium platform.
- [#3160](#) Duplicate tutorial code in boost::thread.
- [#4345](#) thread::id and joining problem with cascade of threads.
- [#4521](#) Error using boost::move on packaged_task (MSVC 10).
- [#4711](#) Must use implementation details to return move-only types.
- [#4921](#) BOOST_THREAD_USE_DLL and BOOST_THREAD_USE_LIB are crucial and need to be documented.
- [#5013](#) documentation: boost::thread: pthreads_exit causes terminate().
- [#5173](#) boost::this_thread::get_id is very slow.
- [#5351](#) interrupt a future get boost::unknown_exception.
- [#5516](#) Upgrade lock is not acquired when previous upgrade lock releases if another read lock is present.
- [#5990](#) shared_future<T>::get() has wrong return type.
- [#6174](#) packaged_task doesn't correctly handle moving results.
- [#6222](#) Compile error with SunStudio: unique_future move.
- [#6354](#) PGI: Compiler threading support is not turned on.
- [#6673](#) shared_lock: move assign doesn't works with c++11.

- [#6674](#) shared_mutex: try_lock_upgrade_until doesn't work.
- [#6908](#) Compile error due to unprotected definitions of _WIN32_WINNT and WINVER.
- [#6940](#) TIME_UTC is a macro in C11.
- [#6959](#) call of abs is ambiguous.
- Fix issue signaled on the ML with task_object(task_object const&) in presence of task_object(task_object &&)

Version 2.1.1 - boost 1.49

Fixed Bugs:

- [#2309](#) Lack of g++ symbol visibility support in Boost.Thread.
- [#2639](#) documentation should be extended(defer_lock, try_to_lock, ...).
- [#3639](#) Boost.Thread doesn't build with Sun-5.9 on Linux.
- [#3762](#) Thread can't be compiled with winscw (Codewarrior by Nokia).
- [#3885](#) document about mix usage of boost.thread and native thread api.
- [#3975](#) Incorrect precondition for promise::set_wait_callback().
- [#4048](#) thread::id formatting involves locale
- [#4315](#) gcc 4.4 Warning: inline ... declared as dllimport: attribute ignored.
- [#4480](#) OpenVMS patches for compiler issues workarounds.
- [#4819](#) boost.thread's documentation misprints.
- [#5423](#) thread issues with C++0x.
- [#5617](#) boost::thread::id copy ctor.
- [#5739](#) set-but-not-used warnings with gcc-4.6.
- [#5826](#) threads.cpp: resource leak on threads creation failure.
- [#5839](#) thread.cpp: ThreadProxy leaks on exceptions.
- [#5859](#) win32 shared_mutex constructor leaks on exceptions.
- [#6100](#) Compute hardware_concurrency() using get_nprocs() on GLIBC systems.
- [#6168](#) recursive_mutex is using wrong config symbol (possible typo).
- [#6175](#) Compile error with SunStudio.
- [#6200](#) patch to have condition_variable and mutex error better handle EINTR.
- [#6207](#) shared_lock swap compiler error on clang 3.0 c++11.
- [#6208](#) try_lock_wrapper swap compiler error on clang 3.0 c++11.

Version 2.1.0 - Changes since boost 1.40

The 1.41.0 release of Boost adds futures to the thread library. There are also a few minor changes.

Changes since boost 1.35

The 1.36.0 release of Boost includes a few new features in the thread library:

- New generic `lock()` and `try_lock()` functions for locking multiple mutexes at once.
- Rvalue reference support for move semantics where the compilers supports it.
- A few bugs fixed and missing functions added (including the serious win32 condition variable bug).
- `scoped_try_lock` types are now backwards-compatible with Boost 1.34.0 and previous releases.
- Support for passing function arguments to the thread function by supplying additional arguments to the `boost::thread` constructor.
- Backwards-compatibility overloads added for `timed_lock` and `timed_wait` functions to allow use of `xtime` for timeouts.

Version 2.0.0 - Changes since boost 1.34

Almost every line of code in **Boost.Thread** has been changed since the 1.34 release of boost. However, most of the interface changes have been extensions, so the new code is largely backwards-compatible with the old code. The new features and breaking changes are described below.

New Features

- Instances of `boost::thread` and of the various lock types are now movable.
- Threads can be interrupted at *interruption points*.
- Condition variables can now be used with any type that implements the [Lockable concept](#), through the use of `boost::condition_variable_any` (`boost::condition` is a typedef to `boost::condition_variable_any`, provided for backwards compatibility). `boost::condition_variable` is provided as an optimization, and will only work with `boost::unique_lock<boost::mutex>` (`boost::mutex::scoped_lock`).
- Thread IDs are separated from `boost::thread`, so a thread can obtain it's own ID (using `boost::this_thread::get_id()`), and IDs can be used as keys in associative containers, as they have the full set of comparison operators.
- Timeouts are now implemented using the Boost DateTime library, through a typedef `boost::system_time` for absolute timeouts, and with support for relative timeouts in many cases. `boost::xtime` is supported for backwards compatibility only.
- Locks are implemented as publicly accessible templates `boost::lock_guard`, `boost::unique_lock`, `boost::shared_lock`, and `boost::upgrade_lock`, which are templated on the type of the mutex. The [Lockable concept](#) has been extended to include publicly available `lock()` and `unlock()` member functions, which are used by the lock types.

Breaking Changes

The list below should cover all changes to the public interface which break backwards compatibility.

- `boost::try_mutex` has been removed, and the functionality subsumed into `boost::mutex`. `boost::try_mutex` is left as a typedef, but is no longer a separate class.
- `boost::recursive_try_mutex` has been removed, and the functionality subsumed into `boost::recursive_mutex`. `boost::recursive_try_mutex` is left as a typedef, but is no longer a separate class.
- `boost::detail::thread::lock_ops` has been removed. Code that relies on the `lock_ops` implementation detail will no longer work, as this has been removed, as it is no longer necessary now that mutex types now have public `lock()` and `unlock()` member functions.
- `scoped_lock` constructors with a second parameter of type `bool` are no longer provided. With previous boost releases,

```
boost::mutex::scoped_lock some_lock(some_mutex, false);
```

could be used to create a lock object that was associated with a mutex, but did not lock it on construction. This facility has now been replaced with the constructor that takes a `boost::defer_lock_type` as the second parameter:

```
boost::mutex::scoped_lock some_lock(some_mutex, boost::defer_lock);
```

- The `locked()` member function of the `scoped_lock` types has been renamed to `owns_lock()`.
- You can no longer obtain a `boost::thread` instance representing the current thread: a default-constructed `boost::thread` object is not associated with any thread. The only use for such a thread object was to support the comparison operators: this functionality has been moved to `boost::thread::id`.
- The broken `boost::read_write_mutex` has been replaced with `boost::shared_mutex`.
- `boost::mutex` is now never recursive. For Boost releases prior to 1.35 `boost::mutex` was recursive on Windows and not on POSIX platforms.
- When using a `boost::recursive_mutex` with a call to `boost::condition_variable_any::wait()`, the mutex is only unlocked one level, and not completely. This prior behaviour was not guaranteed and did not feature in the tests.

Future

The following features will be included in next releases.

1. Complete the C++11 missing features, in particular

- [#6227](#) C++11 compliance: Use of variadic templates on Generic Locking Algorithms on compilers providing them.

2. Add some minor features, in particular

- [#7589](#) Synchro: Add polymorphic lockables.

3. Add some features based on C++ proposals, in particular

- [#8273](#) Synchro: Add externally locked streams.
- [#8514](#) Async: Add a `thread_pool` executor with work stealing.

4. Add some of the extension proposed in [A Standardized Representation of Asynchronous Operations](#) or extension to them, in particular

- [#7446](#) Async: Add `when_any`.
- [#7447](#) Async: Add `when_all`.
- [#7448](#) Async: Add `async` taking a scheduler parameter.
- [#8516](#) Async: Add `future/shared_future::then` taking a scheduler as parameter.

5. And some additional extensions related to futures as:

- [#8517](#) Async: Add a variadic `shared_future::then`.

Thread Management

Synopsis

```
#include <boost/thread/thread.hpp>

namespace boost
{
    class thread;
    void swap(thread& lhs, thread& rhs) noexcept;

    namespace this_thread
    {
        thread::id get_id() noexcept;
        template<typename TimeDuration>
        void yield() noexcept; // DEPRECATED
        template <class Clock, class Duration>
        void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
        template <class Rep, class Period>
        void sleep_for(const chrono::duration<Rep, Period>& rel_time);

        template<typename Callable>
        void at_thread_exit(Callable func); // EXTENSION

        void interruption_point(); // EXTENSION
        bool interruption_requested() noexcept; // EXTENSION
        bool interruption_enabled() noexcept; // EXTENSION
        class disable_interruption; // EXTENSION
        class restore_interruption; // EXTENSION

        #if defined BOOST_THREAD_USES_DATETIME
            template <TimeDuration>
            void sleep(TimeDuration const& rel_time); // DEPRECATED
            void sleep(system_time const& abs_time); // DEPRECATED
        #endif
    }
    class thread_group; // EXTENSION
}
```

Tutorial

The `boost::thread` class is responsible for launching and managing threads. Each `boost::thread` object represents a single thread of execution, or *Not-a-Thread*, and at most one `boost::thread` object represents a given thread of execution: objects of type `boost::thread` are not copyable.

Objects of type `boost::thread` are movable, however, so they can be stored in move-aware containers, and returned from functions. This allows the details of thread creation to be wrapped in a function.

```
boost::thread make_thread();

void f()
{
    boost::thread some_thread=make_thread();
    some_thread.join();
}
```



Note

On compilers that support rvalue references, `boost::thread` provides a proper move constructor and move-assignment operator, and therefore meets the C++0x *MoveConstructible* and *MoveAssignable* concepts. With such compilers, `boost::thread` can therefore be used with containers that support those concepts.

For other compilers, move support is provided with a move emulation layer, so containers must explicitly detect that move emulation layer. See `<boost/thread/detail/move.hpp>` for details.

Launching threads

A new thread is launched by passing an object of a callable type that can be invoked with no parameters to the constructor. The object is then copied into internal storage, and invoked on the newly-created thread of execution. If the object must not (or cannot) be copied, then `boost::ref` can be used to pass in a reference to the function object. In this case, the user of **Boost.Thread** must ensure that the referred-to object outlives the newly-created thread of execution.

```
struct callable
{
    void operator()();
};

boost::thread copies_are_safe()
{
    callable x;
    return boost::thread(x);
} // x is destroyed, but the newly-created thread has a copy, so this is OK

boost::thread oops()
{
    callable x;
    return boost::thread(boost::ref(x));
} // x is destroyed, but the newly-created thread still has a reference
// this leads to undefined behaviour
```

If you wish to construct an instance of `boost::thread` with a function or callable object that requires arguments to be supplied, this can be done by passing additional arguments to the `boost::thread` constructor:

```
void find_the_question(int the_answer);

boost::thread deep_thought_2(find_the_question, 42);
```

The arguments are *copied* into the internal thread structure: if a reference is required, use `boost::ref`, just as for references to callable functions.

There is an unspecified limit on the number of additional arguments that can be passed.

Thread attributes

Thread launched in this way are created with implementation defined thread attributes as stack size, scheduling, priority, ... or any platform specific attributes. It is not evident how to provide a portable interface that allows the user to set the platform specific attributes. `Boost.Thread` stay in the middle road through the class `thread::attributes` which allows to set at least in a portable way the stack size as follows:

```
boost::thread::attributes attrs;
attrs.set_size(4096*10);
boost::thread deep_thought_2(attrs, find_the_question, 42);
```

Even for this simple attribute there could be portable issues as some platforms could require that the stack size should have a minimal size and/or be a multiple of a given page size. The library adapts the requested size to the platform constraints so that the user doesn't need to take care of it.

This is the single attribute that is provided in a portable way. In order to set any other thread attribute at construction time the user needs to use non portable code.

On PThread platforms the user will need to get the thread attributes handle and use it for whatever attribute.

Next follows how the user could set the stack size and the scheduling policy on PThread platforms.

```
boost::thread::attributes attrs;
// set portable attributes
// ...
attr.set_stack_size(4096*10);
#if defined(BOOST_THREAD_PLATFORM_WIN32)
    // ... window version
#elif defined(BOOST_THREAD_PLATFORM_PTHREAD)
    // ... pthread version
    pthread_attr_setschedpolicy(attr.get_native_handle(), SCHED_RR);
#else
#error "Boost threads unavailable on this platform"
#endif
boost::thread th(attrs, find_the_question, 42);
```

On Windows platforms it is not so simple as there is no type that compiles the thread attributes. There is a linked to the creation of a thread on Windows that is emulated via the `thread::attributes` class. This is the `LPSECURITY_ATTRIBUTES` `lpThreadAttributes`. `Boost.Thread` provides a non portable `set_security` function so that the user can provide it before the thread creation as follows

```
#if defined(BOOST_THREAD_PLATFORM_WIN32)
    boost::thread::attributes attrs;
    // set portable attributes
    attr.set_stack_size(4096*10);
    // set non portable attribute
    LPSECURITY_ATTRIBUTES sec;
    // init sec
    attr.set_security(sec);
    boost::thread th(attrs, find_the_question, 42);
    // Set other thread attributes using the native_handle_type.
    //...
#else
#error "Platform not supported"
#endif
```

Exceptions in thread functions

If the function or callable object passed to the `boost::thread` constructor propagates an exception when invoked that is not of type `boost::thread_interrupted`, `std::terminate()` is called.

Detaching thread

A thread can be detached by explicitly invoking the `detach()` member function on the `boost::thread` object. In this case, the `boost::thread` object ceases to represent the now-detached thread, and instead represents *Not-a-Thread*.

```
int main()
{
    boost::thread t(my_func);
    t.detach();
}
```

Joining a thread

In order to wait for a thread of execution to finish, the `join()`, `__join_for` or `__join_until` (`timed_join()` deprecated) member functions of the `boost::thread` object must be used. `join()` will block the calling thread until the thread represented by the `boost::thread` object has completed.

```
int main()
{
    boost::thread t(my_func);
    t.join();
}
```

If the thread of execution represented by the `boost::thread` object has already completed, or the `boost::thread` object represents *Not-a-Thread*, then `join()` returns immediately.

```
int main()
{
    boost::thread t;
    t.join(); // do nothing
}
```

Timed based join are similar, except that a call to `__join_for` or `__join_until` will also return if the thread being waited for does not complete when the specified time has elapsed or reached respectively.

```
int main()
{
    boost::thread t;
    if ( t.join_for(boost::chrono::milliseconds(500)) )
        // do something else
    t.join(); // join anyway
}
```

Destructor V1

When the `boost::thread` object that represents a thread of execution is destroyed the thread becomes *detached*. Once a thread is detached, it will continue executing until the invocation of the function or callable object supplied on construction has completed, or the program is terminated. A thread can also be detached by explicitly invoking the `detach()` member function on the `boost::thread` object. In this case, the `boost::thread` object ceases to represent the now-detached thread, and instead represents *Not-a-Thread*.

Destructor V2

When the `boost::thread` object that represents a thread of execution is destroyed the program terminates if the thread is `__joinable__`.

```
int main()
{
    boost::thread t(my_func);
} // calls std::terminate()
```

You can use a `thread_joiner` to ensure that the thread has been joined at the thread destructor.

```
int main()
{
    boost::thread t(my_func);
    boost::thread_joiner g(t);
    // do something else
} // here the thread_joiner destructor will join the thread before it is destroyed.
```

Interruption

A running thread can be *interrupted* by invoking the `interrupt()` member function of the corresponding `boost::thread` object. When the interrupted thread next executes one of the specified *interruption points* (or if it is currently *blocked* whilst executing one) with interruption enabled, then a `boost::thread_interrupted` exception will be thrown in the interrupted thread. If not caught, this will cause the execution of the interrupted thread to terminate. As with any other exception, the stack will be unwound, and destructors for objects of automatic storage duration will be executed.

If a thread wishes to avoid being interrupted, it can create an instance of `boost::this_thread::disable_interruption`. Objects of this class disable interruption for the thread that created them on construction, and restore the interruption state to whatever it was before on destruction:

```
void f()
{
    // interruption enabled here
    {
        boost::this_thread::disable_interruption di;
        // interruption disabled
        {
            boost::this_thread::disable_interruption di2;
            // interruption still disabled
        } // di2 destroyed, interruption state restored
        // interruption still disabled
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

The effects of an instance of `boost::this_thread::disable_interruption` can be temporarily reversed by constructing an instance of `boost::this_thread::restore_interruption`, passing in the `boost::this_thread::disable_interruption` object in question. This will restore the interruption state to what it was when the `boost::this_thread::disable_interruption` object was constructed, and then disable interruption again when the `boost::this_thread::restore_interruption` object is destroyed.

```
void g()
{
    // interruption enabled here
    {
        boost::this_thread::disable_interruption di;
        // interruption disabled
        {
            boost::this_thread::restore_interruption ri(di);
            // interruption now enabled
        } // ri destroyed, interruption disable again
    } // di destroyed, interruption state restored
    // interruption now enabled
}
```

At any point, the interruption state for the current thread can be queried by calling `boost::this_thread::interruption_enabled()`.

Predefined Interruption Points

The following functions are *interruption points*, which will throw `boost::thread_interrupted` if interruption is enabled for the current thread, and interruption is requested for the current thread:

- `boost::thread::join()`
- `boost::thread::timed_join()`
- `boost::thread::try_join_for()`,
- `boost::thread::try_join_until()`,
- `boost::condition_variable::wait()`
- `boost::condition_variable::timed_wait()`
- `boost::condition_variable::wait_for()`
- `boost::condition_variable::wait_until()`
- `boost::condition_variable_any::wait()`
- `boost::condition_variable_any::timed_wait()`
- `boost::condition_variable_any::wait_for()`
- `boost::condition_variable_any::wait_until()`
- `boost::thread::sleep()`
- `boost::this_thread::sleep_for()`
- `boost::this_thread::sleep_until()`
- `boost::this_thread::interruption_point()`

Thread IDs

Objects of class `boost::thread::id` can be used to identify threads. Each running thread of execution has a unique ID obtainable from the corresponding `boost::thread` by calling the `get_id()` member function, or by calling `boost::this_thread::get_id()` from within the thread. Objects of class `boost::thread::id` can be copied, and used as keys in associative containers: the full range of comparison operators is provided. Thread IDs can also be written to an output stream using the stream insertion operator, though the output format is unspecified.

Each instance of `boost::thread::id` either refers to some thread, or *Not-a-Thread*. Instances that refer to *Not-a-Thread* compare equal to each other, but not equal to any instances that refer to an actual thread of execution. The comparison operators on `boost::thread::id` yield a total order for every non-equal thread ID.

Using native interfaces with Boost.Thread resources

`boost::thread` class has members `native_handle_type` and `native_handle` providing access to the underlying native handle.

This native handle can be used to change for example the scheduling.

In general, it is not safe to use this handle with operations that can conflict with the ones provided by Boost.Thread. An example of bad usage could be detaching a thread directly as it will not change the internals of the `boost::thread` instance, so for example the joinable function will continue to return true, while the native thread is no more joinable.

```
thread t(fct);
thread::native_handle_type hnd=t.native_handle();
pthread_detach(hnd);
assert(t.joinable());
```

Using Boost.Thread interfaces in a native thread

Any thread of execution created using the native interface is called a native thread in this documentation.

The first example of a native thread of execution is the main thread.

The user can access to some synchronization functions related to the native current thread using the `boost::this_thread::yield`, `sleep`, `sleep_for`, `sleep_until`, functions.

```
int main() {
    // ...
    boost::this_thread::sleep_for(boost::chrono::milliseconds(10));
    // ...
}
```

Of course all the synchronization facilities provided by Boost.Thread are also available on native threads.

The `boost::this_thread::interrupt` related functions behave in a degraded mode when called from a thread created using the native interface, i.e. `boost::this_thread::interrupt_enabled()` returns false. As consequence the use of `boost::this_thread::disable_interruption` and `boost::this_thread::restore_interruption` will do nothing and calls to `boost::this_thread::interruption_point()` will be just ignored.

As the single way to interrupt a thread is through a `boost::thread` instance, `interruption_request()` will returns false for the native threads.

`pthread_exit` POSIX limitation

`pthread_exit` in glibc/NPTL causes a "forced unwind" that is almost like a C++ exception, but not quite. On Mac OS X, for example, `pthread_exit` unwinds without calling C++ destructors.

This behavior is incompatible with the current Boost.Thread design, so the use of this function in a POSIX thread result in undefined behavior of any Boost.Thread function.

Class `thread`

```
#include <boost/thread/thread.hpp>

class thread
{
public:
    class attributes; // EXTENSION

    thread() noexcept;
    thread(const thread&) = delete;
    thread& operator=(const thread&) = delete;

    thread(thread&&) noexcept;
    thread& operator=(thread&&) noexcept;
    ~thread();

    template <class F>
    explicit thread(F f);
    template <class F>
    thread(F &&f);

    template <class F, class A1, class A2, ...>
    thread(F f, A1 a1, A2 a2, ...);
    template <class F, class ...Args>
    explicit thread(F&& f, Args&&... args);

    template <class F>
    explicit thread(attributes& attrs, F f); // EXTENSION
    template <class F>
    thread(attributes& attrs, F &&f); // EXTENSION
    template <class F, class ...Args>
    explicit thread(attributes& attrs, F&& f, Args&&... args);

    // move support
    thread(thread && x) noexcept;
    thread& operator=(thread && x) noexcept;

    void swap(thread& x) noexcept;

    class id;

    id get_id() const noexcept;

    bool joinable() const noexcept;
    void join();
    template <class Rep, class Period>
    bool try_join_for(const chrono::duration<Rep, Period>& rel_time); // EXTENSION
    template <class Clock, class Duration>
    bool try_join_until(const chrono::time_point<Clock, Duration>& t); // EXTENSION

    void detach();

    static unsigned hardware_concurrency() noexcept;
    static unsigned physical_concurrency() noexcept;

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    void interrupt(); // EXTENSION
    bool interruption_requested() const noexcept; // EXTENSION
```

```

#if defined BOOST_THREAD_USES_DATETIME
    bool timed_join(const system_time& wait_until); // DEPRECATED
    template<typename TimeDuration>
    bool timed_join(TimeDuration const& rel_time); // DEPRECATED
    static void sleep(const system_time& xt); // DEPRECATED
#endif

#if defined BOOST_THREAD_PROVIDES_THREAD_EQ
    bool operator==(const thread& other) const; // DEPRECATED
    bool operator!=(const thread& other) const; // DEPRECATED
#endif

    static void yield() noexcept; // DEPRECATED
};

void swap(thread& lhs, thread& rhs) noexcept;

```

Default Constructor

```
thread() noexcept;
```

Effects: Constructs a `boost::thread` instance that refers to *Not-a-Thread*.

Postconditions: `this->get_id()==thread::id()`

Throws: Nothing

Move Constructor

```
thread(thread&& other) noexcept;
```

Effects: Transfers ownership of the thread managed by `other` (if any) to the newly constructed `boost::thread` instance.

Postconditions: `other.get_id()==thread::id()` and `get_id()` returns the value of `other.get_id()` prior to the construction

Throws: Nothing

Move assignment operator

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

Effects: Transfers ownership of the thread managed by `other` (if any) to `*this`.

- if defined `BOOST_THREAD_DONT_PROVIDE_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE`: If the thread is joinable call `detach()`, DEPRECATED

- if defined `BOOST_THREAD_PROVIDES_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE`: If the thread is joinable calls to `std::terminate()`.

Postconditions: `other->get_id()==thread::id()` and `get_id()` returns the value of `other.get_id()` prior to the assignment.

Throws: Nothing

Thread Constructor

```
template<typename Callable>
thread(Callable func);
```

Requires:	Callable must be Copyable and func() must be a valid expression.
Effects:	func is copied into storage managed internally by the thread library, and that copy is invoked on a newly-created thread of execution. If this invocation results in an exception being propagated into the internals of the thread library that is not of type boost::thread_interrupted, then std::terminate() will be called. Any return value from this invocation is ignored.
Postconditions:	*this refers to the newly created thread of execution and this->get_id()!=thread::id().
Throws:	boost::thread_resource_error if an error occurs.
Error Conditions:	resource_unavailable_try_again : the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

Thread Attributes Constructor EXTENSION

```
template<typename Callable>
thread(attributes& attrs, Callable func);
```

Preconditions:	Callable must be copyable.
Effects:	func is copied into storage managed internally by the thread library, and that copy is invoked on a newly-created thread of execution with the specified attributes. If this invocation results in an exception being propagated into the internals of the thread library that is not of type boost::thread_interrupted, then std::terminate() will be called. Any return value from this invocation is ignored. If the attributes declare the native thread as detached, the boost::thread will be detached.
Postconditions:	*this refers to the newly created thread of execution and this->get_id()!=thread::id().
Throws:	boost::thread_resource_error if an error occurs.
Error Conditions:	resource_unavailable_try_again : the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

Thread Callable Move Constructor

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

Preconditions:	Callable must be Movable.
Effects:	func is moved into storage managed internally by the thread library, and that copy is invoked on a newly-created thread of execution. If this invocation results in an exception being propagated into the internals of the thread library that is not of type boost::thread_interrupted, then std::terminate() will be called. Any return value from this invocation is ignored.
Postconditions:	*this refers to the newly created thread of execution and this->get_id()!=thread::id().
Throws:	boost::thread_resource_error if an error occurs.
Error Conditions:	resource_unavailable_try_again : the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

Thread Attributes Move Constructor EXTENSION

```
template<typename Callable>
thread(attributes& attrs, Callable func);
```

Preconditions:	Callable must be copyable.
Effects:	func is copied into storage managed internally by the thread library, and that copy is invoked on a newly-created thread of execution with the specified attributes. If this invocation results in an exception being propagated into the internals of the thread library that is not of type <code>boost::thread_interrupted</code> , then <code>std::terminate()</code> will be called. Any return value from this invocation is ignored. If the attributes declare the native thread as detached, the <code>boost::thread</code> will be detached.
Postconditions:	*this refers to the newly created thread of execution and <code>this->get_id() != thread::id()</code> .
Throws:	<code>boost::thread_resource_error</code> if an error occurs.
Error Conditions:	resource_unavailable_try_again : the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

Thread Constructor with arguments

```
template <class F, class A1, class A2, ...>
thread(F f, A1 a1, A2 a2, ...);
```

Preconditions:	F and each An must be copyable or movable.
Effects:	As if <code>thread(boost::bind(f, a1, a2, ...))</code> . Consequently, f and each an are copied into internal storage for access by the new thread.
Postconditions:	*this refers to the newly created thread of execution.
Throws:	<code>boost::thread_resource_error</code> if an error occurs.
Error Conditions:	resource_unavailable_try_again : the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.
Note:	Currently up to nine additional arguments a1 to a9 can be specified in addition to the function f.

Thread Destructor

```
~thread();
```

Effects:	<ul style="list-style-type: none"> - if defined <code>BOOST_THREAD_DONT_PROVIDE_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE</code>: If the thread is joinable calls <code>detach()</code>, DEPRECATED - if defined <code>BOOST_THREAD_PROVIDES_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE</code>: If the thread is joinable calls to <code>std::terminate</code>. Destroys *this.
Throws:	Nothing.
Note:	The reason to moving to <code>std::terminate</code> is that either implicitly detaching or joining a <code>joinable()</code> thread in its destructor could result in difficult to debug correctness (for <code>detach</code>) or performance (for <code>join</code>) bugs encountered only when an exception is raised. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. Join the thread before destroying or use an scoped thread.

Member function `joinable()`

```
bool joinable() const noexcept;
```

Returns: true if `*this` refers to a thread of execution, false otherwise.

Throws: Nothing

Member function `join()`

```
void join();
```

Preconditions: the thread is joinable.

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete.

Synchronization: The completion of the thread represented by `*this` synchronizes with the corresponding successful `join()` return.

Note: Operations on `*this` are not synchronized.

Postconditions: If `*this` refers to a thread of execution on entry, that thread of execution has completed. `*this` no longer refers to any thread of execution.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted or `system_error`

Error Conditions: **resource_deadlock_would_occur:** if deadlock is detected or `this->get_id() == boost::this_thread::get_id()`.

invalid_argument: if the thread is not joinable and `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined.

Notes: `join()` is one of the predefined *interruption points*.

Member function `timed_join()` DEPRECATED

```
bool timed_join(const system_time& wait_until);

template<typename TimeDuration>
bool timed_join(TimeDuration const& rel_time);
```



Warning

DEPRECATED since 3.00.

Use instead `try_join_for`, `try_join_until`.

Preconditions: the thread is joinable.

Effects: If `*this` refers to a thread of execution, waits for that thread of execution to complete, the time `wait_until` has been reach or the specified duration `rel_time` has elapsed. If `*this` doesn't refer to a thread of execution, returns immediately.

Returns: true if `*this` refers to a thread of execution on entry, and that thread of execution has completed before the call times out, false otherwise.

Postconditions:	If <code>*this</code> refers to a thread of execution on entry, and <code>timed_join</code> returns <code>true</code> , that thread of execution has completed, and <code>*this</code> no longer refers to any thread of execution. If this call to <code>timed_join</code> returns <code>false</code> , <code>*this</code> is unchanged.
Throws:	<code>boost::thread_interrupted</code> if the current thread of execution is interrupted or <code>system_error</code>
Error Conditions:	resource_deadlock_would_occur: if deadlock is detected or <code>this->get_id() == boost::this_thread::get_id()</code> . invalid_argument: if the thread is not joinable and <code>BOOST_THREAD_TRHOW_IF_PRECONDITION_NOT_SATISFIED</code> is defined.
Notes:	<code>timed_join()</code> is one of the predefined <i>interruption points</i> .

Member function `try_join_for()` EXTENSION

```
template <class Rep, class Period>
bool try_join_for(const chrono::duration<Rep, Period>& rel_time);
```

Preconditions:	the thread is joinable.
Effects:	If <code>*this</code> refers to a thread of execution, waits for that thread of execution to complete, the specified duration <code>rel_time</code> has elapsed. If <code>*this</code> doesn't refer to a thread of execution, returns immediately.
Returns:	<code>true</code> if <code>*this</code> refers to a thread of execution on entry, and that thread of execution has completed before the call times out, <code>false</code> otherwise.
Postconditions:	If <code>*this</code> refers to a thread of execution on entry, and <code>try_join_for</code> returns <code>true</code> , that thread of execution has completed, and <code>*this</code> no longer refers to any thread of execution. If this call to <code>try_join_for</code> returns <code>false</code> , <code>*this</code> is unchanged.
Throws:	<code>boost::thread_interrupted</code> if the current thread of execution is interrupted or <code>system_error</code>
Error Conditions:	resource_deadlock_would_occur: if deadlock is detected or <code>this->get_id() == boost::this_thread::get_id()</code> . invalid_argument: if the thread is not joinable and <code>BOOST_THREAD_TRHOW_IF_PRECONDITION_NOT_SATISFIED</code> is defined.
Notes:	<code>try_join_for()</code> is one of the predefined <i>interruption points</i> .

Member function `try_join_until()` EXTENSION

```
template <class Clock, class Duration>
bool try_join_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Preconditions:	the thread is joinable.
Effects:	If <code>*this</code> refers to a thread of execution, waits for that thread of execution to complete, the time <code>abs_time</code> has been reach. If <code>*this</code> doesn't refer to a thread of execution, returns immediately.
Returns:	<code>true</code> if <code>*this</code> refers to a thread of execution on entry, and that thread of execution has completed before the call times out, <code>false</code> otherwise.
Postconditions:	If <code>*this</code> refers to a thread of execution on entry, and <code>try_join_until</code> returns <code>true</code> , that thread of execution has completed, and <code>*this</code> no longer refers to any thread of execution. If this call to <code>try_join_until</code> returns <code>false</code> , <code>*this</code> is unchanged.
Throws:	<code>boost::thread_interrupted</code> if the current thread of execution is interrupted or <code>system_error</code>
Error Conditions:	resource_deadlock_would_occur: if deadlock is detected or <code>this->get_id() == boost::this_thread::get_id()</code> .

invalid_argument: if the thread is not joinable and `BOOST_THREAD_TRHOW_IF_PRECONDITION_NOT_SATISFIED` is defined.

Notes: `try_join_until()` is one of the predefined *interruption points*.

Member function `detach()`

```
void detach();
```

Preconditions: the thread is joinable.

Effects: The thread of execution becomes detached, and no longer has an associated `boost::thread` object.

Postconditions: `*this` no longer refers to any thread of execution.

Throws: `system_error`

Error Conditions: **no_such_process:** if the thread is not valid.

invalid_argument: if the thread is not joinable and `BOOST_THREAD_TRHOW_IF_PRECONDITION_NOT_SATISFIED` is defined.

Member function `get_id()`

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

Returns: If `*this` refers to a thread of execution, an instance of `boost::thread::id` that represents that thread. Otherwise returns a default-constructed `boost::thread::id`.

Throws: Nothing

Member function `interrupt()` EXTENSION

```
void interrupt();
```

Effects: If `*this` refers to a thread of execution, request that the thread will be interrupted the next time it enters one of the predefined *interruption points* with interruption enabled, or if it is currently *blocked* in a call to one of the predefined *interruption points* with interruption enabled.

Throws: Nothing

Static member function `hardware_concurrency()`

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

Returns: The number of hardware threads available on the current system (e.g. number of CPUs or cores or hyperthreading units), or 0 if this information is not available.

Throws: Nothing

Static member function `physical_concurrency()`

```
unsigned physical_concurrency() noexcept;
```

Returns: The number of physical cores available on the current system. In contrast to `hardware_concurrency()` it does not return the number of virtual cores, but it counts only physical cores.

Throws: Nothing

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

operator== **DEPRECATED**

```
bool operator==(const thread& other) const;
```

**Warning**

DEPRECATED since 4.0.0.

Use `a.get_id()==b.get_id()` instead.

Returns: `get_id()==other.get_id()`

operator!= **DEPRECATED**

```
bool operator!=(const thread& other) const;
```

**Warning**

DEPRECATED since 4.0.0.

Use `a.get_id()!=b.get_id()` instead.

Returns: `get_id()!=other.get_id()`

Static member function `sleep()` **DEPRECATED**

```
void sleep(system_time const& abs_time);
```



Warning

DEPRECATED since 3.0.0.

Use `this_thread::sleep_for()` or `this_thread::sleep_until()`.

Effects: Suspends the current thread until the specified time has been reached.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `sleep()` is one of the predefined *interruption points*.

Static member function `yield()` DEPRECATED

```
void yield();
```



Warning

DEPRECATED since 3.0.0.

Use `this_thread::yield()`.

Effects: See `boost::this_thread::yield()`.

Member function `swap()`

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

Effects: Exchanges the threads of execution associated with `*this` and `other`, so `*this` is associated with the thread of execution associated with `other` prior to the call, and vice-versa.

Postconditions: `this->get_id()` returns the same value as `other.get_id()` prior to the call. `other.get_id()` returns the same value as `this->get_id()` prior to the call.

Throws: Nothing.

Non-member function `swap()`

```
#include <boost/thread/thread.hpp>

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

Effects: `lhs.swap(rhs)`.

Class `boost::thread::id`

```
#include <boost/thread/thread.hpp>

class thread::id
{
public:
    id() noexcept;

    bool operator==(const id& y) const noexcept;
    bool operator!=(const id& y) const noexcept;
    bool operator<(const id& y) const noexcept;
    bool operator>(const id& y) const noexcept;
    bool operator<=(const id& y) const noexcept;
    bool operator>=(const id& y) const noexcept;

    template<class charT, class traits>
    friend std::basic_ostream<charT, traits>&
    operator<<(std::basic_ostream<charT, traits>& os, const id& x);
};
```

Default constructor

```
id() noexcept;
```

Effects: Constructs a `boost::thread::id` instance that represents *Not-a-Thread*.

Throws: Nothing

`operator==`

```
bool operator==(const id& y) const noexcept;
```

Returns: true if `*this` and `y` both represent the same thread of execution, or both represent *Not-a-Thread*, false otherwise.

Throws: Nothing

`operator!=`

```
bool operator!=(const id& y) const noexcept;
```

Returns: true if `*this` and `y` represent different threads of execution, or one represents a thread of execution, and the other represent *Not-a-Thread*, false otherwise.

Throws: Nothing

`operator<`

```
bool operator<(const id& y) const noexcept;
```

Returns: true if `*this!=y` is true and the implementation-defined total order of `boost::thread::id` values places `*this` before `y`, false otherwise.

Throws: Nothing

Note: A `boost::thread::id` instance representing *Not-a-Thread* will always compare less than an instance representing a thread of execution.

operator>

```
bool operator>(const id& y) const noexcept;
```

Returns: `y<*this`

Throws: Nothing

operator<=

```
bool operator<=(const id& y) const noexcept;
```

Returns: `!(y<*this)`

Throws: Nothing

operator>=

```
bool operator>=(const id& y) const noexcept;
```

Returns: `!(*this<y)`

Throws: Nothing

Friend operator<<

```
template<class charT, class traits>
friend std::basic_ostream<charT, traits>&
operator<<(std::basic_ostream<charT, traits>& os, const id& x);
```

Effects: Writes a representation of the `boost::thread::id` instance `x` to the stream `os`, such that the representation of two instances of `boost::thread::id` `a` and `b` is the same if `a==b`, and different if `a!=b`.

Returns: `os`

Class `boost::thread::attributes` EXTENSION

```
class thread::attributes {
public:
    attributes() noexcept;
    ~ attributes()=default;
    // stack
    void set_stack_size(std::size_t size) noexcept;
    std::size_t get_stack_size() const noexcept;

#if defined BOOST_THREAD_DEFINES_THREAD_ATTRIBUTES_NATIVE_HANDLE
    typedef platform-specific-type native_handle_type;
    native_handle_type* native_handle() noexcept;
    const native_handle_type* native_handle() const noexcept;
#endif
};
```

Default constructor

```
thread_attributes() noexcept;
```

Effects: Constructs a thread attributes instance with its default values.

Throws: Nothing

Member function `set_stack_size()`

```
void set_stack_size(std::size_t size) noexcept;
```

Effects: Stores the stack size to be used to create a thread. This is an hint that the implementation can choose a better size if too small or too big or not aligned to a page.

Postconditions: `this-> get_stack_size()` returns the chosen stack size.

Throws: Nothing.

Member function `get_stack_size()`

```
std::size_t get_stack_size() const noexcept;
```

Returns: The stack size to be used on the creation of a thread. Note that this function can return 0 meaning the default.

Throws: Nothing.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;  
typedef platform-specific-type native_handle_type;  
native_handle_type* native_handle() noexcept;  
const native_handle_type* native_handle() const noexcept;
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying thread attributes implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present and `BOOST_THREAD_DEFINES_THREAD_ATTRIBUTES_NATIVE_HANDLE` is not defined.

Throws: Nothing.

Namespace `this_thread`

```
namespace boost {
  namespace this_thread {
    thread::id get_id() noexcept;
    template<typename TimeDuration>
    void yield() noexcept;
    template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);

    template<typename Callable>
    void at_thread_exit(Callable func); // EXTENSION

    void interruption_point(); // EXTENSION
    bool interruption_requested() noexcept; // EXTENSION
    bool interruption_enabled() noexcept; // EXTENSION
    class disable_interruption; // EXTENSION
    class restore_interruption; // EXTENSION

    #if defined BOOST_THREAD_USES_DATETIME
    void sleep(TimeDuration const& rel_time); // DEPRECATED
    void sleep(system_time const& abs_time); // DEPRECATED
    #endif
  }
}
```

Non-member function `get_id()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
  thread::id get_id() noexcept;
}
```

Returns: An instance of `boost::thread::id` that represents that currently executing thread.

Throws: `boost::thread_resource_error` if an error occurs.

Non-member function `interruption_point()` **EXTENSION**

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
  void interruption_point();
}
```

Effects: Check to see if the current thread has been interrupted.

Throws: `boost::thread_interrupted` if `boost::this_thread::interruption_enabled()` and `boost::this_thread::interruption_requested()` both return true.

Non-member function `interruption_requested()` EXTENSION

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    bool interruption_requested() noexcept;
}
```

Returns: true if interruption has been requested for the current thread, false otherwise.

Throws: Nothing.

Non-member function `interruption_enabled()` EXTENSION

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    bool interruption_enabled() noexcept;
}
```

Returns: true if interruption has been enabled for the current thread, false otherwise.

Throws: Nothing.

Non-member function `sleep()` DEPRECATED

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    template<typename TimeDuration>
    void sleep(TimeDuration const& rel_time);
    void sleep(system_time const& abs_time)
}
```



Warning

DEPRECATED since 3.0.0.

Use `sleep_for()` and `sleep_until()` instead.

Effects: Suspends the current thread until the time period specified by `rel_time` has elapsed or the time point specified by `abs_time` has been reached.

Throws: `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `sleep()` is one of the predefined *interruption points*.

Non-member function `sleep_until()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
}
```

Effects: Suspends the current thread until the time point specified by `abs_time` has been reached.

Throws: Nothing if `Clock` satisfies the `TrivialClock` requirements and operations of `Duration` do not throw exceptions. `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `sleep_until()` is one of the predefined *interruption points*.

Non-member function `sleep_for()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
}
```

Effects: Suspends the current thread until the duration specified by `rel_time` has elapsed.

Throws: Nothing if operations of `chrono::duration<Rep, Period>` do not throw exceptions. `boost::thread_interrupted` if the current thread of execution is interrupted.

Notes: `sleep_for()` is one of the predefined *interruption points*.

Non-member function `yield()`

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    void yield() noexcept;
}
```

Effects: Gives up the remainder of the current thread's time slice, to allow other threads to run.

Throws: Nothing.

Class `disable_interruption` EXTENSION

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    class disable_interruption
    {
    public:
        disable_interruption(const disable_interruption&) = delete;
        disable_interruption& operator=(const disable_interruption&) = delete;
        disable_interruption() noexcept;
        ~disable_interruption() noexcept;
    };
}
```

`boost::this_thread::disable_interruption` disables interruption for the current thread on construction, and restores the prior interruption state on destruction. Instances of `disable_interruption` cannot be copied or moved.

Constructor

```
disable_interruption() noexcept;
```

Effects: Stores the current state of `boost::this_thread::interruption_enabled()` and disables interruption for the current thread.

Postconditions: `boost::this_thread::interruption_enabled()` returns `false` for the current thread.

Throws: Nothing.

Destructor

```
~disable_interruption() noexcept;
```

Preconditions: Must be called from the same thread from which `*this` was constructed.

Effects: Restores the current state of `boost::this_thread::interruption_enabled()` for the current thread to that prior to the construction of `*this`.

Postconditions: `boost::this_thread::interruption_enabled()` for the current thread returns the value stored in the constructor of `*this`.

Throws: Nothing.

Class `restore_interruption` EXTENSION

```
#include <boost/thread/thread.hpp>

namespace this_thread
{
    class restore_interruption
    {
    public:
        restore_interruption(const restore_interruption&) = delete;
        restore_interruption& operator=(const restore_interruption&) = delete;
        explicit restore_interruption(disable_interruption& disabler) noexcept;
        ~restore_interruption() noexcept;
    };
}
```

On construction of an instance of `boost::this_thread::restore_interruption`, the interruption state for the current thread is restored to the interruption state stored by the constructor of the supplied instance of `boost::this_thread::disable_interruption`. When the instance is destroyed, interruption is again disabled. Instances of `restore_interruption` cannot be copied or moved.

Constructor

```
explicit restore_interruption(disable_interruption& disabler) noexcept;
```

- Preconditions:** Must be called from the same thread from which `disabler` was constructed.
- Effects:** Restores the current state of `boost::this_thread::interruption_enabled()` for the current thread to that prior to the construction of `disabler`.
- Postconditions:** `boost::this_thread::interruption_enabled()` for the current thread returns the value stored in the constructor of `disabler`.
- Throws:** Nothing.

Destructor

```
~restore_interruption() noexcept;
```

- Preconditions:** Must be called from the same thread from which `*this` was constructed.
- Effects:** Disables interruption for the current thread.
- Postconditions:** `boost::this_thread::interruption_enabled()` for the current thread returns `false`.
- Throws:** Nothing.

Non-member function template `at_thread_exit()` EXTENSION

```
#include <boost/thread/thread.hpp>

template<typename Callable>
void at_thread_exit(Callable func);
```

- Effects:** A copy of `func` is placed in thread-specific storage. This copy is invoked when the current thread exits (even if the thread has been interrupted).

Postconditions:	A copy of func has been saved for invocation on thread exit.
Throws:	<code>std::bad_alloc</code> if memory cannot be allocated for the copy of the function, <code>boost::thread_resource_error</code> if any other error occurs within the thread library. Any exception thrown whilst copying func into internal storage.
Note:	This function is not called if the thread was terminated forcefully using platform-specific APIs, or if the thread is terminated due to a call to <code>exit()</code> , <code>abort()</code> or <code>std::terminate()</code> . In particular, returning from <code>main()</code> is equivalent to call to <code>exit()</code> , so will not call any functions registered with <code>at_thread_exit()</code>

Class `thread_group` EXTENSION

```
#include <boost/thread/thread.hpp>

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

    thread_group();
    ~thread_group();

    template<typename F>
    thread* create_thread(F threadfunc);
    void add_thread(thread* thrd);
    void remove_thread(thread* thrd);
    bool is_this_thread_in();
    bool is_thread_in(thread* thrd);
    void join_all();
    void interrupt_all();
    int size() const;
};
```

`thread_group` provides for a collection of threads that are related in some fashion. New threads can be added to the group with `add_thread` and `create_thread` member functions. `thread_group` is not copyable or movable.

Constructor

```
thread_group();
```

Effects: Create a new thread group with no threads.

Destructor

```
~thread_group();
```

Effects: Destroy `*this` and delete all `boost::thread` objects in the group.

Member function `create_thread()`

```
template<typename F>
thread* create_thread(F threadfunc);
```

Effects: Create a new `boost::thread` object as-if by `new thread(threadfunc)` and add it to the group.

Postcondition: `this->size()` is increased by one, the new thread is running.

Returns: A pointer to the new `boost::thread` object.

Member function `add_thread()`

```
void add_thread(thread* thrd);
```

Precondition: The expression `delete thrd` is well-formed and will not result in undefined behaviour and `is_thread_in(thrd) == false`.

Effects: Take ownership of the `boost::thread` object pointed to by `thrd` and add it to the group.

Postcondition: `this->size()` is increased by one.

Member function `remove_thread()`

```
void remove_thread(thread* thrd);
```

Effects: If `thrd` is a member of the group, remove it without calling `delete`.

Postcondition: If `thrd` was a member of the group, `this->size()` is decreased by one.

Member function `join_all()`

```
void join_all();
```

Requires: `is_this_thread_in() == false`.

Effects: Call `join()` on each `boost::thread` object in the group.

Postcondition: Every thread in the group has terminated.

Note: Since `join()` is one of the predefined *interruption points*, `join_all()` is also an interruption point.

Member function `is_this_thread_in()`

```
bool is_this_thread_in();
```

Returns: true if there is a thread `th` in the group such that `th.get_id() == this_thread::get_id()`.

Member function `is_thread_in()`

```
bool is_thread_in(thread* thrd);
```

Returns: true if there is a thread `th` in the group such that `th.get_id() == thrd->get_id()`.

Member function `interrupt_all()`

```
void interrupt_all();
```

Effects: Call `interrupt()` on each `boost::thread` object in the group.

Member function `size()`

```
int size();
```

Returns: The number of threads in the group.

Throws: Nothing.

Scoped Threads

Synopsis

```
//#include <boost/thread/scoped_thread.hpp>

struct detach;
struct join_if_joinable;
struct interrupt_and_join_if_joinable;
template <class CallableThread = join_if_joinable>
class strict_scoped_thread;
template <class CallableThread = join_if_joinable>
class scoped_thread;
void swap(scoped_thread& lhs,scoped_thread& rhs) noexcept;
```

Motivation

Based on the `scoped_thread` class defined in C++ Concurrency in Action Boost.Thread defines a thread wrapper class that instead of calling `terminate` if the thread is joinable on destruction, call a specific action given as template parameter.

While the `scoped_thread` class defined in C++ Concurrency in Action is closer to `strict_scoped_thread` class that doesn't allow any change in the wrapped thread, Boost.Thread provides a class `scoped_thread` that provides the same non-deprecated interface than `thread`.

Tutorial

Scoped Threads are wrappers around a thread that allows the user to state what to do at destruction time. One of the common uses is to join the thread at destruction time so this is the default behavior. This is the single difference respect to a thread. While `thread` call `std::terminate()` on the destructor if the thread is joinable, `strict_scoped_thread<>` or `scoped_thread<>` join the thread if joinable.

The difference between `strict_scoped_thread` and `scoped_thread` is that the `strict_scoped_thread` hides completely the owned thread and so the user can do nothing with the owned thread other than the specific action given as parameter, while `scoped_thread` provide the same interface than `thread` and forwards all the operations.

```
boost::strict_scoped_thread<> t1((boost::thread(f)));
//t1.detach(); // compile fails
boost::scoped_thread<> t2((boost::thread(f)));
t2.detach();
```

Free Thread Functors

```
//#include <boost/thread/scoped_thread.hpp>

struct detach;
struct join_if_joinable;
struct interrupt_and_join_if_joinable;
```

Functor detach

```
struct detach
{
    void operator()(thread& t)
    {
        t.detach();
    }
};
```

Functor join_if_joinable

```
struct join_if_joinable
{
    void operator()(thread& t)
    {
        if (t.joinable())
        {
            t.join();
        }
    }
};
```

Functor interrupt_and_join_if_joinable

```
struct interrupt_and_join_if_joinable
{
    void operator()(thread& t)
    {
        t.interrupt();
        if (t.joinable())
        {
            t.join();
        }
    }
};
```

Class strict_scoped_thread

```
// #include <boost/thread/scoped_thread.hpp>

template <class CallableThread = join_if_joinable>
class strict_scoped_thread
{
    thread t_; // for exposition purposes only
public:

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

    explicit strict_scoped_thread(thread&& t) noexcept;
    template <typename F&&, typename ...Args>
    explicit strict_scoped_thread(F&&, Args&&...);

    ~strict_scoped_thread();
};
```


RAI `thread` wrapper adding a specific destroyer allowing to master what can be done at destruction time.

CallableThread: A callable `void(thread&)`.

The default is a `join_if_joinable`.

`std::boost::thread` destructor terminates the program if the `thread` is not joinable. This wrapper can be used to join the thread before destroying it seems a natural need.

Example

```
boost::strict_scoped_thread<> t((boost::thread(F)));
```

Constructor from a `thread`

```
explicit strict_scoped_thread(thread&& t) noexcept;
```

Effects: move the thread to own `t_`

Throws: Nothing

Move Constructor from a Callable

```
template <typename F&&, typename ...Args>
explicit strict_scoped_thread(F&&, Args&&...);
```

Effects: Construct a internal thread in place.

Postconditions: `*this.t_` refers to the newly created thread of execution and `this->get_id() != thread::id()`.

Throws: Any exception the thread construction can throw.

Destructor

```
~strict_scoped_thread();
```

Effects: Equivalent to `CallableThread()(t_)`.

Throws: Nothing: The `CallableThread()(t_)` should not throw when joining the thread as the scoped variable is on a scope outside the thread function.

Class `scoped_thread`

```
#include <boost/thread/scoped_thread.hpp>

template <class CallableThread>
class scoped_thread
{
    thread t_; // for exposition purposes only
public:
    scoped_thread() noexcept;
    scoped_thread(const scoped_thread&) = delete;
    scoped_thread& operator=(const scoped_thread&) = delete;

    explicit scoped_thread(thread&& th) noexcept;
    template <typename F&&, typename ...Args>
    explicit scoped_thread(F&&, Args&&...);

    ~scoped_thread();

    // move support
    scoped_thread(scoped_thread && x) noexcept;
    scoped_thread& operator=(scoped_thread && x) noexcept;

    void swap(scoped_thread& x) noexcept;

    typedef thread::id id;

    id get_id() const noexcept;

    bool joinable() const noexcept;
    void join();
#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_join_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_join_until(const chrono::time_point<Clock, Duration>& t);
#endif

    void detach();

    static unsigned hardware_concurrency() noexcept;
    static unsigned physical_concurrency() noexcept;

    typedef thread::native_handle_type native_handle_type;
    native_handle_type native_handle();

#ifdef defined BOOST_THREAD_PROVIDES_INTERRUPTIONS
    void interrupt();
    bool interruption_requested() const noexcept;
#endif

};

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

RAI `thread` wrapper adding a specific destroyer allowing to master what can be done at destruction time.

CallableThread: A callable void(thread&). The default is join_if_joinable.

thread std::thread destructor terminates the program if the thread is not joinable. Having a wrapper that can join the thread before destroying it seems a natural need.

Remark: `scoped_thread` is not a `thread` as `thread` is not designed to be derived from as a polymorphic type.

Anyway `scoped_thread` can be used in most of the contexts a `thread` could be used as it has the same non-deprecated interface with the exception of the construction.

Example

```
boost::scoped_thread<> t((boost::thread(F)));
t.interrupt();
```

Default Constructor

```
scoped_thread() noexcept;
```

Effects: Constructs a `scoped_thread` instance that wraps to *Not-a-Thread*.

Postconditions: `this->get_id()==thread::id()`

Throws: Nothing

Move Constructor

```
scoped_thread(scoped_thread&& other) noexcept;
```

Effects: Transfers ownership of the `scoped_thread` managed by `other` (if any) to the newly constructed `scoped_thread` instance.

Postconditions: `other.get_id()==thread::id()` and `get_id()` returns the value of `other.get_id()` prior to the construction

Throws: Nothing

Move assignment operator

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

Effects: Transfers ownership of the `scoped_thread` managed by `other` (if any) to `*this`.

- if defined `BOOST_THREAD_DONT_PROVIDE_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE`:
If there was a `scoped_thread` previously associated with `*this` then that `scoped_thread` is detached,
DEPRECATED

- if defined `BOOST_THREAD_PROVIDES_THREAD_MOVE_ASSIGN_CALLS_TERMINATE_IF_JOINABLE`: If
the `scoped_thread` is joinable calls to `std::terminate`.

Postconditions: `other->get_id()==thread::id()` and `get_id()` returns the value of `other.get_id()` prior to the assignment.

Throws: Nothing

Move Constructor from a `thread`

```
scoped_thread(thread&& t);
```

Effects: move the thread to own `t_`.

Postconditions: `*this.t_` refers to the newly created thread of execution and `this->get_id()!=thread::id()`.

Throws: Nothing

Move Constructor from a Callable

```
template <typename F&&, typename ...Args>
explicit scoped_thread(F&&, Args&&...);
```

Effects: Construct a internal thread in place.

Postconditions: `*this.t_` refers to the newly created thread of execution and `this->get_id()!=thread::id()`.

Throws: Any exception the thread construction can throw.

Destructor

```
~scoped_thread();
```

Effects: Equivalent to `CallableThread()(t_)`.

Throws: Nothing: The `CallableThread()(t_)` should not throw when joining the thread as the scoped variable is on a scope outside the thread function.

Member function `joinable()`

```
bool joinable() const noexcept;
```

Returns: Equivalent to return `t_.joinable()`.

Throws: Nothing

Member function `join()`

```
void join();
```

Effects: Equivalent to `t_.join()`.

Member function `try_join_for()`

```
template <class Rep, class Period>
bool try_join_for(const chrono::duration<Rep, Period>& rel_time);
```

Effects: Equivalent to return `t_.try_join_for(rel_time)`.

Member function `try_join_until()`

```
template <class Clock, class Duration>
bool try_join_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Effects: Equivalent to return `t_.try_join_until(abs_time)`.

Member function `detach()`

```
void detach();
```

Effects: Equivalent to `t_.detach()`.

Member function `get_id()`

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

Effects: Equivalent to return `t_.get_id()`.

Member function `interrupt()`

```
void interrupt();
```

Effects: Equivalent to `t_.interrupt()`.

Static member function `hardware_concurrency()`

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

Effects: Equivalent to return `thread::hardware_concurrency()`.

Static member function `physical_concurrency()`

```
unsigned physical_concurrency() noexcept;
```

Effects: Equivalent to return `thread::physical_concurrency()`.

Member function `native_handle()`

```
typedef thread::native_handle_type native_handle_type;
native_handle_type native_handle();
```

Effects: Equivalent to return `t_.native_handle()`.

Member function `swap()`

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

Effects: Equivalent `t_.swap(other.t_)`.

Non-member function `swap(scoped_thread&,scoped_thread&)`

```
#include <boost/thread/scoped_thread.hpp>

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

Effects: `lhs.swap(rhs)`.

Synchronization

Tutorial

[Handling mutexes in C++](#) is an excellent tutorial. You need just replace `std` and `ting` by `boost`.

[Mutex, Lock, Condition Variable Rationale](#) adds rationale for the design decisions made for mutexes, locks and condition variables.

In addition to the C++11 standard locks, `Boost.Thread` provides other locks and some utilities that help the user to make their code thread-safe.

Internal Locking



Note

This tutorial is an adaptation of chapter Concurrency of the Object-Oriented Programming in the BETA Programming Language and of the paper of Andrei Alexandrescu "Multithreading and the C++ Type System" to the Boost library.

Concurrent threads of execution

Consider, for example, modeling a bank account class that supports simultaneous deposits and withdrawals from multiple locations (arguably the "Hello, World" of multithreaded programming).

From here a component is a model of the `Callable` concept.

In C++11 (Boost) concurrent execution of a component is obtained by means of the `std::thread(boost::thread)`:

```
boost::thread thread1(S);
```

where `S` is a model of `Callable`. The meaning of this expression is that execution of `S()` will take place concurrently with the current thread of execution executing the expression.

The following example includes a bank account of a person (Joe) and two components, one corresponding to a bank agent depositing money in Joe's account, and one representing Joe. Joe will only be withdrawing money from the account:

```

class BankAccount;

BankAccount JoesAccount;

void bankAgent()
{
    for (int i =10; i>0; --i) {
        //...
        JoesAccount.Deposit(500);
        //...
    }
}

void Joe() {
    for (int i =10; i>0; --i) {
        //...
        int myPocket = JoesAccount.Withdraw(100);
        std::cout << myPocket << std::endl;
        //...
    }
}

int main() {
    //...
    boost::thread thread1(bankAgent); // start concurrent execution of bankAgent
    boost::thread thread2(Joe); // start concurrent execution of Joe
    thread1.join();
    thread2.join();
    return 0;
}

```

From time to time, the bankAgent will deposit \$500 in JoesAccount. Joe will similarly withdraw \$100 from his account. These sentences describe that the bankAgent and Joe are executed concurrently.

Internal locking

The above example works well as long as the components bankAgent and Joe doesn't access JoesAccount at the same time. There is, however, no guarantee that this will not happen. We may use a mutex to guarantee exclusive access to each bank.

```

class BankAccount {
    boost::mutex mtx_;
    int balance_;
public:
    void Deposit(int amount) {
        mtx_.lock();
        balance_ += amount;
        mtx_.unlock();
    }
    void Withdraw(int amount) {
        mtx_.lock();
        balance_ -= amount;
        mtx_.unlock();
    }
    int GetBalance() {
        mtx_.lock();
        int b = balance_;
        mtx_.unlock();
        return b;
    }
};

```

Execution of the `Deposit` and `Withdraw` operations will no longer be able to make simultaneous access to `balance`.

A mutex is a simple and basic mechanism for obtaining synchronization. In the above example it is relatively easy to be convinced that the synchronization works correctly (in the absence of exception). In a system with several concurrent objects and several shared objects, it may be difficult to describe synchronization by means of mutexes. Programs that make heavy use of mutexes may be difficult to read and write. Instead, we shall introduce a number of generic classes for handling more complicated forms of synchronization and communication.

With the RAII idiom we can simplify a lot this using the scoped locks. In the code below, `guard`'s constructor locks the passed-in object `mtx_`, and `guard`'s destructor unlocks `mtx_`.

```
class BankAccount {
    boost::mutex mtx_; // explicit mutex declaration
    int balance_;
public:
    void Deposit(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ += amount;
    }
    void Withdraw(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ -= amount;
    }
    int GetBalance() {
        boost::lock_guard<boost::mutex> guard(mtx_);
        return balance_;
    }
};
```

The object-level locking idiom doesn't cover the entire richness of a threading model. For example, the model above is quite deadlock-prone when you try to coordinate multi-object transactions. Nonetheless, object-level locking is useful in many cases, and in combination with other mechanisms can provide a satisfactory solution to many threaded access problems in object-oriented programs.

Internal and external locking

The `BankAccount` class above uses internal locking. Basically, a class that uses internal locking guarantees that any concurrent calls to its public member functions don't corrupt an instance of that class. This is typically ensured by having each public member function acquire a lock on the object upon entry. This way, for any given object of that class, there can be only one member function call active at any moment, so the operations are nicely serialized.

This approach is reasonably easy to implement and has an attractive simplicity. Unfortunately, "simple" might sometimes morph into "simplistic."

Internal locking is insufficient for many real-world synchronization tasks. Imagine that you want to implement an ATM withdrawal transaction with the `BankAccount` class. The requirements are simple. The ATM transaction consists of two withdrawals—one for the actual money and one for the \$2 commission. The two withdrawals must appear in strict sequence; that is, no other transaction can exist between them.

The obvious implementation is erratic:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    acct.Withdraw(sum);
    // preemption possible
    acct.Withdraw(2);
}
```

The problem is that between the two calls above, another thread can perform another operation on the account, thus breaking the second design requirement.

In an attempt to solve this problem, let's lock the account from the outside during the two operations:


```
void ATMWithdrawal(BankAccount& acct, int sum) {
    boost::lock_guard<boost::mutex> guard(acct.mtx_); 1
    acct.Withdraw(sum);
    acct.Withdraw(2);
}
```

Notice that the code above doesn't compile, the `mtx_` field is private. We have two possibilities:

- make `mtx_` public which seems odd
- make the `BankAccount` lockable by adding the lock/unlock functions

We can add these functions explicitly

```
class BankAccount {
    boost::mutex mtx_;
    int balance_;
public:
    void Deposit(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ += amount;
    }
    void Withdraw(int amount) {
        boost::lock_guard<boost::mutex> guard(mtx_);
        balance_ -= amount;
    }
    void lock() {
        mtx_.lock();
    }
    void unlock() {
        mtx_.unlock();
    }
};
```

or inheriting from a class which add these lockable functions.

The `basic_lockable_adapter` class helps to define the `BankAccount` class as

```
class BankAccount
: public basic_lockable_adapter<mutex>
{
    int balance_;
public:
    void Deposit(int amount) {
        boost::lock_guard<BankAccount> guard(*this);
        balance_ += amount;
    }
    void Withdraw(int amount) {
        boost::lock_guard<BankAccount> guard(*this);
        balance_ -= amount;
    }
    int GetBalance() {
        boost::lock_guard<BankAccount> guard(*this);
        return balance_;
    }
};
```

and the code that doesn't compile becomes

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    boost::lock_guard<BankAccount> guard(acct);
    acct.Withdraw(sum);
    acct.Withdraw(2);
}
```

Notice that now `acct` is being locked by `Withdraw` after it has already been locked by `guard`. When running such code, one of two things happens.

- Your mutex implementation might support the so-called recursive mutex semantics. This means that the same thread can lock the same mutex several times successfully. In this case, the implementation works but has a performance overhead due to unnecessary locking. (The locking/unlocking sequence in the two `Withdraw` calls is not needed but performed anyway-and that costs time.)
- Your mutex implementation might not support recursive locking, which means that as soon as you try to acquire it the second time, it blocks-so the `ATMWithdrawal` function enters the dreaded deadlock.

As `boost::mutex` is not recursive, we need to use its recursive version `boost::recursive_mutex`.

```
class BankAccount
: public basic_lockable_adapter<recursive_mutex>
{
    // ...
};
```

The caller-ensured locking approach is more flexible and the most efficient, but very dangerous. In an implementation using caller-ensured locking, `BankAccount` still holds a mutex, but its member functions don't manipulate it at all. `Deposit` and `Withdraw` are not thread-safe anymore. Instead, the client code is responsible for locking `BankAccount` properly.

```
class BankAccount
: public basic_lockable_adapter<boost::mutex> {
    int balance_;
public:
    void Deposit(int amount) {
        balance_ += amount;
    }
    void Withdraw(int amount) {
        balance_ -= amount;
    }
};
```

Obviously, the caller-ensured locking approach has a safety problem. `BankAccount`'s implementation code is finite, and easy to reach and maintain, but there's an unbounded amount of client code that manipulates `BankAccount` objects. In designing applications, it's important to differentiate between requirements imposed on bounded code and unbounded code. If your class makes undue requirements on unbounded code, that's usually a sign that encapsulation is out the window.

To conclude, if in designing a multi-threaded class you settle on internal locking, you expose yourself to inefficiency or deadlocks. On the other hand, if you rely on caller-provided locking, you make your class error-prone and difficult to use. Finally, external locking completely avoids the issue by leaving it all to the client code.

External Locking -- `strict_lock` and `externally_locked` classes



Note

This tutorial is an adaptation of the paper by Andrei Alexandrescu "Multithreading and the C++ Type System" to the Boost library.

Locks as permits

So what to do? Ideally, the `BankAccount` class should do the following:

- Support both locking models (internal and external).
- Be efficient; that is, use no unnecessary locking.
- Be safe; that is, `BankAccount` objects cannot be manipulated without appropriate locking.

Let's make a worthwhile observation: Whenever you lock a `BankAccount`, you do so by using a `lock_guard<BankAccount>` object. Turning this statement around, wherever there's a `lock_guard<BankAccount>`, there's also a locked `BankAccount` somewhere. Thus, you can think of-and use-a `lock_guard<BankAccount>` object as a permit. Owning a `lock_guard<BankAccount>` gives you rights to do certain things. The `lock_guard<BankAccount>` object should not be copied or aliased (it's not a transmissible permit).

1. As long as a permit is still alive, the `BankAccount` object stays locked.
2. When the `lock_guard<BankAccount>` is destroyed, the `BankAccount`'s mutex is released.

The net effect is that at any point in your code, having access to a `lock_guard<BankAccount>` object guarantees that a `BankAccount` is locked. (You don't know exactly which `BankAccount` is locked, however-an issue that we'll address soon.)

For now, let's make a couple of enhancements to the `lock_guard` class template defined in `Boost.Thread`. We'll call the enhanced version `strict_lock`. Essentially, a `strict_lock`'s role is only to live on the stack as an automatic variable. `strict_lock` must adhere to a non-copy and non-alias policy. `strict_lock` disables copying by making the copy constructor and the assignment operator private.

```
template <typename Lockable>
class strict_lock {
public:
    typedef Lockable lockable_type;

    explicit strict_lock(lockable_type& obj) : obj_(obj) {
        obj.lock(); // locks on construction
    }
    strict_lock() = delete;
    strict_lock(strict_lock const&) = delete;
    strict_lock& operator=(strict_lock const&) = delete;

    ~strict_lock() { obj_.unlock(); } // unlocks on destruction

    bool owns_lock(mutex_type const* l) const noexcept // strict lockers specific function
    {
        return l == &obj_;
    }
private:
    lockable_type& obj_;
};
```

Silence can be sometimes louder than words-what's forbidden to do with a `strict_lock` is as important as what you can do. Let's see what you can and what you cannot do with a `strict_lock` instantiation:

- You can create a `strict_lock<T>` only starting from a valid `T` object. Notice that there is no other way you can create a `strict_lock<T>`.

```
BankAccount myAccount("John Doe", "123-45-6789");
strict_lock<BankAccount> myLock(myAccount); // ok
```

- You cannot copy `strict_locks` to one another. In particular, you cannot pass `strict_locks` by value to functions or have them returned by functions:

```
extern strict_lock<BankAccount> Foo(); // compile-time error
extern void Bar(strict_lock<BankAccount>); // compile-time error
```

- However, you still can pass `strict_locks` by reference to and from functions:

```
// ok, Foo returns a reference to strict_lock<BankAccount>
extern strict_lock<BankAccount>& Foo();
// ok, Bar takes a reference to strict_lock<BankAccount>
extern void Bar(strict_lock<BankAccount>&);
```

All these rules were put in place with one purpose-enforcing that owning a `strict_lock<T>` is a reasonably strong guarantee that

1. you locked a `T` object, and
2. that object will be unlocked at a later point.

Now that we have such a strict `strict_lock`, how do we harness its power in defining a safe, flexible interface for `BankAccount`? The idea is as follows:

- Each of `BankAccount`'s interface functions (in our case, `Deposit` and `Withdraw`) comes in two overloaded variants.
- One version keeps the same signature as before, and the other takes an additional argument of type `strict_lock<BankAccount>`. The first version is internally locked; the second one requires external locking. External locking is enforced at compile time by requiring client code to create a `strict_lock<BankAccount>` object.
- `BankAccount` avoids code bloating by having the internal locked functions forward to the external locked functions, which do the actual job.

A little code is worth 1,000 words, a (hacked into) saying goes, so here's the new `BankAccount` class:

```
class BankAccount
: public basic_lockable_adapter<boost::recursive_mutex>
{
    int balance_;
public:
    void Deposit(int amount, strict_lock<BankAccount>&) {
        // Externally locked
        balance_ += amount;
    }
    void Deposit(int amount) {
        strict_lock<boost::mutex> guard(*this); // Internally locked
        Deposit(amount, guard);
    }
    void Withdraw(int amount, strict_lock<BankAccount>&) {
        // Externally locked
        balance_ -= amount;
    }
    void Withdraw(int amount) {
        strict_lock<boost::mutex> guard(*this); // Internally locked
        Withdraw(amount, guard);
    }
};
```

Now, if you want the benefit of internal locking, you simply call `Deposit(int)` and `Withdraw(int)`. If you want to use external locking, you lock the object by constructing a `strict_lock<BankAccount>` and then you call `Deposit(int, strict_lock<BankAccount>&)` and `Withdraw(int, strict_lock<BankAccount>&)`. For example, here's the `ATMWithdrawal` function implemented correctly:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    strict_lock<BankAccount> guard(acct);
    acct.Withdraw(sum, guard);
    acct.Withdraw(2, guard);
}
```

This function has the best of both worlds-it's reasonably safe and efficient at the same time.

It's worth noting that `strict_lock` being a template gives extra safety compared to a straight polymorphic approach. In such a design, `BankAccount` would derive from a `Lockable` interface. `strict_lock` would manipulate `Lockable` references so there's no need for templates. This approach is sound; however, it provides fewer compile-time guarantees. Having a `strict_lock` object would only tell that some object derived from `Lockable` is currently locked. In the templated approach, having a `strict_lock<BankAccount>` gives a stronger guarantee-it's a `BankAccount` that stays locked.

There's a weasel word in there-I mentioned that `ATMWithdrawal` is reasonably safe. It's not really safe because there's no enforcement that the `strict_lock<BankAccount>` object locks the appropriate `BankAccount` object. The type system only ensures that some `BankAccount` object is locked. For example, consider the following phony implementation of `ATMWithdrawal`:

```
void ATMWithdrawal(BankAccount& acct, int sum) {
    BankAccount fakeAcct("John Doe", "123-45-6789");
    strict_lock<BankAccount> guard(fakeAcct);
    acct.Withdraw(sum, guard);
    acct.Withdraw(2, guard);
}
```

This code compiles warning-free but obviously doesn't do the right thing-it locks one account and uses another.

It's important to understand what can be enforced within the realm of the C++ type system and what needs to be enforced at runtime. The mechanism we've put in place so far ensures that some `BankAccount` object is locked during the call to `BankAccount::Withdraw(int, strict_lock<BankAccount>&)`. We must enforce at runtime exactly what object is locked.

If our scheme still needs runtime checks, how is it useful? An unwary or malicious programmer can easily lock the wrong object and manipulate any `BankAccount` without actually locking it.

First, let's get the malice issue out of the way. C is a language that requires a lot of attention and discipline from the programmer. C++ made some progress by asking a little less of those, while still fundamentally trusting the programmer. These languages are not concerned with malice (as Java is, for example). After all, you can break any C/C++ design simply by using casts "appropriately" (if appropriately is an, er, appropriate word in this context).

The scheme is useful because the likelihood of a programmer forgetting about any locking whatsoever is much greater than the likelihood of a programmer who does remember about locking, but locks the wrong object.

Using `strict_lock` permits compile-time checking of the most common source of errors, and runtime checking of the less frequent problem.

Let's see how to enforce that the appropriate `BankAccount` object is locked. First, we need to add a member function to the `strict_lock` class template. The `bool strict_lock<T>::owns_lock(Lockable*)` function returns a reference to the locked object.

```
template <class Lockable> class strict_lock {
    ... as before ...
public:
    bool owns_lock(Lockable* mtx) const { return mtx==&obj_; }
};
```

Second, `BankAccount` needs to use this function compare the locked object against this:

```

class BankAccount {
: public basic_lockable_adapter<boost::recursive_mutex>
    int balance_;
public:
    void Deposit(int amount, strict_lock<BankAccount>& guard) {
        // Externally locked
        if (!guard.owns_lock(*this))
            throw "Locking Error: Wrong Object Locked";
        balance_ += amount;
    }
    // ...
};

```

The overhead incurred by the test above is much lower than locking a recursive mutex for the second time.

Improving External Locking

Now let's assume that BankAccount doesn't use its own locking at all, and has only a thread-neutral implementation:

```

class BankAccount {
    int balance_;
public:
    void Deposit(int amount) {
        balance_ += amount;
    }
    void Withdraw(int amount) {
        balance_ -= amount;
    }
};

```

Now you can use BankAccount in single-threaded and multi-threaded applications alike, but you need to provide your own synchronization in the latter case.

Say we have an AccountManager class that holds and manipulates a BankAccount object:

```

class AccountManager
: public basic_lockable_adapter<boost::mutex>
{
    BankAccount checkingAcct_;
    BankAccount savingsAcct_;
    ...
};

```

Let's also assume that, by design, AccountManager must stay locked while accessing its BankAccount members. The question is, how can we express this design constraint using the C++ type system? How can we state "You have access to this BankAccount object only after locking its parent AccountManager object"?

The solution is to use a little bridge template `externally_locked` that controls access to a BankAccount.

```

template <typename T, typename Lockable>
class externally_locked {
    BOOST_CONCEPT_ASSERT((LockableConcept<Lockable>));

public:
    externally_locked(T& obj, Lockable& lockable)
        : obj_(obj)
        , lockable_(lockable)
    {}

    externally_locked(Lockable& lockable)
        : obj_()
        , lockable_(lockable)
    {}

    T& get(strict_lock<Lockable>& lock) {

#ifdef BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED
        if (!lock.owns_lock(&lockable_)) throw lock_error(
            "run time check throw if not locks the same");
#endif
        return obj_;
    }

    void set(const T& obj, Lockable& lockable) {
        obj_ = obj;
        lockable_ = lockable;
    }

private:
    T obj_;
    Lockable& lockable_;
};

```

`externally_locked` cloaks an object of type `T`, and actually provides full access to that object through the `get` and `set` member functions, provided you pass a reference to a `strict_lock<Owner>` object.

Instead of making `checkingAcct_` and `savingsAcct_` of type `BankAccount`, `AccountManager` holds objects of type `externally_locked<BankAccount, AccountManager>`:

```

class AccountManager
    : public basic_lockable_adapter<thread_mutex>
{
public:
    typedef basic_lockable_adapter<thread_mutex> lockable_base_type;
    AccountManager()
        : checkingAcct_( *this )
        , savingsAcct_( *this )
    {}
    inline void Checking2Savings(int amount);
    inline void AMoreComplicatedChecking2Savings(int amount);
private:

    externally_locked<BankAccount, AccountManager> checkingAcct_;
    externally_locked<BankAccount, AccountManager> savingsAcct_;
};

```

The pattern is the same as before - to access the `BankAccount` object cloaked by `checkingAcct_`, you need to call `get`. To call `get`, you need to pass it a `strict_lock<AccountManager>`. The one thing you have to take care of is to not hold pointers or references you obtained by calling `get`. If you do that, make sure that you don't use them after the `strict_lock` has been destroyed. That is, if you alias the cloaked objects, you're back from "the compiler takes care of that" mode to "you must pay attention" mode.

Typically, you use `externally_locked` as shown below. Suppose you want to execute an atomic transfer from your checking account to your savings account:

```
void AccountManager::Checking2Savings(int amount) {
    strict_lock<AccountManager> guard(*this);
    checkingAcct_.get(guard).Withdraw(amount);
    savingsAcct_.get(guard).Deposit(amount);
}
```

We achieved two important goals. First, the declaration of `checkingAcct_` and `savingsAcct_` makes it clear to the code reader that that variable is protected by a lock on an `AccountManager`. Second, the design makes it impossible to manipulate the two accounts without actually locking a `BankAccount`. `externally_locked` is what could be called active documentation.

Allowing other strict locks

Now imagine that the `AccountManager` function needs to take a `unique_lock` in order to reduce the critical regions. And at some time it needs to access to the `checkingAcct_`. As `unique_lock` is not a strict lock the following code doesn't compile:

```
void AccountManager::AMoreComplicatedChecking2Savings(int amount) {
    unique_lock<AccountManager> guard(*this, defer_lock);
    if (some_condition()) {
        guard.lock();
    }
    checkingAcct_.get(guard).Withdraw(amount); // COMPILE ERROR
    savingsAcct_.get(guard).Deposit(amount);  // COMPILE ERROR
    do_something_else();
}
```

We need a way to transfer the ownership from the `unique_lock` to a `strict_lock` during the time we are working with `savingsAcct_` and then restore the ownership on `unique_lock`.

```
void AccountManager::AMoreComplicatedChecking2Savings(int amount) {
    unique_lock<AccountManager> guard1(*this, defer_lock);
    if (some_condition()) {
        guard1.lock();
    }
    {
        strict_lock<AccountManager> guard(guard1);
        checkingAcct_.get(guard).Withdraw(amount);
        savingsAcct_.get(guard).Deposit(amount);
    }
    guard1.unlock();
}
```

In order to make this code compilable we need to store either a `Lockable` or a `unique_lock<Lockable>` reference depending on the constructor. We also need to store which kind of reference we have stored, and in the destructor call either to the `Lockable` `unlock` or restore the ownership.

This seems too complicated to me. Another possibility is to define a nested strict lock class. The drawback is that instead of having only one strict lock we have two and we need either to duplicate every function taking a `strict_lock` or make these function templates. The problem with template functions is that we don't profit anymore of the C++ type system. We must add some static metafunction that checks that the `Locker` parameter is a strict lock. The problem is that we can not really check this or can we?. The `is_strict_lock` metafunction must be specialized by the strict lock developer. We need to believe it "sur parole". The advantage is that now we can manage with more than two strict locks without changing our code. This is really nice.

Now we need to state that both classes are `strict_locks`.


```

template <typename Locker>
struct is_strict_lock : mpl::false_ {};

template <typename Lockable>
struct is_strict_lock<strict_lock<Lockable> > : mpl::true_ {}

template <typename Locker>
struct is_strict_lock<nested_strict_lock<Locker> > : mpl::true_ {}

```

Well let me show what this nested_strict_lock class looks like and the impacts on the externally_locked class and the AccountManager::AMoreComplicatedFunction function.

First nested_strict_lock class will store on a temporary lock the Locker, and transfer the lock ownership on the constructor. On destruction it will restore the ownership. Note the use of lock_traits and that the Locker needs to have a reference to the mutex otherwise an exception is thrown.

```

template <typename Locker >
class nested_strict_lock
{
    BOOST_CONCEPT_ASSERT((MovableLockerConcept<Locker>));
public:
    typedef typename lockable_type<Locker>::type lockable_type;
    typedef typename syntactic_lock_traits<lockable_type>::lock_error lock_error;

    nested_strict_lock(Locker& lock)
        : lock_(lock) // Store reference to locker
        , tmp_lock_(lock.move()) // Move ownership to temporary locker
    {
        #ifdef BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED
        if (tmp_lock_.mutex()==0) {
            lock_=tmp_lock_.move(); // Rollback for coherency purposes
            throw lock_error();
        }
        #endif
        if (!tmp_lock_) tmp_lock_.lock(); // ensures it is locked
    }
    ~nested_strict_lock() {
        lock_=tmp_lock_.move(); // Move ownership to nesting locker
    }
    bool owns_lock() const { return true; }
    lockable_type* mutex() const { return tmp_lock_.mutex(); }
    bool owns_lock(lockable_type* l) const { return l==mutex(); }

private:
    Locker& lock_;
    Locker tmp_lock_;
};

```

The externally_locked get function is now a template function taking a Locker as parameters instead of a strict_lock. We can add test in debug mode that ensure that the Lockable object is locked.

```

template <typename T, typename Lockable>
class externally_locked {
public:
    // ...
    template <class Locker>
    T& get(Locker& lock) {
        BOOST_CONCEPT_ASSERT((StrictLockerConcept<Locker>));

        BOOST_STATIC_ASSERT((is_strict_lock<Locker>::value)); // locker is a strict locker "sur la
parole"
        BOOST_STATIC_ASSERT((is_same<Lockable,
                                typename lockable_type<Locker>::type>::value)); // that locks the same type
#ifdef BOOST_THREAD_EXTERNALLY_LOCKED_DONT_CHECK_OWNERSHIP // define BOOST_THREAD_EXTERNALLY_LOCKED_NO_CHECK_OWNERSHIP if you don't want to check locker ownership
        if (! lock ) throw lock_error(); // run time check throw if no locked
#endif
#ifdef BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED
        if (!lock.owns_lock(&lockable_)) throw lock_error();
#endif
        return obj_;
    }
};

```

The `AccountManager::AMoreComplicatedFunction` function needs only to replace the `strict_lock` by a nested `strict_lock`.

```

void AccountManager::AMoreComplicatedChecking2Savings(int amount) {
    unique_lock<AccountManager> guard1(*this);
    if (some_condition()) {
        guard1.lock();
    }
    {
        nested_strict_lock<unique_lock<AccountManager> > guard(guard1);
        checkingAcct_.get(guard).Withdraw(amount);
        savingsAcct_.get(guard).Deposit(amount);
    }
    guard1.unlock();
}

```

Executing Around a Function

In particular, the library provides a way to lock around the execution of a function.

```

template <class Lockable, class Function, class... Args>
auto with_lock_guard(
    Lockable& m,
    Function&& func,
    Args&&... args
) -> decltype(func(boost::forward<Args>(args)...)) {
    boost::lock_guard<Lockable> lock(m);
    return func(boost::forward<Args>(args)...);
}

```

that can be used with regular functions:

```
int func(int, int&);
//...
boost::mutex m;
int a;
int result = boost::with_lock_guard(m, func, 1, boost::ref(a));
```

with `boost::bind`:

```
int result = boost::with_lock_guard(
    m, boost::bind(func, 2, boost::ref(a))
);
```

or with lambda expression:

```
int a;
int result = boost::with_lock_guard(
    m,
    [&a](int x) {
        // this scope is protected by mutex m
        a = 3;
        return x + 4;
    },
    5
);
```

Mutex Concepts

A mutex object facilitates protection against data races and allows thread-safe synchronization of data between threads. A thread obtains ownership of a mutex object by calling one of the lock functions and relinquishes ownership by calling the corresponding unlock function. Mutexes may be either recursive or non-recursive, and may grant simultaneous ownership to one or many threads. **Boost.Thread** supplies recursive and non-recursive mutexes with exclusive ownership semantics, along with a shared ownership (multiple-reader / single-writer) mutex.

Boost.Thread supports four basic concepts for lockable objects: [Lockable](#), [TimedLockable](#), [SharedLockable](#) and [Upgrade-Lockable](#). Each mutex type implements one or more of these concepts, as do the various lock types.

BasicLockable Concept

```
// #include <boost/thread/lockable_concepts.hpp>

namespace boost
{
    template<typename L>
    class BasicLockable; // EXTENSION
}
```

The [BasicLockable](#) concept models exclusive ownership. A type `L` meets the [BasicLockable](#) requirements if the following expressions are well-formed and have the specified semantics (`m` denotes a value of type `L`):

- `m.lock();`
- `m.unlock();`

Lock ownership acquired through a call to `lock()` must be released through a call to `unlock()`.

m.lock();

Requires:	The calling thread doesn't own the mutex if the mutex is not recursive.
Effects:	The current thread blocks until ownership can be obtained for the current thread.
Synchronization:	Prior <code>unlock()</code> operations on the same object synchronizes with this operation.
Postcondition:	The current thread owns <code>m</code> .
Return type:	<code>void</code> .
Throws:	<code>lock_error</code> if an error occurs.
Error Conditions:	operation_not_permitted : if the thread does not have the privilege to perform the operation. resource_deadlock_would_occur : if the implementation detects that a deadlock would occur. device_or_resource_busy : if the mutex is already locked and blocking is not possible.
Thread safety:	If an exception is thrown then a lock shall not have been acquired for the current thread.

m.unlock();

Requires:	The current thread owns <code>m</code> .
Synchronization:	This operation synchronizes with subsequent lock operations that obtain ownership on the same object.
Effects:	Releases a lock on <code>m</code> by the current thread.
Return type:	<code>void</code> .
Throws:	Nothing.

is_basic_lockable trait -- EXTENSION

```
// #include <boost/thread/lockable_traits.hpp>

namespace boost
{
    namespace sync
    {
        template<typename L>
        class is_basic_lockable; // EXTENSION
    }
}
```

Some of the algorithms on mutexes use this trait via SFINAE.

This trait is `true_type` if the parameter `L` meets the `Lockable` requirements.

**Warning**

If `BOOST_THREAD_NO_AUTO_DETECT_MUTEX_TYPES` is defined you will need to specialize this trait for the models of `BasicLockable` you could build.

Lockable Concept

```
// #include <boost/thread/lockable_concepts.hpp>
namespace boost
{
    template<typename L>
    class Lockable;
}
```

A type `L` meets the `Lockable` requirements if it meets the `BasicLockable` requirements and the following expressions are well-formed and have the specified semantics (`m` denotes a value of type `L`):

- `m.try_lock()`

Lock ownership acquired through a call to `try_lock()` must be released through a call to `unlock()`.

`m.try_lock()`

Requires:	The calling thread doesn't own the mutex if the mutex is not recursive.
Effects:	Attempt to obtain ownership for the current thread without blocking.
Synchronization:	If <code>try_lock()</code> returns true, prior <code>unlock()</code> operations on the same object synchronize with this operation.
Note:	Since <code>lock()</code> does not synchronize with a failed subsequent <code>try_lock()</code> , the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures.
Return type:	<code>bool</code> .
Returns:	true if ownership was obtained for the current thread, false otherwise.
Postcondition:	If the call returns true, the current thread owns the <code>m</code> .
Throws:	Nothing.

`is_lockable` trait -- EXTENSION

```
// #include <boost/thread/lockable_traits.hpp>
namespace boost
{
    namespace sync
    {
        template<typename L>
        class is_lockable; // EXTENSION
    }
}
```

Some of the algorithms on mutexes use this trait via SFINAE.

This trait is `true_type` if the parameter `L` meets the `Lockable` requirements.



Warning

If `BOOST_THREAD_NO_AUTO_DETECT_MUTEX_TYPES` is defined you will need to specialize this trait for the models of `Lockable` you could build.

Recursive Lockable Concept

The user could require that the mutex passed to an algorithm is a recursive one. Whether a lockable is recursive or not can not be checked using template meta-programming. This is the motivation for the following trait.

`is_recursive_mutex_sur_parole` trait -- EXTENSION

```
// #include <boost/thread/lockable_traits.hpp>

namespace boost
{
    namespace sync
    {
        template<typename L>
        class is_recursive_mutex_sur_parole: false_type; // EXTENSION
        template<>
        class is_recursive_mutex_sur_parole<recursive_mutex>: true_type; // EXTENSION
        template<>
        class is_recursive_mutex_sur_parole<timed_recursive_mutex>: true_type; // EXTENSION
    }
}
```

The trait `is_recursive_mutex_sur_parole` is `false_type` by default and is specialized for the provide `recursive_mutex` and `timed_recursive_mutex`.

It should be specialized by the user providing other model of recursive lockable.

`is_recursive_basic_lockable` trait -- EXTENSION

```
// #include <boost/thread/lockable_traits.hpp>
namespace boost
{
    namespace sync
    {
        template<typename L>
        class is_recursive_basic_lockable; // EXTENSION
    }
}
```

This traits is `true_type` if `is_basic_lockable` and `is_recursive_mutex_sur_parole`.

`is_recursive_lockable` trait -- EXTENSION

```
// #include <boost/thread/lockable_traits.hpp>
namespace boost
{
    namespace sync
    {
        template<typename L>
        class is_recursive_lockable; // EXTENSION
    }
}
```

This traits is `true_type` if `is_lockable` and `is_recursive_mutex_sur_parole`.

TimedLockable Concept

```
// #include <boost/thread/lockable_concepts.hpp>

namespace boost
{
    template<typename L>
    class TimedLockable; // EXTENSION
}
```

The **TimedLockable concept** refines the **Lockable concept** to add support for timeouts when trying to acquire the lock.

A type **L** meets the **TimedLockable** requirements if it meets the **Lockable** requirements and the following expressions are well-formed and have the specified semantics.

Variables:

- **m** denotes a value of type **L**,
- **rel_time** denotes a value of an instantiation of `chrono::duration`, and
- **abs_time** denotes a value of an instantiation of `chrono::time_point`:

Expressions:

- `m.try_lock_for(rel_time)`
- `m.try_lock_until(abs_time)`

Lock ownership acquired through a call to `try_lock_for` or `try_lock_until` must be released through a call to `unlock`.

m.try_lock_until(abs_time)

Requires:	The calling thread doesn't owns the mutex if the mutex is not recursive.
Effects:	Attempt to obtain ownership for the current thread. Blocks until ownership can be obtained, or the specified time is reached. If the specified time has already passed, behaves as <code>try_lock()</code> .
Synchronization:	If <code>try_lock_until()</code> returns true, prior <code>unlock()</code> operations on the same object synchronize with this operation.
Return type:	<code>bool</code> .
Returns:	true if ownership was obtained for the current thread, false otherwise.
Postcondition:	If the call returns true, the current thread owns m .
Throws:	Nothing.

m.try_lock_for(rel_time)

Requires:	The calling thread doesn't owns the mutex if the mutex is not recursive.
Effects:	As-if <code>try_lock_until(chrono::steady_clock::now() + rel_time)</code> .
Synchronization:	If <code>try_lock_for()</code> returns true, prior <code>unlock()</code> operations on the same object synchronize with this operation.



Warning

DEPRECATED since 4.00. The following expressions were required on version 2, but are now deprecated.

Use instead `try_lock_for`, `try_lock_until`.

Variables:

- `rel_time` denotes a value of an instantiation of an unspecified `DurationType` arithmetic compatible with `boost::system_time`, and
- `abs_time` denotes a value of an instantiation of `boost::system_time`:

Expressions:

- `m.timed_lock(rel_time)`
- `m.timed_lock(abs_time)`

Lock ownership acquired through a call to `timed_lock()` must be released through a call to `unlock()`.

`m.timed_lock(abs_time)`

Effects: Attempt to obtain ownership for the current thread. Blocks until ownership can be obtained, or the specified time is reached. If the specified time has already passed, behaves as `try_lock()`.

Returns: `true` if ownership was obtained for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread owns `m`.

Throws: `lock_error` if an error occurs.

`m.timed_lock(rel_time)`

Effects: As-if `timed_lock(boost::get_system_time()+rel_time)`.

SharedLockable Concept -- C++14

```
// #include <boost/thread/lockable_concepts.hpp>

namespace boost
{
    template<typename L>
    class SharedLockable; // C++14
}
```

The `SharedLockable` concept is a refinement of the `TimedLockable` concept that allows for *shared ownership* as well as *exclusive ownership*. This is the standard multiple-reader / single-write model: at most one thread can have exclusive ownership, and if any thread does have exclusive ownership, no other threads can have shared or exclusive ownership. Alternatively, many threads may have shared ownership.

A type `L` meets the `SharedLockable` requirements if it meets the `TimedLockable` requirements and the following expressions are well-formed and have the specified semantics.

Variables:

- `m` denotes a value of type `L`,
- `rel_time` denotes a value of an instantiation of `chrono::duration`, and

- `abs_time` denotes a value of an instantiation of `chrono::time_point`:

Expressions:

- `m.lock_shared()`;
- `m.try_lock_shared()`
- `m.try_lock_shared_for(rel_time)`
- `m.try_lock_shared_until(abs_time)`
- `m.unlock_shared()`;

Lock ownership acquired through a call to `lock_shared()`, `try_lock_shared()`, `try_lock_shared_for` or `try_lock_shared_until` must be released through a call to `unlock_shared()`.

`m.lock_shared()`

Effects: The current thread blocks until shared ownership can be obtained for the current thread.

Postcondition: The current thread has shared ownership of `m`.

Throws: `lock_error` if an error occurs.

`m.try_lock_shared()`

Effects: Attempt to obtain shared ownership for the current thread without blocking.

Returns: `true` if shared ownership was obtained for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has shared ownership of `m`.

Throws: `lock_error` if an error occurs.

`m.try_lock_shared_for(rel_time)`

Effects: Attempt to obtain shared ownership for the current thread. Blocks until shared ownership can be obtained, or the specified duration is elapsed. If the specified duration is already elapsed, behaves as `try_lock_shared()`.

Returns: `true` if shared ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has shared ownership of `m`.

Throws: `lock_error` if an error occurs.

`m.try_lock_shared_until(abs_time)`

Effects: Attempt to obtain shared ownership for the current thread. Blocks until shared ownership can be obtained, or the specified time is reached. If the specified time has already passed, behaves as `try_lock_shared()`.

Returns: `true` if shared ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has shared ownership of `m`.

Throws: `lock_error` if an error occurs.

m.unlock_shared()

Precondition:	The current thread has shared ownership of m.
Effects:	Releases shared ownership of m by the current thread.
Postcondition:	The current thread no longer has shared ownership of m.
Throws:	Nothing

**Warning**

DEPRECATED since 3.00. The following expressions were required on version 2, but are now deprecated.

Use instead `try_lock_shared_for`, `try_lock_shared_until`.

Variables:

- `abs_time` denotes a value of an instantiation of `boost::system_time`:

Expressions:

- `m.timed_lock_shared(abs_time);`

Lock ownership acquired through a call to `timed_lock_shared()` must be released through a call to `unlock_shared()`.

m.timed_lock_shared(abs_time)

Effects:	Attempt to obtain shared ownership for the current thread. Blocks until shared ownership can be obtained, or the specified time is reached. If the specified time has already passed, behaves as <code>try_lock_shared()</code> .
Returns:	<code>true</code> if shared ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has shared ownership of m.
Throws:	<code>lock_error</code> if an error occurs.

UpgradeLockable Concept -- EXTENSION

```
// #include <boost/thread/lockable_concepts.hpp>

namespace boost
{
    template<typename L>
    class UpgradeLockable; // EXTENSION
}
```

The **UpgradeLockable concept** is a refinement of the **SharedLockable concept** that allows for *upgradable ownership* as well as *shared ownership* and *exclusive ownership*. This is an extension to the multiple-reader / single-write model provided by the **SharedLockable concept**: a single thread may have *upgradable ownership* at the same time as others have *shared ownership*. The thread with *upgradable ownership* may at any time attempt to upgrade that ownership to *exclusive ownership*. If no other threads have *shared ownership*, the upgrade is completed immediately, and the thread now has *exclusive ownership*, which must be relinquished by a call to `unlock()`, just as if it had been acquired by a call to `lock()`.

If a thread with *upgradable ownership* tries to upgrade whilst other threads have *shared ownership*, the attempt will fail and the thread will block until *exclusive ownership* can be acquired.

Ownership can also be *downgraded* as well as *upgraded*: exclusive ownership of an implementation of the [UpgradeLockable concept](#) can be downgraded to upgradable ownership or shared ownership, and upgradable ownership can be downgraded to plain shared ownership.

A type `L` meets the [UpgradeLockable](#) requirements if it meets the [SharedLockable](#) requirements and the following expressions are well-formed and have the specified semantics.

Variables:

- `m` denotes a value of type `L`,
- `rel_time` denotes a value of an instantiation of `chrono::duration`, and
- `abs_time` denotes a value of an instantiation of `chrono::time_point`:

Expressions:

- `m.lock_upgrade()`;
- `m.unlock_upgrade()`
- `m.try_lock_upgrade()`
- `m.try_lock_upgrade_for(rel_time)`
- `m.try_lock_upgrade_until(abs_time)`
- `m.unlock_and_lock_shared()`
- `m.unlock_and_lock_upgrade()`;
- `m.unlock_upgrade_and_lock()`;
- `m.try_unlock_upgrade_and_lock()`
- `m.try_unlock_upgrade_and_lock_for(rel_time)`
- `m.try_unlock_upgrade_and_lock_until(abs_time)`
- `m.unlock_upgrade_and_lock_shared()`;

If `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` is defined the following expressions are also required:

- `m.try_unlock_shared_and_lock()`;
- `m.try_unlock_shared_and_lock_for(rel_time)`;
- `m.try_unlock_shared_and_lock_until(abs_time)`;
- `m.try_unlock_shared_and_lock_upgrade()`;
- `m.try_unlock_shared_and_lock_upgrade_for(rel_time)`;
- `m.try_unlock_shared_and_lock_upgrade_until(abs_time)`;

Lock ownership acquired through a call to `lock_upgrade()` must be released through a call to `unlock_upgrade()`. If the ownership type is changed through a call to one of the `unlock_xxx_and_lock_yyy()` functions, ownership must be released through a call to the unlock function corresponding to the new level of ownership.

m.lock_upgrade()

Precondition:	The calling thread has no ownership of the mutex.
Effects:	The current thread blocks until upgrade ownership can be obtained for the current thread.
Postcondition:	The current thread has upgrade ownership of m.
Synchronization:	Prior <code>unlock_upgrade()</code> operations on the same object synchronize with this operation.
Throws:	<code>lock_error</code> if an error occurs.

m.unlock_upgrade()

Precondition:	The current thread has upgrade ownership of m.
Effects:	Releases upgrade ownership of m by the current thread.
Postcondition:	The current thread no longer has upgrade ownership of m.
Synchronization:	This operation synchronizes with subsequent lock operations that obtain ownership on the same object.
Throws:	Nothing

m.try_lock_upgrade()

Precondition:	The calling thread has no ownership of the mutex.
Effects:	Attempts to obtain upgrade ownership of the mutex for the calling thread without blocking. If upgrade ownership is not obtained, there is no effect and <code>try_lock_upgrade()</code> immediately returns.
Returns:	<code>true</code> if upgrade ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has upgrade ownership of m.
Synchronization:	If <code>try_lock_upgrade()</code> returns <code>true</code> , prior <code>unlock_upgrade()</code> operations on the same object synchronize with this operation.
Throws:	Nothing

m.try_lock_upgrade_for(rel_time)

Precondition:	The calling thread has no ownership of the mutex.
Effects:	If the tick period of <code>rel_time</code> is not exactly convertible to the native tick period, the duration shall be rounded up to the nearest native tick period. Attempts to obtain upgrade lock ownership for the calling thread within the relative timeout specified by <code>rel_time</code> . If the time specified by <code>rel_time</code> is less than or equal to <code>rel_time.zero()</code> , the function attempts to obtain ownership without blocking (as if by calling <code>try_lock_upgrade()</code>). The function returns within the timeout specified by <code>rel_time</code> only if it has obtained upgrade ownership of the mutex object.
Returns:	<code>true</code> if upgrade ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has upgrade ownership of m.
Synchronization:	If <code>try_lock_upgrade_for(rel_time)</code> returns <code>true</code> , prior <code>unlock_upgrade()</code> operations on the same object synchronize with this operation.
Throws:	Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.try_lock_upgrade_until(abs_time)`

Precondition: The calling thread has no ownership of the mutex.

Effects: The function attempts to obtain upgrade ownership of the mutex. If `abs_time` has already passed, the function attempts to obtain upgrade ownership without blocking (as if by calling `try_lock_upgrade()`). The function returns before the absolute timeout specified by `abs_time` only if it has obtained upgrade ownership of the mutex object.

Returns: `true` if upgrade ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has upgrade ownership of `m`.

Synchronization: If `try_lock_upgrade_until(abs_time)` returns `true`, prior `unlock_upgrade()` operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.try_unlock_shared_and_lock()`

Precondition: The calling thread must hold a shared lock on the mutex.

Effects: The function attempts to atomically convert the ownership from shared to exclusive for the calling thread without blocking. For this conversion to be successful, this thread must be the only thread holding any ownership of the lock. If the conversion is not successful, the shared ownership of `m` is retained.

Returns: `true` if exclusive ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has exclusive ownership of `m`.

Synchronization: If `try_unlock_shared_and_lock()` returns `true`, prior `unlock()` and subsequent lock operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.try_unlock_shared_and_lock_for(rel_time)`

Precondition: The calling thread shall hold a shared lock on the mutex.

Effects: If the tick period of `rel_time` is not exactly convertible to the native tick period, the duration shall be rounded up to the nearest native tick period. The function attempts to atomically convert the ownership from shared to exclusive for the calling thread within the relative timeout specified by `rel_time`. If the time specified by `rel_time` is less than or equal to `rel_time.zero()`, the function attempts to obtain exclusive ownership without blocking (as if by calling `try_unlock_shared_and_lock()`). The function shall return within the timeout specified by `rel_time` only if it has obtained exclusive ownership of the mutex object. For this conversion to be successful, this thread must be the only thread holding any ownership of the lock at the moment of conversion. If the conversion is not successful, the shared ownership of the mutex is retained.

Returns: `true` if exclusive ownership was acquired for the current thread, `false` otherwise.

Postcondition:	If the call returns <code>true</code> , the current thread has exclusive ownership of <code>m</code> .
Synchronization:	If <code>try_unlock_shared_and_lock_for(rel_time)</code> returns <code>true</code> , prior <code>unlock()</code> and subsequent lock operations on the same object synchronize with this operation.
Throws:	Nothing
Notes:	Available only if <code>BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION</code> and <code>BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN</code> is defined on Windows platform

`m.try_unlock_shared_and_lock_until(abs_time)`

Precondition:	The calling thread shall hold a shared lock on the mutex.
Effects:	The function attempts to atomically convert the ownership from shared to exclusive for the calling thread within the absolute timeout specified by <code>abs_time</code> . If <code>abs_time</code> has already passed, the function attempts to obtain exclusive ownership without blocking (as if by calling <code>try_unlock_shared_and_lock()</code>). The function shall return before the absolute timeout specified by <code>abs_time</code> only if it has obtained exclusive ownership of the mutex object. For this conversion to be successful, this thread must be the only thread holding any ownership of the lock at the moment of conversion. If the conversion is not successful, the shared ownership of the mutex is retained.
Returns:	<code>true</code> if exclusive ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has exclusive ownership of <code>m</code> .
Synchronization:	If <code>try_unlock_shared_and_lock_until(rel_time)</code> returns <code>true</code> , prior <code>unlock()</code> and subsequent lock operations on the same object synchronize with this operation.
Throws:	Nothing
Notes:	Available only if <code>BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION</code> and <code>BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN</code> is defined on Windows platform

`m.unlock_and_lock_shared()`

Precondition:	The calling thread shall hold an exclusive lock on <code>m</code> .
Effects:	Atomically converts the ownership from exclusive to shared for the calling thread.
Postcondition:	The current thread has shared ownership of <code>m</code> .
Synchronization:	This operation synchronizes with subsequent lock operations that obtain ownership of the same object.
Throws:	Nothing

`m.try_unlock_shared_and_lock_upgrade()`

Precondition:	The calling thread shall hold a shared lock on the mutex.
Effects:	The function attempts to atomically convert the ownership from shared to upgrade for the calling thread without blocking. For this conversion to be successful, there must be no thread holding upgrade ownership of this object. If the conversion is not successful, the shared ownership of the mutex is retained.
Returns:	<code>true</code> if upgrade ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has upgrade ownership of <code>m</code> .
Synchronization:	If <code>try_unlock_shared_and_lock_upgrade()</code> returns <code>true</code> , prior <code>unlock_upgrade()</code> and subsequent lock operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.try_unlock_shared_and_lock_upgrade_for(rel_time)`

Precondition: The calling thread shall hold a shared lock on the mutex.

Effects: If the tick period of `rel_time` is not exactly convertible to the native tick period, the duration shall be rounded up to the nearest native tick period. The function attempts to atomically convert the ownership from shared to upgrade for the calling thread within the relative timeout specified by `rel_time`. If the time specified by `rel_time` is less than or equal to `rel_time.zero()`, the function attempts to obtain upgrade ownership without blocking (as if by calling `try_unlock_shared_and_lock_upgrade()`). The function shall return within the timeout specified by `rel_time` only if it has obtained exclusive ownership of the mutex object. For this conversion to be successful, there must be no thread holding upgrade ownership of this object at the moment of conversion. If the conversion is not successful, the shared ownership of `m` is retained.

Returns: `true` if upgrade ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has upgrade ownership of `m`.

Synchronization: If `try_unlock_shared_and_lock_upgrade_for(rel_time)` returns `true`, prior `unlock_upgrade()` and subsequent lock operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.try_unlock_shared_and_lock_upgrade_until(abs_time)`

Precondition: The calling thread shall hold a shared lock on the mutex.

Effects: The function attempts to atomically convert the ownership from shared to upgrade for the calling thread within the absolute timeout specified by `abs_time`. If `abs_time` has already passed, the function attempts to obtain upgrade ownership without blocking (as if by calling `try_unlock_shared_and_lock_upgrade()`). The function shall return before the absolute timeout specified by `abs_time` only if it has obtained upgrade ownership of the mutex object. For this conversion to be successful, there must be no thread holding upgrade ownership of this object at the moment of conversion. If the conversion is not successful, the shared ownership of the mutex is retained.

Returns: `true` if upgrade ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has upgrade ownership of `m`.

Synchronization: If `try_unlock_shared_and_lock_upgrade_until(rel_time)` returns `true`, prior `unlock_upgrade()` and subsequent lock operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.unlock_and_lock_upgrade()`

Precondition: The current thread has exclusive ownership of `m`.

Effects:	Atomically releases exclusive ownership of <code>m</code> by the current thread and acquires upgrade ownership of <code>m</code> without blocking.
Postcondition:	The current thread has upgrade ownership of <code>m</code> .
Synchronization:	This operation synchronizes with subsequent lock operations that obtain ownership of the same object.
Throws:	Nothing

`m.unlock_upgrade_and_lock()`

Precondition:	The current thread has upgrade ownership of <code>m</code> .
Effects:	Atomically releases upgrade ownership of <code>m</code> by the current thread and acquires exclusive ownership of <code>m</code> . If any other threads have shared ownership, blocks until exclusive ownership can be acquired.
Postcondition:	The current thread has exclusive ownership of <code>m</code> .
Synchronization:	This operation synchronizes with prior <code>unlock_shared()</code> and subsequent lock operations that obtain ownership of the same object.
Throws:	Nothing

`m.try_unlock_upgrade_and_lock()`

Precondition:	The calling thread shall hold a upgrade lock on the mutex.
Effects:	The function attempts to atomically convert the ownership from upgrade to exclusive for the calling thread without blocking. For this conversion to be successful, this thread must be the only thread holding any ownership of the lock. If the conversion is not successful, the upgrade ownership of <code>m</code> is retained.
Returns:	<code>true</code> if exclusive ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has exclusive ownership of <code>m</code> .
Synchronization:	If <code>try_unlock_upgrade_and_lock()</code> returns <code>true</code> , prior <code>unlock()</code> and subsequent lock operations on the same object synchronize with this operation.
Throws:	Nothing
Notes:	Available only if <code>BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN</code> is defined on Windows platform

`m.try_unlock_upgrade_and_lock_for(rel_time)`

Precondition:	The calling thread shall hold a upgrade lock on the mutex.
Effects:	If the tick period of <code>rel_time</code> is not exactly convertible to the native tick period, the duration shall be rounded up to the nearest native tick period. The function attempts to atomically convert the ownership from upgrade to exclusive for the calling thread within the relative timeout specified by <code>rel_time</code> . If the time specified by <code>rel_time</code> is less than or equal to <code>rel_time.zero()</code> , the function attempts to obtain exclusive ownership without blocking (as if by calling <code>try_unlock_upgrade_and_lock()</code>). The function shall return within the timeout specified by <code>rel_time</code> only if it has obtained exclusive ownership of the mutex object. For this conversion to be successful, this thread shall be the only thread holding any ownership of the lock at the moment of conversion. If the conversion is not successful, the upgrade ownership of <code>m</code> is retained.
Returns:	<code>true</code> if exclusive ownership was acquired for the current thread, <code>false</code> otherwise.
Postcondition:	If the call returns <code>true</code> , the current thread has exclusive ownership of <code>m</code> .

Synchronization: If `try_unlock_upgrade_and_lock_for(rel_time)` returns true, prior `unlock()` and subsequent lock operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.try_unlock_upgrade_and_lock_until(abs_time)`

Precondition: The calling thread shall hold a upgrade lock on the mutex.

Effects: The function attempts to atomically convert the ownership from upgrade to exclusive for the calling thread within the absolute timeout specified by `abs_time`. If `abs_time` has already passed, the function attempts to obtain exclusive ownership without blocking (as if by calling `try_unlock_upgrade_and_lock()`). The function shall return before the absolute timeout specified by `abs_time` only if it has obtained exclusive ownership of the mutex object. For this conversion to be successful, this thread shall be the only thread holding any ownership of the lock at the moment of conversion. If the conversion is not successful, the upgrade ownership of `m` is retained.

Returns: `true` if exclusive ownership was acquired for the current thread, `false` otherwise.

Postcondition: If the call returns `true`, the current thread has exclusive ownership of `m`.

Synchronization: If `try_unlock_upgrade_and_lock_for(rel_time)` returns true, prior `unlock()` and subsequent lock operations on the same object synchronize with this operation.

Throws: Nothing

Notes: Available only if `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`m.unlock_upgrade_and_lock_shared()`

Precondition: The current thread has upgrade ownership of `m`.

Effects: Atomically releases upgrade ownership of `m` by the current thread and acquires shared ownership of `m` without blocking.

Postcondition: The current thread has shared ownership of `m`.

Synchronization: This operation synchronizes with prior `unlock_shared()` and subsequent lock operations that obtain ownership of the same object.

Throws: Nothing

Lock Options

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/locks_options.hpp>

namespace boost
{
    struct defer_lock_t {};
    struct try_to_lock_t {};
    struct adopt_lock_t {};
    constexpr defer_lock_t defer_lock;
    constexpr try_to_lock_t try_to_lock;
    constexpr adopt_lock_t adopt_lock;
}
```

Lock option tags

```
#include <boost/thread/locks.hpp>
#include <boost/thread/locks_options.hpp>

struct defer_lock_t {};
struct try_to_lock_t {};
struct adopt_lock_t {};
const defer_lock_t defer_lock;
const try_to_lock_t try_to_lock;
const adopt_lock_t adopt_lock;
```

These tags are used in scoped locks constructors to specify a specific behavior.

- `defer_lock_t`: is used to construct the scoped lock without locking it.
- `try_to_lock_t`: is used to construct the scoped lock trying to lock it.
- `adopt_lock_t`: is used to construct the scoped lock without locking it but adopting ownership.

Lock Guard

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_guard.hpp>

namespace boost
{
    template<typename Lockable>
    class lock_guard
    #if ! defined BOOST_THREAD_NO_MAKE_LOCK_GUARD
    template <typename Lockable>
    lock_guard<Lockable> make_lock_guard(Lockable& mtx); // EXTENSION
    template <typename Lockable>
    lock_guard<Lockable> make_lock_guard(Lockable& mtx, adopt_lock_t); // EXTENSION
    #endif
}
```

Class template `lock_guard`

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_guard.hpp>

template<typename Lockable>
class lock_guard
{
public:
    explicit lock_guard(Lockable& m_);
    lock_guard(Lockable& m_, boost::adopt_lock_t);

    ~lock_guard();
};
```

`boost::lock_guard` is very simple: on construction it acquires ownership of the implementation of the `Lockable` concept supplied as the constructor parameter. On destruction, the ownership is released. This provides simple RAII-style locking of a `Lockable` object, to facilitate exception-safe locking and unlocking. In addition, the `lock_guard(Lockable & m, boost::adopt_lock_t)` constructor allows the `boost::lock_guard` object to take ownership of a lock already held by the current thread.

`lock_guard(Lockable & m)`

Effects: Stores a reference to `m`. Invokes `m.lock()`.

Throws: Any exception thrown by the call to `m.lock()`.

`lock_guard(Lockable & m, boost::adopt_lock_t)`

Precondition: The current thread owns a lock on `m` equivalent to one obtained by a call to `m.lock()`.

Effects: Stores a reference to `m`. Takes ownership of the lock state of `m`.

Throws: Nothing.

`~lock_guard()`

Effects: Invokes `m.unlock()` on the `Lockable` object passed to the constructor.

Throws: Nothing.

Non Member Function `make_lock_guard`

```
template <typename Lockable>
lock_guard<Lockable> make_lock_guard(Lockable& m); // EXTENSION
```

Returns: a `lock_guard` as if initialized with `{m}`.

Throws: Any exception thrown by the call to `m.lock()`.

Non Member Function `make_lock_guard`

```
template <typename Lockable>
lock_guard<Lockable> make_lock_guard(Lockable& m, adopt_lock_t); // EXTENSION
```

Returns: a `lock_guard` as if initialized with `{m, adopt_lock}`.

Throws: Any exception thrown by the call to `m.lock()`.

With Lock Guard

```
// #include <boost/thread/with_lock_guard.hpp>

namespace boost
{
    template <class Lockable, class Function, class... Args>
    auto with_lock_guard(Lockable& m, Function&& func, Args&&... args) -> decltype(func(boost::for_ward<Args>(args)...));
}
```

Non Member Function `with_lock_guard`

```
template <class Lockable, class Function, class... Args>
auto with_lock_guard(
    Lockable& m,
    Function&& func,
    Args&&... args
) -> decltype(func(boost::forward<Args>(args)...));
```

Precondition:	m must be in unlocked state
Effects:	call func in scope locked by m
Returns:	Result of func(args...) call
Throws:	Any exception thrown by the call to m.lock and func(args...)
Postcondition:	m is in unlocked state
Limitations:	Without c++11 variadic templates support number of arguments is limited to 4 Without rvalue references support calling class method with boost::bind must be const For correct work with lambda macro BOOST_RESULT_OF_USE_DECLTYPE may be needed to define

Lock Concepts

StrictLock -- EXTENSION

```
// #include <boost/thread/lock_concepts.hpp>

namespace boost
{
    template<typename Lock>
    class StrictLock;
}
```

A StrictLock is a lock that ensures that the associated mutex is locked during the lifetime of the lock.

A type L meets the StrictLock requirements if the following expressions are well-formed and have the specified semantics

- L::mutex_type
- is_strict_lock<L>
- cl.owns_lock(m);

and BasicLockable<L::mutex_type>

where

- cl denotes a value of type L const&,
- m denotes a value of type L::mutex_type const*,

L::mutex_type

The type L::mutex_type denotes the mutex that is locked by this lock.

`is_strict_lock_sur_parole<L>`

As the semantic "ensures that the associated mutex is locked during the lifetime of the lock. " can not be described by syntactic requirements a `is_strict_lock_sur_parole` trait must be specialized by the user defining the lock so that the following assertion is true:

```
is_strict_lock_sur_parole<L>::value == true
```

```
cl.owns_lock(m);
```

Return Type: `bool`

Returns: Whether the strict lock is locking the mutex `m`

Throws: Nothing.

Models

The following classes are models of `StrictLock`:

- `strict_lock`: ensured by construction,
- `nested_strict_lock`: "sur parole" as the user could use `adopt_lock_t` on `unique_lock` constructor overload without having locked the mutex,
- `boost::lock_guard`: "sur parole" as the user could use `adopt_lock_t` constructor overload without having locked the mutex.

Lock Types

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_types.hpp>

namespace boost
{
    template<typename Lockable>
    class unique_lock;
    template<typename Mutex>
    void swap(unique_lock<Mutex>& lhs, unique_lock<Mutex>& rhs);
    template<typename Lockable>
    class shared_lock; // C++14
    template<typename Mutex>
    void swap(shared_lock<Mutex>& lhs, shared_lock<Mutex>& rhs); // C++14
    template<typename Lockable>
    class upgrade_lock; // EXTENSION
    template<typename Mutex>
    void swap(upgrade_lock<Mutex>& lhs, upgrade_lock<Mutex>& rhs); // EXTENSION
    template<class Mutex>
    class upgrade_to_unique_lock; // EXTENSION
}
```

Class template `unique_lock`

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_types.hpp>

template<typename Lockable>
class unique_lock
{
public:
    typedef Lockable mutex_type;
    unique_lock() noexcept;
    explicit unique_lock(Lockable& m_);
    unique_lock(Lockable& m_, adopt_lock_t);
    unique_lock(Lockable& m_, defer_lock_t) noexcept;
    unique_lock(Lockable& m_, try_to_lock_t);

#ifdef BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION
    unique_lock(shared_lock<mutex_type>&& sl, try_to_lock_t); // C++14
    template <class Clock, class Duration>
    unique_lock(shared_lock<mutex_type>&& sl,
                const chrono::time_point<Clock, Duration>& abs_time); // C++14
    template <class Rep, class Period>
    unique_lock(shared_lock<mutex_type>&& sl,
                const chrono::duration<Rep, Period>& rel_time); // C++14
#endif

    template <class Clock, class Duration>
    unique_lock(Mutex& mtx, const chrono::time_point<Clock, Duration>& t);
    template <class Rep, class Period>
    unique_lock(Mutex& mtx, const chrono::duration<Rep, Period>& d);
    ~unique_lock();

    unique_lock(unique_lock const&) = delete;
    unique_lock& operator=(unique_lock const&) = delete;
    unique_lock(unique_lock<Lockable>&& other) noexcept;
    explicit unique_lock(upgrade_lock<Lockable>&& other) noexcept; // EXTENSION

    unique_lock& operator=(unique_lock<Lockable>&& other) noexcept;

    void swap(unique_lock& other) noexcept;
    Lockable* release() noexcept;

    void lock();
    bool try_lock();

    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

    void unlock();

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

    mutex_type* mutex() const noexcept;

#ifdef defined BOOST_THREAD_USE_DATE_TIME || defined BOOST_THREAD_DONT_USE_CHRONO
    unique_lock(Lockable& m_, system_time const& target_time);
#endif
};
```

```

template<typename TimeDuration>
bool timed_lock(TimeDuration const& relative_time);
bool timed_lock(::boost::system_time const& absolute_time);
#endif
};

```

`boost::unique_lock` is more complex than `boost::lock_guard`: not only does it provide for RAII-style locking, it also allows for deferring acquiring the lock until the `lock()` member function is called explicitly, or trying to acquire the lock in a non-blocking fashion, or with a timeout. Consequently, `unlock()` is only called in the destructor if the lock object has locked the `Lockable` object, or otherwise adopted a lock on the `Lockable` object.

Specializations of `boost::unique_lock` model the `TimedLockable` concept if the supplied `Lockable` type itself models `TimedLockable` concept (e.g. `boost::unique_lock<boost::timed_mutex>`), or the `Lockable` concept if the supplied `Lockable` type itself models `Lockable` concept (e.g. `boost::unique_lock<boost::mutex>`), or the `BasicLockable` concept if the supplied `Lockable` type itself models `BasicLockable` concept.

An instance of `boost::unique_lock` is said to *own* the lock state of a `Lockable` `m` if `mutex()` returns a pointer to `m` and `owns_lock()` returns true. If an object that *owns* the lock state of a `Lockable` object is destroyed, then the destructor will invoke `mutex()->unlock()`.

The member functions of `boost::unique_lock` are not thread-safe. In particular, `boost::unique_lock` is intended to model the ownership of a `Lockable` object by a particular thread, and the member functions that release ownership of the lock state (including the destructor) must be called by the same thread that acquired ownership of the lock state.

`unique_lock()`

Effects: Creates a lock object with no associated mutex.

Postcondition: `owns_lock()` returns false. `mutex()` returns NULL.

Throws: Nothing.

`unique_lock(Lockable & m)`

Effects: Stores a reference to `m`. Invokes `m.lock()`.

Postcondition: `owns_lock()` returns true. `mutex()` returns `&m`.

Throws: Any exception thrown by the call to `m.lock()`.

`unique_lock(Lockable & m, boost::adopt_lock_t)`

Precondition: The current thread owns an exclusive lock on `m`.

Effects: Stores a reference to `m`. Takes ownership of the lock state of `m`.

Postcondition: `owns_lock()` returns true. `mutex()` returns `&m`.

Throws: Nothing.

`unique_lock(Lockable & m, boost::defer_lock_t)`

Effects: Stores a reference to `m`.

Postcondition: `owns_lock()` returns false. `mutex()` returns `&m`.

Throws: Nothing.

`unique_lock(Lockable & m, boost::try_to_lock_t)`

Effects: Stores a reference to `m`. Invokes `m.try_lock()`, and takes ownership of the lock state if the call returns `true`.

Postcondition: `mutex()` returns `&m`. If the call to `try_lock()` returned `true`, then `owns_lock()` returns `true`, otherwise `owns_lock()` returns `false`.

Throws: Nothing.

`unique_lock(shared_lock<mutex_type>&& sl, try_to_lock_t)`

Requires: The supplied Mutex type must implement `try_unlock_shared_and_lock()`.

Effects: Constructs an object of type `boost::unique_lock`. Let `pm` be the pointer to the mutex and `owns` the ownership state. Initializes `pm` with `nullptr` and `owns` with `false`. If `sl.owns_lock()` returns `false`, sets `pm` to the return value of `sl.release()`. Else `sl.owns_lock()` returns `true`, and in this case if `sl.mutex()->try_unlock_shared_and_lock()` returns `true`, sets `pm` to the value returned by `sl.release()` and sets `owns` to `true`.

Note: If `sl.owns_lock()` returns `true` and `sl.mutex()->try_unlock_shared_and_lock()` returns `false`, `sl` is not modified.

Throws: Nothing.

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`unique_lock(shared_lock<mutex_type>&&, const chrono::time_point<Clock, Duration>&)`

```
template <class Clock, class Duration>
unique_lock(shared_lock<mutex_type>&& sl,
            const chrono::time_point<Clock, Duration>& abs_time);
```

Requires: The supplied Mutex type shall implement `try_unlock_shared_and_lock_until(abs_time)`.

Effects: Constructs an object of type `boost::unique_lock`, initializing `pm` with `nullptr` and `owns` with `false`. If `sl.owns_lock()` returns `false`, sets `pm` to the return value of `sl.release()`. Else `sl.owns_lock()` returns `true`, and in this case if `sl.mutex()->try_unlock_shared_and_lock_until(abs_time)` returns `true`, sets `pm` to the value returned by `sl.release()` and sets `owns` to `true`.

Note: If `sl.owns_lock()` returns `true` and `sl.mutex()->try_unlock_shared_and_lock_until(abs_time)` returns `false`, `sl` is not modified.

Throws: Nothing.

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`unique_lock(shared_lock<mutex_type>&&, const chrono::duration<Rep, Period>&)`

```
template <class Rep, class Period>
unique_lock(shared_lock<mutex_type>&& sl,
            const chrono::duration<Rep, Period>& rel_time)
```

Requires: The supplied Mutex type shall implement `try_unlock_shared_and_lock_for(rel_time)`.

Effects: Constructs an object of type `boost::unique_lock`, initializing `pm` with `nullptr` and `owns` with `false`. If `sl.owns_lock()` returns `false`, sets `pm` to the return value of `sl.release()`. Else `sl.owns_lock()`

returns true, and in this case if `sl.mutex()->try_unlock_shared_and_lock_for(rel_time)` returns true, sets `pm` to the value returned by `sl.release()` and sets `owns` to true.

Note: If `sl.owns_lock()` returns true and `sl.mutex()->try_unlock_shared_and_lock_for(rel_time)` returns false, `sl` is not modified.

Postcondition: .

Throws: Nothing.

Notes: Available only if `BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION` and `BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN` is defined on Windows platform

`unique_lock(Lockable & m, boost::system_time const& abs_time)`

Effects: Stores a reference to `m`. Invokes `m.timed_lock(abs_time)`, and takes ownership of the lock state if the call returns true.

Postcondition: `mutex()` returns `&m`. If the call to `timed_lock()` returned true, then `owns_lock()` returns true, otherwise `owns_lock()` returns false.

Throws: Any exceptions thrown by the call to `m.timed_lock(abs_time)`.

`template <class Clock, class Duration> unique_lock(Lockable & m, const chrono::time_point<Clock, Duration>& abs_time)`

Effects: Stores a reference to `m`. Invokes `m.try_lock_until(abs_time)`, and takes ownership of the lock state if the call returns true.

Postcondition: `mutex()` returns `&m`. If the call to `try_lock_until` returned true, then `owns_lock()` returns true, otherwise `owns_lock()` returns false.

Throws: Any exceptions thrown by the call to `m.try_lock_until(abs_time)`.

`template <class Rep, class Period> unique_lock(Lockable & m, const chrono::duration<Rep, Period>& abs_time)`

Effects: Stores a reference to `m`. Invokes `m.try_lock_for(rel_time)`, and takes ownership of the lock state if the call returns true.

Postcondition: `mutex()` returns `&m`. If the call to `try_lock_for` returned true, then `owns_lock()` returns true, otherwise `owns_lock()` returns false.

Throws: Any exceptions thrown by the call to `m.try_lock_for(rel_time)`.

`~unique_lock()`

Effects: Invokes `mutex()->unlock()` if `owns_lock()` returns true.

Throws: Nothing.

`bool owns_lock() const`

Returns: true if the `*this` owns the lock on the `Lockable` object associated with `*this`.

Throws: Nothing.

Lockable* mutex() const noexcept

Returns: A pointer to the **Lockable** object associated with `*this`, or `NULL` if there is no such object.

Throws: Nothing.

explicit operator bool() const

Returns: `owns_lock()`.

Throws: Nothing.

Lockable* release()

Effects: The association between `*this` and the **Lockable** object is removed, without affecting the lock state of the **Lockable** object. If `owns_lock()` would have returned `true`, it is the responsibility of the calling code to ensure that the **Lockable** is correctly unlocked.

Returns: A pointer to the **Lockable** object associated with `*this` at the point of the call, or `NULL` if there is no such object.

Throws: Nothing.

Postcondition: `*this` is no longer associated with any **Lockable** object. `mutex()` returns `NULL` and `owns_lock()` returns `false`.

Class template `shared_lock` - C++14

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_types.hpp>

template<typename Lockable>
class shared_lock
{
public:
    typedef Lockable mutex_type;

    // Shared locking
    shared_lock();
    explicit shared_lock(Lockable& m_);
    shared_lock(Lockable& m_, adopt_lock_t);
    shared_lock(Lockable& m_, defer_lock_t);
    shared_lock(Lockable& m_, try_to_lock_t);
    template <class Clock, class Duration>
    shared_lock(Mutex& mtx, const chrono::time_point<Clock, Duration>& t);
    template <class Rep, class Period>
    shared_lock(Mutex& mtx, const chrono::duration<Rep, Period>& d);
    ~shared_lock();

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

    shared_lock(shared_lock<Lockable> && other);
    shared_lock& operator=(shared_lock<Lockable> && other);

    void lock();
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    // Conversion from upgrade locking
    explicit shared_lock(upgrade_lock<Lockable> && other); // EXTENSION

    // Conversion from exclusive locking
    explicit shared_lock(unique_lock<Lockable> && other);

    // Setters
    void swap(shared_lock& other);
    mutex_type* release() noexcept;

    // Getters
    explicit operator bool() const;
    bool owns_lock() const;
    mutex_type mutex() const;

#if defined BOOST_THREAD_USE_DATE_TIME || defined BOOST_THREAD_DONT_USE_CHRONO
    shared_lock(Lockable& m_, system_time const& target_time);
    bool timed_lock(boost::system_time const& target_time);
#endif
};
```

Like `boost::unique_lock`, `boost::shared_lock` models the `Lockable` concept, but rather than acquiring unique ownership of the supplied `Lockable` object, locking an instance of `boost::shared_lock` acquires shared ownership.

Like `boost::unique_lock`, not only does it provide for RAII-style locking, it also allows for deferring acquiring the lock until the `lock()` member function is called explicitly, or trying to acquire the lock in a non-blocking fashion, or with a timeout. Consequently, `unlock()` is only called in the destructor if the lock object has locked the `Lockable` object, or otherwise adopted a lock on the `Lockable` object.

An instance of `boost::shared_lock` is said to *own* the lock state of a `Lockable` `m` if `mutex()` returns a pointer to `m` and `owns_lock()` returns `true`. If an object that *owns* the lock state of a `Lockable` object is destroyed, then the destructor will invoke `mutex()->unlock_shared()`.

The member functions of `boost::shared_lock` are not thread-safe. In particular, `boost::shared_lock` is intended to model the shared ownership of a `Lockable` object by a particular thread, and the member functions that release ownership of the lock state (including the destructor) must be called by the same thread that acquired ownership of the lock state.

`shared_lock()`

Effects: Creates a lock object with no associated mutex.

Postcondition: `owns_lock()` returns `false`. `mutex()` returns `NULL`.

Throws: Nothing.

`shared_lock(Lockable & m)`

Effects: Stores a reference to `m`. Invokes `m.lock_shared()`.

Postcondition: `owns_lock()` returns `true`. `mutex()` returns `&m`.

Throws: Any exception thrown by the call to `m.lock_shared()`.

`shared_lock(Lockable & m, boost::adopt_lock_t)`

Precondition: The current thread owns an exclusive lock on `m`.

Effects: Stores a reference to `m`. Takes ownership of the lock state of `m`.

Postcondition: `owns_lock()` returns `true`. `mutex()` returns `&m`.

Throws: Nothing.

`shared_lock(Lockable & m, boost::defer_lock_t)`

Effects: Stores a reference to `m`.

Postcondition: `owns_lock()` returns `false`. `mutex()` returns `&m`.

Throws: Nothing.

`shared_lock(Lockable & m, boost::try_to_lock_t)`

Effects: Stores a reference to `m`. Invokes `m.try_lock_shared()`, and takes ownership of the lock state if the call returns `true`.

Postcondition: `mutex()` returns `&m`. If the call to `try_lock_shared()` returned `true`, then `owns_lock()` returns `true`, otherwise `owns_lock()` returns `false`.

Throws: Nothing.

shared_lock(Lockable & m, boost::system_time const& abs_time)

Effects: Stores a reference to m. Invokes `m.timed_lock(abs_time)`, and takes ownership of the lock state if the call returns true.

Postcondition: `mutex()` returns &m. If the call to `timed_lock_shared()` returned true, then `owns_lock()` returns true, otherwise `owns_lock()` returns false.

Throws: Any exceptions thrown by the call to `m.timed_lock(abs_time)`.

~shared_lock()

Effects: Invokes `mutex()->unlock_shared()` if `owns_lock()` returns true.

Throws: Nothing.

bool owns_lock() const

Returns: true if the *this owns the lock on the `Lockable` object associated with *this.

Throws: Nothing.

Lockable* mutex() const

Returns: A pointer to the `Lockable` object associated with *this, or NULL if there is no such object.

Throws: Nothing.

explicit operator bool() const

Returns: `owns_lock()`.

Throws: Nothing.

Lockable* release()

Effects: The association between *this and the `Lockable` object is removed, without affecting the lock state of the `Lockable` object. If `owns_lock()` would have returned true, it is the responsibility of the calling code to ensure that the `Lockable` is correctly unlocked.

Returns: A pointer to the `Lockable` object associated with *this at the point of the call, or NULL if there is no such object.

Throws: Nothing.

Postcondition: *this is no longer associated with any `Lockable` object. `mutex()` returns NULL and `owns_lock()` returns false.

Class template `upgrade_lock` - EXTENSION

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_types.hpp>

template<typename Lockable>
class upgrade_lock
{
public:
    typedef Lockable mutex_type;

    // Upgrade locking

    upgrade_lock();
    explicit upgrade_lock(mutex_type& m_);
    upgrade_lock(mutex_type& m, defer_lock_t) noexcept;
    upgrade_lock(mutex_type& m, try_to_lock_t);
    upgrade_lock(mutex_type& m, adopt_lock_t);
    template <class Clock, class Duration>
    upgrade_lock(mutex_type& m,
        const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
    upgrade_lock(mutex_type& m,
        const chrono::duration<Rep, Period>& rel_time);
    ~upgrade_lock();

    upgrade_lock(const upgrade_lock& other) = delete;
    upgrade_lock& operator=(const upgrade_lock<Lockable> & other) = delete;

    upgrade_lock(upgrade_lock<Lockable> && other);
    upgrade_lock& operator=(upgrade_lock<Lockable> && other);

    void lock();
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

#ifdef BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSION
    // Conversion from shared locking
    upgrade_lock(shared_lock<mutex_type>&& sl, try_to_lock_t);
    template <class Clock, class Duration>
    upgrade_lock(shared_lock<mutex_type>&& sl,
        const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
    upgrade_lock(shared_lock<mutex_type>&& sl,
        const chrono::duration<Rep, Period>& rel_time);
#endif

    // Conversion from exclusive locking
    explicit upgrade_lock(unique_lock<Lockable> && other);

    // Setters
    void swap(upgrade_lock& other);
    mutex_type* release() noexcept;

    // Getters
    explicit operator bool() const;
    bool owns_lock() const;
    mutex_type mutex() const;
};
```

Like `boost::unique_lock`, `boost::upgrade_lock` models the `Lockable` concept, but rather than acquiring unique ownership of the supplied `Lockable` object, locking an instance of `boost::upgrade_lock` acquires upgrade ownership.

Like `boost::unique_lock`, not only does it provide for RAII-style locking, it also allows for deferring acquiring the lock until the `lock()` member function is called explicitly, or trying to acquire the lock in a non-blocking fashion, or with a timeout. Consequently, `unlock()` is only called in the destructor if the lock object has locked the `Lockable` object, or otherwise adopted a lock on the `Lockable` object.

An instance of `boost::upgrade_lock` is said to *own* the lock state of a `Lockable` `m` if `mutex()` returns a pointer to `m` and `owns_lock()` returns `true`. If an object that *owns* the lock state of a `Lockable` object is destroyed, then the destructor will invoke `mutex()->unlock_upgrade()`.

The member functions of `boost::upgrade_lock` are not thread-safe. In particular, `boost::upgrade_lock` is intended to model the upgrade ownership of a `UpgradeLockable` object by a particular thread, and the member functions that release ownership of the lock state (including the destructor) must be called by the same thread that acquired ownership of the lock state.

Class template `upgrade_to_unique_lock` -- EXTENSION

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_types.hpp>

template <class Lockable>
class upgrade_to_unique_lock
{
public:
    typedef Lockable mutex_type;
    explicit upgrade_to_unique_lock(upgrade_lock<Lockable>& m_);
    ~upgrade_to_unique_lock();

    upgrade_to_unique_lock(upgrade_to_unique_lock const& other) = delete;
    upgrade_to_unique_lock& operator=(upgrade_to_unique_lock<Lockable> const& other) = delete;

    upgrade_to_unique_lock(upgrade_to_unique_lock<Lockable> && other);
    upgrade_to_unique_lock& operator=(upgrade_to_unique_lock<Lockable> && other);

    void swap(upgrade_to_unique_lock& other);

    explicit operator bool() const;
    bool owns_lock() const;
    mutex_type* mutex() const;

};
```

`boost::upgrade_to_unique_lock` allows for a temporary upgrade of an `boost::upgrade_lock` to exclusive ownership. When constructed with a reference to an instance of `boost::upgrade_lock`, if that instance has upgrade ownership on some `Lockable` object, that ownership is upgraded to exclusive ownership. When the `boost::upgrade_to_unique_lock` instance is destroyed, the ownership of the `Lockable` is downgraded back to *upgrade ownership*.

Mutex-specific class `scoped_try_lock` -- DEPRECATED

```
class MutexType::scoped_try_lock
{
private:
    MutexType::scoped_try_lock(MutexType::scoped_try_lock<MutexType>& other);
    MutexType::scoped_try_lock& operator=(MutexType::scoped_try_lock<MutexType>& other);
public:
    MutexType::scoped_try_lock();
    explicit MutexType::scoped_try_lock(MutexType& m);
    MutexType::scoped_try_lock(MutexType& m_, adopt_lock_t);
    MutexType::scoped_try_lock(MutexType& m_, defer_lock_t);
    MutexType::scoped_try_lock(MutexType& m_, try_to_lock_t);

    MutexType::scoped_try_lock(MutexType::scoped_try_lock<MutexType>&& other);
    MutexType::scoped_try_lock& operator=(MutexType::scoped_try_lock<MutexType>&& other);

    void swap(MutexType::scoped_try_lock&& other);

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

    MutexType* mutex() const;
    MutexType* release();

    explicit operator bool() const;
    bool owns_lock() const;
};
```

The member typedef `scoped_try_lock` is provided for each distinct `MutexType` as a typedef to a class with the preceding definition. The semantics of each constructor and member function are identical to those of `boost::unique_lock<MutexType>` for the same `MutexType`, except that the constructor that takes a single reference to a mutex will call `m.try_lock()` rather than `m.lock()`.

Other Lock Types - EXTENSION

Strict Locks

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/strict_lock.hpp>

namespace boost
{
    template<typename Lockable>
    class strict_lock;
    template <typename Lock>
    class nested_strict_lock;
    template <typename Lockable>
    struct is_strict_lock_sur_parole<strict_lock<Lockable> >;
    template <typename Lock>
    struct is_strict_lock_sur_parole<nested_strict_lock<Lock> >;

#if ! defined BOOST_THREAD_NO_MAKE_STRICT_LOCK
    template <typename Lockable>
    strict_lock<Lockable> make_strict_lock(Lockable& mtx);
#endif
#if ! defined BOOST_THREAD_NO_MAKE_NESTED_STRICT_LOCK
    template <typename Lock>
    nested_strict_lock<Lock> make_nested_strict_lock(Lock& lk);
#endif
}
```

Class template `strict_lock`

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/strict_lock.hpp>

template<typename BasicLockable>
class strict_lock
{
public:
    typedef BasicLockable mutex_type;
    strict_lock(strict_lock const& m_) = delete;
    strict_lock& operator=(strict_lock const& m_) = delete;
    explicit strict_lock(mutex_type& m_);
    ~strict_lock();

    bool owns_lock(mutex_type const* l) const noexcept;
};
```

`strict_lock` is a model of `StrictLock`.

`strict_lock` is the simplest `StrictLock`: on construction it acquires ownership of the implementation of the `BasicLockable` concept supplied as the constructor parameter. On destruction, the ownership is released. This provides simple RAII-style locking of a `BasicLockable` object, to facilitate exception-safe locking and unlocking.

See also `boost::lock_guard`

`strict_lock(Lockable & m)`

Effects: Stores a reference to `m`. Invokes `m.lock()`.

Throws: Any exception thrown by the call to `m.lock()`.

`~strict_lock()`

Effects: Invokes `m.unlock()` on the `Lockable` object passed to the constructor.

Throws: Nothing.

Class template `nested_strict_lock`

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/strict_lock.hpp>

template<typename Lock>
class nested_strict_lock
{
public:
    typedef BasicLockable mutex_type;
    nested_strict_lock(nested_strict_lock const& m_) = delete;
    nested_strict_lock& operator=(nested_strict_lock const& m_) = delete;
    explicit nested_strict_lock(Lock& lk),
    ~nested_strict_lock() noexcept;

    bool owns_lock(mutex_type const* l) const noexcept;
};
```

`nested_strict_lock` is a model of `StrictLock`.

A nested strict lock is a scoped lock guard ensuring a mutex is locked on its scope, by taking ownership of an nesting lock, locking the mutex on construction if not already locked and restoring the ownership to the nesting lock on destruction.

See also `strict_lock`, `boost::unique_lock`

`nested_strict_lock(Lock & lk)`

Requires: `lk.mutex() != null_ptr`.

Effects: Stores the reference to the lock parameter `lk` and takes ownership on it. If the lock doesn't owns the mutex lock it.

Postcondition: `owns_lock(lk.mutex())`.

Throws: - `lock_error` when `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and `lk.mutex() == null_ptr`

- Any exception that `@c lk.lock()` can throw.

`~nested_strict_lock() noexcept`

Effects: Restores ownership to the nesting lock.

`bool owns_lock(mutex_type const* l) const noexcept`

Return: Whether if this lock is locking that mutex.

Non Member Function `make_strict_lock`

```
template <typename Lockable>
strict_lock<Lockable> make_strict_lock(Lockable& m); // EXTENSION
```

Returns: a `strict_lock` as if initialized with `{m}`.

Throws: Any exception thrown by the call to `m.lock()`.

Non Member Function `make_nested_strict_lock`

```
template <typename Lock>
nested_strict_lock<Lock> make_nested_strict_lock(Lock& lk); // EXTENSION
```

Returns: a `nested_strict_lock` as if initialized with `{lk}`.

Throws: Any exception thrown by the call to `lk.lock()`.

Locking pointers

```
// #include <boost/thread/synchroniezd_value.hpp>
// #include <boost/thread/strict_lock_ptr.hpp>

namespace boost
{
    template<typename T, typename Lockable = mutex>
    class strict_lock_ptr;
    template<typename T, typename Lockable = mutex>
    class const_strict_lock_ptr;
}
```

Class template `const_strict_lock_ptr`

```
// #include <boost/thread/synchroniezd_value.hpp>
// #include <boost/thread/strict_lock_ptr.hpp>

template <typename T, typename Lockable = mutex>
class const_strict_lock_ptr
{
public:
    typedef T value_type;
    typedef Lockable mutex_type;

    const_strict_lock_ptr(const_strict_lock_ptr const& m_) = delete;
    const_strict_lock_ptr& operator=(const_strict_lock_ptr const& m_) = delete;

    const_strict_lock_ptr(T const& val, Lockable & mtx);
    const_strict_lock_ptr(T const& val, Lockable & mtx, adopt_lock_t tag);

    ~const_strict_lock_ptr();

    const T* operator->() const;
    const T& operator*() const;
};
```

const_strict_lock_ptr(T const&, Lockable&)

```
const_strict_lock_ptr(T const& val, Lockable & m);
```

Effects: Invokes `m.lock()`, stores a reference to it and to the value type `val`.

Throws: Any exception thrown by the call to `m.lock()`.

const_strict_lock_ptr(T const&, Lockable&, adopt_lock_t)

```
const_strict_lock_ptr(T const& val, Lockable & m, adopt_lock_t tag);
```

Effects: Stores a reference to it and to the value type `val`.

Throws: Nothing.

~const_strict_lock_ptr()

```
~const_strict_lock_ptr();
```

Effects: Invokes `m.unlock()` on the `Lockable` object passed to the constructor.

Throws: Nothing.

operator->() const

```
const T* operator->() const;
```

Return: return a constant pointer to the protected value.

Throws: Nothing.

operator*() const

```
const T& operator*() const;
```

Return: return a constant reference to the protected value.

Throws: Nothing.

Class template `strict_lock_ptr`

```
// #include <boost/thread/synchroniezd_value.hpp>
// #include <boost/thread/strict_lock_ptr.hpp>

template <typename T, typename Lockable = mutex>
class strict_lock_ptr : public const_strict_lock_ptr<T, Lockable>
{
public:
    strict_lock_ptr(strict_lock_ptr const& m_) = delete;
    strict_lock_ptr& operator=(strict_lock_ptr const& m_) = delete;

    strict_lock_ptr(T & val, Lockable & mtx);
    strict_lock_ptr(T & val, Lockable & mtx, adopt_lock_t tag);
    ~strict_lock_ptr();

    T* operator->();
    T& operator*();

};
```

`strict_lock_ptr(T const&, Lockable&)`

```
strict_lock_ptr(T const& val, Lockable & m);
```

Effects: Invokes `m.lock()`, stores a reference to it and to the value type `val`.

Throws: Any exception thrown by the call to `m.lock()`.

`strict_lock_ptr(T const&, Lockable&, adopt_lock_t)`

```
strict_lock_ptr(T const& val, Lockable & m, adopt_lock_t tag);
```

Effects: Stores a reference to it and to the value type `val`.

Throws: Nothing.

`~strict_lock_ptr()`

```
~ strict_lock_ptr();
```

Effects: Invokes `m.unlock()` on the `Lockable` object passed to the constructor.

Throws: Nothing.

`operator->()`

```
T* operator->();
```

Return: return a pointer to the protected value.

Throws: Nothing.

operator*()

```
T& operator*();
```

Return: return a reference to the protected value.

Throws: Nothing.

Externally Locked

```
// #include <boost/thread/externally_locked.hpp>
template <class T, typename MutexType = boost::mutex>
class externally_locked;
template <class T, typename MutexType>
class externally_locked<T&, MutexType>;

template <typename T, typename MutexType>
void swap(externally_locked<T, MutexType> & lhs, externally_locked<T, MutexType> & rhs);
```

Template Class `externally_locked`

```
// #include <boost/thread/externally_locked.hpp>

template <class T, typename MutexType>
class externally_locked
{
    //BOOST_CONCEPT_ASSERT(( CopyConstructible<T> ));
    BOOST_CONCEPT_ASSERT(( BasicLockable<MutexType> ));

public:
    typedef MutexType mutex_type;

    externally_locked(mutex_type& mtx, const T& obj);
    externally_locked(mutex_type& mtx, T&& obj);
    explicit externally_locked(mutex_type& mtx);
    externally_locked(externally_locked const& rhs);
    externally_locked(externally_locked&& rhs);
    externally_locked& operator=(externally_locked const& rhs);
    externally_locked& operator=(externally_locked&& rhs);

    // observers
    T& get(strict_lock<mutex_type>& lk);
    const T& get(strict_lock<mutex_type>& lk) const;

    template <class Lock>
    T& get(nested_strict_lock<Lock>& lk);
    template <class Lock>
    const T& get(nested_strict_lock<Lock>& lk) const;

    template <class Lock>
    T& get(Lock& lk);
    template <class Lock>
    T const& get(Lock& lk) const;

    mutex_type* mutex() const noexcept;

    // modifiers
    void lock();
    void unlock();
    bool try_lock();
    void swap(externally_locked&);
};
```

`externally_locked` is a model of `Lockable`, it cloaks an object of type `T`, and actually provides full access to that object through the `get` and `set` member functions, provided you pass a reference to a strict lock object.

Only the specificities respect to `Lockable` are described here.

`externally_locked(mutex_type&, const T&)`

```
externally_locked(mutex_type& mtx, const T& obj);
```

Requires: `T` is a model of `CopyConstructible`.

Effects: Constructs an externally locked object copying the cloaked type.

Throws: Any exception thrown by the call to `T(obj)`.

externally_locked(mutex_type&, T&&)

```
externally_locked(mutex_type& mtx, T&& obj);
```

Requires: T is a model of Movable.

Effects: Constructs an externally locked object by moving the cloaked type.

Throws: Any exception thrown by the call to `T(obj)`.

externally_locked(mutex_type&)

```
externally_locked(mutex_type& mtx);
```

Requires: T is a model of DefaultConstructible.

Effects: Constructs an externally locked object by default constructing the cloaked type.

Throws: Any exception thrown by the call to `T()`.

externally_locked(externally_locked&&)

```
externally_locked(externally_locked&& rhs);
```

Requires: T is a model of Movable.

Effects: Move constructs an externally locked object by moving the cloaked type and copying the mutex reference

Throws: Any exception thrown by the call to `T(T&&)`.

externally_locked(externally_locked&)

```
externally_locked(externally_locked& rhs);
```

Requires: T is a model of Copyable.

Effects: Copy constructs an externally locked object by copying the cloaked type and copying the mutex reference

Throws: Any exception thrown by the call to `T(T&)`.

externally_locked(externally_locked&&)

```
externally_locked& operator=(externally_locked&& rhs);
```

Requires: T is a model of Movable.

Effects: Move assigns an externally locked object by moving the cloaked type and copying the mutex reference

Throws: Any exception thrown by the call to `T::operator=(T&&)`.

externally_locked(externally_locked&)

```
externally_locked& operator=(externally_locked const& rhs);
```

Requires: T is a model of Copyable.

Effects: Copy assigns an externally locked object by copying the cloaked type and copying the mutex reference

Throws: Any exception thrown by the call to `T::operator=(T&)`.

`get(strict_lock<mutex_type>&)`

```
T& get(strict_lock<mutex_type>& lk);  
const T& get(strict_lock<mutex_type>& lk) const;
```

Requires: The `lk` parameter must be locking the associated mutex.

Returns: A reference to the cloaked object

Throws: `lock_error` if `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and the run-time preconditions are not satisfied.

`get(strict_lock<nested_strict_lock<Lock>>&)`

```
template <class Lock>  
T& get(nested_strict_lock<Lock>& lk);  
template <class Lock>  
const T& get(nested_strict_lock<Lock>& lk) const;
```

Requires: `is_same<mutex_type, typename Lock::mutex_type>` and the `lk` parameter must be locking the associated mutex.

Returns: A reference to the cloaked object

Throws: `lock_error` if `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and the run-time preconditions are not satisfied.

`get(strict_lock<nested_strict_lock<Lock>>&)`

```
template <class Lock>  
T& get(Lock& lk);  
template <class Lock>  
T const& get(Lock& lk) const;
```

Requires: `Lock` is a model of `StrictLock`, `is_same<mutex_type, typename Lock::mutex_type>` and the `lk` parameter must be locking the associated mutex.

Returns: A reference to the cloaked object

Throws: `lock_error` if `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and the run-time preconditions are not satisfied.

Template Class `externally_locked<T&>`

```
// #include <boost/thread/externally_locked.hpp>

template <class T, typename MutexType>
class externally_locked<T&, MutexType>
{
    //BOOST_CONCEPT_ASSERT(( CopyConstructible<T> ));
    BOOST_CONCEPT_ASSERT(( BasicLockable<MutexType> ));

public:
    typedef MutexType mutex_type;

    externally_locked(mutex_type& mtx, T& obj);
    explicit externally_locked(mutex_type& mtx);
    externally_locked(externally_locked const& rhs) noexcept;
    externally_locked(externally_locked&& rhs) noexcept;
    externally_locked& operator=(externally_locked const& rhs) noexcept;
    externally_locked& operator=(externally_locked&& rhs) noexcept;

    // observers
    T& get(strict_lock<mutex_type>& lk);
    const T& get(strict_lock<mutex_type>& lk) const;

    template <class Lock>
    T& get(nested_strict_lock<Lock>& lk);
    template <class Lock>
    const T& get(nested_strict_lock<Lock>& lk) const;

    template <class Lock>
    T& get(Lock& lk);
    template <class Lock>
    T const& get(Lock& lk) const;

    mutex_type* mutex() const noexcept;

    // modifiers
    void lock();
    void unlock();
    bool try_lock();
    void swap(externally_locked&) noexcept;
};
```

`externally_locked` is a model of `Lockable`, it cloaks an object of type `T`, and actually provides full access to that object through the `get` and `set` member functions, provided you pass a reference to a strict lock object.

Only the specificities respect to `Lockable` are described here.

`externally_locked<T&>(mutex_type&, T&)`

```
externally_locked<T&>(mutex_type& mtx, T& obj) noexcept;
```

Effects: Constructs an externally locked object copying the cloaked reference.

`externally_locked<T&>(externally_locked&&)`

```
externally_locked(externally_locked&& rhs) noexcept;
```

Effects: Moves an externally locked object by moving the cloaked type and copying the mutex reference

externally_locked(externally_locked&&)

```
externally_locked& operator=(externally_locked&& rhs);
```

Effects: Move assigns an externally locked object by copying the cloaked reference and copying the mutex reference

externally_locked(externally_locked&)

```
externally_locked& operator=(externally_locked const& rhs);
```

Requires: T is a model of Copyable.

Effects: Copy assigns an externally locked object by copying the cloaked reference and copying the mutex reference

Throws: Any exception thrown by the call to `T::operator=(T&)`.

get(strict_lock<mutex_type>&)

```
T& get(strict_lock<mutex_type>& lk);
const T& get(strict_lock<mutex_type>& lk) const;
```

Requires: The `lk` parameter must be locking the associated mutex.

Returns: A reference to the cloaked object

Throws: `lock_error` if `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and the run-time preconditions are not satisfied.

get(strict_lock<nested_strict_lock<Lock>>&)

```
template <class Lock>
T& get(nested_strict_lock<Lock>& lk);
template <class Lock>
const T& get(nested_strict_lock<Lock>& lk) const;
```

Requires: `is_same<mutex_type, typename Lock::mutex_type>` and the `lk` parameter must be locking the associated mutex.

Returns: A reference to the cloaked object

Throws: `lock_error` if `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and the run-time preconditions are not satisfied.

get(strict_lock<nested_strict_lock<Lock>>&)

```
template <class Lock>
T& get(Lock& lk);
template <class Lock>
T const& get(Lock& lk) const;
```

Requires: `Lock` is a model of `StrictLock`, `is_same<mutex_type, typename Lock::mutex_type>` and the `lk` parameter must be locking the associated mutex.

Returns: A reference to the cloaked object

Throws: `lock_error` if `BOOST_THREAD_THROW_IF_PRECONDITION_NOT_SATISFIED` is defined and the run-time preconditions are not satisfied.

`swap(externally_locked&, externally_locked&)`

```
template <typename T, typename MutexType>
void swap(externally_locked<T, MutexType> & lhs, externally_locked<T, MutexType> & rhs)
```

Class template `shared_lock_guard`

```
// #include <boost/thread/shared_lock_guard.hpp>
namespace boost
{
    template<typename SharedLockable>
    class shared_lock_guard
    {
    public:
        shared_lock_guard(shared_lock_guard const&) = delete;
        shared_lock_guard& operator=(shared_lock_guard const&) = delete;

        explicit shared_lock_guard(SharedLockable& m_);
        shared_lock_guard(SharedLockable& m_, boost::adopt_lock_t);

        ~shared_lock_guard();
    };
}
```

`shared_lock_guard` is very simple: on construction it acquires shared ownership of the implementation of the `SharedLockable` concept supplied as the constructor parameter. On destruction, the ownership is released. This provides simple RAII-style locking of a `SharedLockable` object, to facilitate exception-safe shared locking and unlocking. In addition, the `shared_lock_guard(SharedLockable &m, boost::adopt_lock_t)` constructor allows the `shared_lock_guard` object to take shared ownership of a lock already held by the current thread.

`shared_lock_guard(SharedLockable & m)`

Effects: Stores a reference to `m`. Invokes `m.lock_shared()`.

Throws: Any exception thrown by the call to `m.lock_shared()`.

`shared_lock_guard(SharedLockable & m, boost::adopt_lock_t)`

Precondition: The current thread owns a lock on `m` equivalent to one obtained by a call to `m.lock_shared()`.

Effects: Stores a reference to `m`. Takes ownership of the lock state of `m`.

Throws: Nothing.

`~shared_lock_guard()`

Effects: Invokes `m.unlock_shared()` on the `SharedLockable` object passed to the constructor.

Throws: Nothing.

Class template `reverse_lock`

```
// #include <boost/thread/reverse_lock.hpp>
namespace boost
{
    template<typename Lock>
    class reverse_lock
    {
    public:
        reverse_lock(reverse_lock const&) = delete;
        reverse_lock& operator=(reverse_lock const&) = delete;

        explicit reverse_lock(Lock& m_);
        ~reverse_lock();
    };
}
```

`reverse_lock` reverse the operations of a lock: it provide for RAII-style, that unlocks the lock at construction time and lock it at destruction time. In addition, it transfer ownership temporarily, so that the mutex can not be locked using the Lock.

An instance of `reverse_lock` doesn't *own* the lock never.

`reverse_lock(Lock & m)`

Effects: Stores a reference to m. Invokes m.`unlock()` if m owns his lock and then stores the mutex by calling m.`release()`.

Postcondition: `!m.owns_lock()` && m.`mutex()==0`.

Throws: Any exception thrown by the call to m.`unlock()`.

`~reverse_lock()`

Effects: Let be mtx the stored mutex*. If not 0 Invokes `mtx->lock()` and gives again the mtx to the Lock using the `adopt_lock_t` overload.

Throws: Any exception thrown by `mtx->lock()`.

Remarks: Note that if `mtx->lock()` throws an exception while unwinding the program will terminate, so don't use `reverse_lock` if an exception can be thrown.

Lock functions

Non-member function `lock(Lockable1, Lockable2, ...)`

```
// #include <boost/thread/locks.hpp>
// #include <boost/thread/lock_algorithms.hpp>
namespace boost
{

    template<typename Lockable1, typename Lockable2>
    void lock(Lockable1& l1, Lockable2& l2);

    template<typename Lockable1, typename Lockable2, typename Lockable3>
    void lock(Lockable1& l1, Lockable2& l2, Lockable3& l3);

    template<typename Lockable1, typename Lockable2, typename Lockable3, typename Lockable4>
    void lock(Lockable1& l1, Lockable2& l2, Lockable3& l3, Lockable4& l4);

    template<typename Lockable1, typename Lockable2, typename Lockable3, typename Lockable4, typename Lockable5>
    void lock(Lockable1& l1, Lockable2& l2, Lockable3& l3, Lockable4& l4, Lockable5& l5);

}
```

Effects: Locks the `Lockable` objects supplied as arguments in an unspecified and indeterminate order in a way that avoids deadlock. It is safe to call this function concurrently from multiple threads with the same mutexes (or other lockable objects) in different orders without risk of deadlock. If any of the `lock()` or `try_lock()` operations on the supplied `Lockable` objects throws an exception any locks acquired by the function will be released before the function exits.

Throws: Any exceptions thrown by calling `lock()` or `try_lock()` on the supplied `Lockable` objects.

Postcondition: All the supplied `Lockable` objects are locked by the calling thread.

Non-member function `lock(begin, end) // EXTENSION`

```
template<typename ForwardIterator>
void lock(ForwardIterator begin, ForwardIterator end);
```

Preconditions: The `value_type` of `ForwardIterator` must implement the `Lockable` concept

Effects: Locks all the `Lockable` objects in the supplied range in an unspecified and indeterminate order in a way that avoids deadlock. It is safe to call this function concurrently from multiple threads with the same mutexes (or other lockable objects) in different orders without risk of deadlock. If any of the `lock()` or `try_lock()` operations on the `Lockable` objects in the supplied range throws an exception any locks acquired by the function will be released before the function exits.

Throws: Any exceptions thrown by calling `lock()` or `try_lock()` on the supplied `Lockable` objects.

Postcondition: All the `Lockable` objects in the supplied range are locked by the calling thread.

Non-member function `try_lock(Lockable1, Lockable2, ...)`

```
template<typename Lockable1, typename Lockable2>
int try_lock(Lockable1& l1, Lockable2& l2);

template<typename Lockable1, typename Lockable2, typename Lockable3>
int try_lock(Lockable1& l1, Lockable2& l2, Lockable3& l3);

template<typename Lockable1, typename Lockable2, typename Lockable3, typename Lockable4>
int try_lock(Lockable1& l1, Lockable2& l2, Lockable3& l3, Lockable4& l4);

template<typename Lockable1, typename Lockable2, typename Lockable3, typename Lockable4, typename Lockable5>
int try_lock(Lockable1& l1, Lockable2& l2, Lockable3& l3, Lockable4& l4, Lockable5& l5);
```

- Effects:** Calls `try_lock()` on each of the `Lockable` objects supplied as arguments. If any of the calls to `try_lock()` returns `false` then all locks acquired are released and the zero-based index of the failed lock is returned.
- If any of the `try_lock()` operations on the supplied `Lockable` objects throws an exception any locks acquired by the function will be released before the function exits.
- Returns:** -1 if all the supplied `Lockable` objects are now locked by the calling thread, the zero-based index of the object which could not be locked otherwise.
- Throws:** Any exceptions thrown by calling `try_lock()` on the supplied `Lockable` objects.
- Postcondition:** If the function returns -1, all the supplied `Lockable` objects are locked by the calling thread. Otherwise any locks acquired by this function will have been released.

Non-member function `try_lock(begin, end) // EXTENSION`

```
template<typename ForwardIterator>
ForwardIterator try_lock(ForwardIterator begin, ForwardIterator end);
```

- Preconditions:** The `value_type` of `ForwardIterator` must implement the `Lockable` concept
- Effects:** Calls `try_lock()` on each of the `Lockable` objects in the supplied range. If any of the calls to `try_lock()` returns `false` then all locks acquired are released and an iterator referencing the failed lock is returned.
- If any of the `try_lock()` operations on the supplied `Lockable` objects throws an exception any locks acquired by the function will be released before the function exits.
- Returns:** `end` if all the supplied `Lockable` objects are now locked by the calling thread, an iterator referencing the object which could not be locked otherwise.
- Throws:** Any exceptions thrown by calling `try_lock()` on the supplied `Lockable` objects.
- Postcondition:** If the function returns `end` then all the `Lockable` objects in the supplied range are locked by the calling thread, otherwise all locks acquired by the function have been released.

Lock Factories - EXTENSION

```
namespace boost
{

    template <typename Lockable>
    unique_lock<Lockable> make_unique_lock(Lockable& mtx); // EXTENSION

    template <typename Lockable>
    unique_lock<Lockable> make_unique_lock(Lockable& mtx, adopt_lock_t); // EXTENSION
    template <typename Lockable>
    unique_lock<Lockable> make_unique_lock(Lockable& mtx, defer_lock_t); // EXTENSION
    template <typename Lockable>
    unique_lock<Lockable> make_unique_lock(Lockable& mtx, try_to_lock_t); // EXTENSION

    #if ! defined(BOOST_THREAD_NO_MAKE_UNIQUE_LOCKS)
        template <typename ...Lockable>
        std::tuple<unique_lock<Lockable> ...> make_unique_locks(Lockable& ...mtx); // EXTENSION
    #endif
}
```

Non Member Function `make_unique_lock(Lockable&)`

```
template <typename Lockable>
unique_lock<Lockable> make_unique_lock(Lockable& mtx); // EXTENSION
```

Returns: a `boost::unique_lock` as if initialized with `unique_lock<Lockable>(mtx)`.

Throws: Any exception thrown by the call to `boost::unique_lock<Lockable>(mtx)`.

Non Member Function `make_unique_lock(Lockable&,tag)`

```
template <typename Lockable>
unique_lock<Lockable> make_unique_lock(Lockable& mtx, adopt_lock_t tag); // EXTENSION

template <typename Lockable>
unique_lock<Lockable> make_unique_lock(Lockable& mtx, defer_lock_t tag); // EXTENSION

template <typename Lockable>
unique_lock<Lockable> make_unique_lock(Lockable& mtx, try_to_lock_t tag); // EXTENSION
```

Returns: a `boost::unique_lock` as if initialized with `unique_lock<Lockable>(mtx, tag)`.

Throws: Any exception thrown by the call to `boost::unique_lock<Lockable>(mtx, tag)`.

Non Member Function `make_unique_locks(Lockable& ...)`

```
template <typename ...Lockable>
std::tuple<unique_lock<Lockable> ...> make_unique_locks(Lockable& ...mtx); // EXTENSION
```

Effect: Locks all the mutexes.

Returns: a `std::tuple` of unique `boost::unique_lock` owning each one of the mutex.

Throws: Any exception thrown by `boost::lock(mtx...)`.

Mutex Types

Class `mutex`

```
#include <boost/thread/mutex.hpp>

class mutex:
    boost::noncopyable
{
public:
    mutex();
    ~mutex();

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

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

`boost::mutex` implements the [Lockable concept](#) to provide an exclusive-ownership mutex. At most one thread can own the lock on a given instance of `boost::mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()` and `unlock()` shall be permitted.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Typedef `try_mutex`

```
#include <boost/thread/mutex.hpp>

typedef mutex try_mutex;
```

`boost::try_mutex` is a typedef to `boost::mutex`, provided for backwards compatibility with previous releases of boost.

Class `timed_mutex`

```

#include <boost/thread/mutex.hpp>

class timed_mutex:
    boost::noncopyable
{
public:
    timed_mutex();
    ~timed_mutex();

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

    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& t);

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<timed_mutex> scoped_timed_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_timed_lock scoped_lock;

    #if defined BOOST_THREAD_PROVIDES_DATE_TIME || defined BOOST_THREAD_DONT_USE_CHRONO
    bool timed_lock(system_time const & abs_time);
    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);
    #endif
};

```

`boost::timed_mutex` implements the [TimedLockable concept](#) to provide an exclusive-ownership mutex. At most one thread can own the lock on a given instance of `boost::timed_mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()`, `timed_lock()`, `timed_lock()` and `unlock()` shall be permitted.

Member function `native_handle()`

```

typedef platform-specific-type native_handle_type;
native_handle_type native_handle();

```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Class `recursive_mutex`

```
#include <boost/thread/recursive_mutex.hpp>

class recursive_mutex:
    boost::noncopyable
{
public:
    recursive_mutex();
    ~recursive_mutex();

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

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<recursive_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

`boost::recursive_mutex` implements the [Lockable concept](#) to provide an exclusive-ownership recursive mutex. At most one thread can own the lock on a given instance of `boost::recursive_mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()` and `unlock()` shall be permitted. A thread that already has exclusive ownership of a given `boost::recursive_mutex` instance can call `lock()` or `try_lock()` to acquire an additional level of ownership of the mutex. `unlock()` must be called once for each level of ownership acquired by a single thread before ownership can be acquired by another thread.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Typedef `recursive_try_mutex`

```
#include <boost/thread/recursive_mutex.hpp>

typedef recursive_mutex recursive_try_mutex;
```

`boost::recursive_try_mutex` is a typedef to `boost::recursive_mutex`, provided for backwards compatibility with previous releases of boost.

Class `recursive_timed_mutex`

```
#include <boost/thread/recursive_mutex.hpp>

class recursive_timed_mutex:
    boost::noncopyable
{
public:
    recursive_timed_mutex();
    ~recursive_timed_mutex();

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

    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& t);

    typedef platform-specific-type native_handle_type;
    native_handle_type native_handle();

    typedef unique_lock<recursive_timed_mutex> scoped_lock;
    typedef unspecified-type scoped_try_lock;
    typedef scoped_lock scoped_timed_lock;

#ifdef BOOST_THREAD_PROVIDES_DATE_TIME || defined BOOST_THREAD_DONT_USE_CHRONO
    bool timed_lock(system_time const & abs_time);
    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);
#endif
};
```

`boost::recursive_timed_mutex` implements the [TimedLockable concept](#) to provide an exclusive-ownership recursive mutex. At most one thread can own the lock on a given instance of `boost::recursive_timed_mutex` at any time. Multiple concurrent calls to `lock()`, `try_lock()`, `timed_lock()`, `timed_lock()` and `unlock()` shall be permitted. A thread that already has exclusive ownership of a given `boost::recursive_timed_mutex` instance can call `lock()`, `timed_lock()`, `timed_lock()` or `try_lock()` to acquire an additional level of ownership of the mutex. `unlock()` must be called once for each level of ownership acquired by a single thread before ownership can be acquired by another thread.

Member function `native_handle()`

```
typedef platform-specific-type native_handle_type;
native_handle_type native_handle();
```

Effects: Returns an instance of `native_handle_type` that can be used with platform-specific APIs to manipulate the underlying implementation. If no such instance exists, `native_handle()` and `native_handle_type` are not present.

Throws: Nothing.

Class `shared_mutex` -- C++14

```
#include <boost/thread/shared_mutex.hpp>

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

    shared_mutex();
    ~shared_mutex();

    void lock_shared();
    bool try_lock_shared();
    template <class Rep, class Period>
    bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock_shared();

    void lock();
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

#ifdef BOOST_THREAD_PROVIDES_DEPRECATED_FEATURES_SINCE_V3_0_0
    // use upgrade_mutex instead.
    void lock_upgrade(); // EXTENSION
    void unlock_upgrade(); // EXTENSION

    void unlock_upgrade_and_lock(); // EXTENSION
    void unlock_and_lock_upgrade(); // EXTENSION
    void unlock_and_lock_shared(); // EXTENSION
    void unlock_upgrade_and_lock_shared(); // EXTENSION
#endif

#ifdef BOOST_THREAD_USES_DATETIME
    bool timed_lock_shared(system_time const& timeout); // DEPRECATED
    bool timed_lock(system_time const& timeout); // DEPRECATED
#endif
};
```

The class `boost::shared_mutex` provides an implementation of a multiple-reader / single-writer mutex. It implements the [SharedLockable concept](#).

Multiple concurrent calls to `lock()`, `try_lock()`, `try_lock_for()`, `try_lock_until()`, `timed_lock()`, `lock_shared()`, `try_lock_shared_for()`, `try_lock_shared_until()`, `try_lock_shared()` and `timed_lock_shared()` are permitted.

Note the the lack of reader-writer priority policies in `shared_mutex`. This is due to an algorithm credited to Alexander Terekhov which lets the OS decide which thread is the next to get the lock without caring whether a unique lock or shared lock is being sought. This results in a complete lack of reader or writer starvation. It is simply fair.

Class `boost::shared_mutex` -- EXTENSION

```
#include <boost/thread/shared_mutex.hpp>

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

    upgrade_mutex();
    ~upgrade_mutex();

    void lock_shared();
    bool try_lock_shared();
    template <class Rep, class Period>
    bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock_shared();

    void lock();
    bool try_lock();
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    void lock_upgrade();
    template <class Rep, class Period>
    bool try_lock_upgrade_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_upgrade_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock_upgrade();

    // Shared <--> Exclusive

#ifdef BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSIONS
    bool try_unlock_shared_and_lock();
    template <class Rep, class Period>
    bool try_unlock_shared_and_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_unlock_shared_and_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_and_lock_shared();

    // Shared <--> Upgrade

#ifdef BOOST_THREAD_PROVIDES_SHARED_MUTEX_UPWARDS_CONVERSIONS
    bool try_unlock_shared_and_lock_upgrade();
    template <class Rep, class Period>
    bool try_unlock_shared_and_lock_upgrade_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_unlock_shared_and_lock_upgrade_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_upgrade_and_lock_shared();

    // Upgrade <--> Exclusive

    void unlock_upgrade_and_lock();
    #if defined(BOOST_THREAD_PLATFORM_PTHREAD)
```

```

    || defined(BOOST_THREAD_PROVIDES_GENERIC_SHARED_MUTEX_ON_WIN)
    bool try_unlock_upgrade_and_lock();
    template <class Rep, class Period>
    bool try_unlock_upgrade_and_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_unlock_upgrade_and_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_and_lock_upgrade();
};

```

The class `boost::upgrade_mutex` provides an implementation of a multiple-reader / single-writer mutex. It implements the [UpgradeLockable concept](#).

Multiple concurrent calls to `lock()`, `try_lock()`, `try_lock_for()`, `try_lock_until()`, `timed_lock()`, `lock_shared()`, `try_lock_shared_for()`, `try_lock_shared_until()`, `try_lock_shared()` and `timed_lock_shared()` are permitted.

Class `null_mutex` -- EXTENSION

```

#include <boost/thread/null_mutex.hpp>

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

    null_mutex();
    ~null_mutex();

    void lock_shared();
    bool try_lock_shared();
#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_shared();

    void lock();
    bool try_lock();
#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock();

    void lock_upgrade();
#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_lock_upgrade_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_lock_upgrade_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_upgrade();

    // Shared <--> Exclusive

    bool try_unlock_shared_and_lock();

```

```

#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_unlock_shared_and_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_unlock_shared_and_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_and_lock_shared();

    // Shared <--> Upgrade

    bool try_unlock_shared_and_lock_upgrade();
#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_unlock_shared_and_lock_upgrade_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_unlock_shared_and_lock_upgrade_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_upgrade_and_lock_shared();

    // Upgrade <--> Exclusive

    void unlock_upgrade_and_lock();
    bool try_unlock_upgrade_and_lock();
#ifdef BOOST_THREAD_USES_CHRONO
    template <class Rep, class Period>
    bool try_unlock_upgrade_and_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
    bool try_unlock_upgrade_and_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
#endif
    void unlock_and_lock_upgrade();
};

```

The class `boost::null_mutex` provides a no-op implementation of a multiple-reader / single-writer mutex. It is a model of the [UpgradeLockable](#) concept.

Condition Variables

Synopsis

```

namespace boost
{
    enum class cv_status;
    {
        no_timeout,
        timeout
    };
    class condition_variable;
    class condition_variable_any;
    void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
}

```

The classes `condition_variable` and `condition_variable_any` provide a mechanism for one thread to wait for notification from another thread that a particular condition has become true. The general usage pattern is that one thread locks a mutex and then calls `wait` on an instance of `condition_variable` or `condition_variable_any`. When the thread is woken from the wait, then it checks to see if the appropriate condition is now true, and continues if so. If the condition is not true, then the thread then calls `wait` again to resume waiting. In the simplest case, this condition is just a boolean variable:


```

boost::condition_variable cond;
boost::mutex mut;
bool data_ready;

void process_data();

void wait_for_data_to_process()
{
    boost::unique_lock<boost::mutex> lock(mut);
    while(!data_ready)
    {
        cond.wait(lock);
    }
    process_data();
}

```

Notice that the `lock` is passed to `wait`: `wait` will atomically add the thread to the set of threads waiting on the condition variable, and unlock the mutex. When the thread is woken, the mutex will be locked again before the call to `wait` returns. This allows other threads to acquire the mutex in order to update the shared data, and ensures that the data associated with the condition is correctly synchronized.

In the mean time, another thread sets the condition to `true`, and then calls either `notify_one` or `notify_all` on the condition variable to wake one waiting thread or all the waiting threads respectively.

```

void retrieve_data();
void prepare_data();

void prepare_data_for_processing()
{
    retrieve_data();
    prepare_data();
    {
        boost::lock_guard<boost::mutex> lock(mut);
        data_ready=true;
    }
    cond.notify_one();
}

```

Note that the same mutex is locked before the shared data is updated, but that the mutex does not have to be locked across the call to `notify_one`.

This example uses an object of type `condition_variable`, but would work just as well with an object of type `condition_variable_any`: `condition_variable_any` is more general, and will work with any kind of lock or mutex, whereas `condition_variable` requires that the lock passed to `wait` is an instance of `boost::unique_lock<boost::mutex>`. This enables `condition_variable` to make optimizations in some cases, based on the knowledge of the mutex type; `condition_variable_any` typically has a more complex implementation than `condition_variable`.

Class `condition_variable`

```
//#include <boost/thread/condition_variable.hpp>

namespace boost
{
    class condition_variable
    {
    public:
        condition_variable();
        ~condition_variable();

        void notify_one() noexcept;
        void notify_all() noexcept;

        void wait(boost::unique_lock<boost::mutex>& lock);

        template<typename predicate_type>
        void wait(boost::unique_lock<boost::mutex>& lock, predicate_type predicate);

        template <class Clock, class Duration>
        typename cv_status::type
        wait_until(
            unique_lock<mutex>& lock,
            const chrono::time_point<Clock, Duration>& t);

        template <class Clock, class Duration, class Predicate>
        bool
        wait_until(
            unique_lock<mutex>& lock,
            const chrono::time_point<Clock, Duration>& t,
            Predicate pred);

        template <class Rep, class Period>
        typename cv_status::type
        wait_for(
            unique_lock<mutex>& lock,
            const chrono::duration<Rep, Period>& d);

        template <class Rep, class Period, class Predicate>
        bool
        wait_for(
            unique_lock<mutex>& lock,
            const chrono::duration<Rep, Period>& d,
            Predicate pred);

        #if defined BOOST_THREAD_USES_DATETIME
        bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::system_time const& abs_time);
        template<typename duration_type>
        bool timed_wait(boost::unique_lock<boost::mutex>& lock, duration_type const& rel_time);
        template<typename predicate_type>
        bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::system_time const& abs_time, predicate_type predicate);
        template<typename duration_type, typename predicate_type>
        bool timed_wait(boost::unique_lock<boost::mutex>& lock, duration_type const& rel_time, predicate_type predicate);
        bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::xtime const& abs_time);

        template<typename predicate_type>

```

```

        bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::xtime const& abs_time, predi-
        cate_type predicate);
    #endif

    };
}

```

condition_variable()

Effects: Constructs an object of class `condition_variable`.

Throws: `boost::thread_resource_error` if an error occurs.

~condition_variable()

Precondition: All threads waiting on `*this` have been notified by a call to `notify_one` or `notify_all` (though the respective calls to `wait` or `timed_wait` need not have returned).

Effects: Destroys the object.

Throws: Nothing.

void notify_one()

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks one of those threads.

Throws: Nothing.

void notify_all()

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks all of those threads.

Throws: Nothing.

void wait(boost::unique_lock<boost::mutex>& lock)

Precondition: `lock` is locked by the current thread, and either no other thread is currently waiting on `*this`, or the execution of the `mutex()` member function on the `lock` objects supplied in the calls to `wait` or `timed_wait` in all the threads currently waiting on `*this` would return the same value as `lock->mutex()` for this call to `wait`.

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

template<typename predicate_type> void wait(boost::unique_lock<boost::mutex>& lock, predicate_type pred)

Effects: As-if

```
while(!pred())
{
    wait(lock);
}
```

```
bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::system_time const& abs_time)
```

- Precondition:** lock is locked by the current thread, and either no other thread is currently waiting on `*this`, or the execution of the `mutex()` member function on the lock objects supplied in the calls to `wait` or `timed_wait` in all the threads currently waiting on `*this` would return the same value as `lock->mutex()` for this call to `wait`.
- Effects:** Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, when the time as reported by `boost::get_system_time()` would be equal to or later than the specified `abs_time`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.
- Returns:** `false` if the call is returning because the time specified by `abs_time` was reached, `true` otherwise.
- Postcondition:** lock is locked by the current thread.
- Throws:** `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

```
template<typename duration_type> bool timed_wait(boost::unique_lock<boost::mutex>& lock, duration_type const& rel_time)
```

- Precondition:** lock is locked by the current thread, and either no other thread is currently waiting on `*this`, or the execution of the `mutex()` member function on the lock objects supplied in the calls to `wait` or `timed_wait` in all the threads currently waiting on `*this` would return the same value as `lock->mutex()` for this call to `wait`.
- Effects:** Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, after the period of time indicated by the `rel_time` argument has elapsed, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.
- Returns:** `false` if the call is returning because the time period specified by `rel_time` has elapsed, `true` otherwise.
- Postcondition:** lock is locked by the current thread.
- Throws:** `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.



Note

The duration overload of `timed_wait` is difficult to use correctly. The overload taking a predicate should be preferred in most cases.

```
template<typename predicate_type> bool timed_wait(boost::unique_lock<boost::mutex>& lock, boost::system_time const& abs_time, predicate_type pred)
```

- Effects:** As-if

```

while(!pred())
{
    if(!timed_wait(lock,abs_time))
    {
        return pred();
    }
}
return true;

```

```

template <class Clock, class Duration> cv_status wait_until(boost::unique_lock<boost::mutex>& lock,
const chrono::time_point<Clock, Duration>& abs_time)

```

- Precondition:** lock is locked by the current thread, and either no other thread is currently waiting on *this, or the execution of the mutex() member function on the lock objects supplied in the calls to wait or wait_for or wait_until in all the threads currently waiting on *this would return the same value as lock->mutex() for this call to wait.
- Effects:** Atomically call lock.unlock() and blocks the current thread. The thread will unblock when notified by a call to this->notify_one() or this->notify_all(), when the time as reported by Clock::now() would be equal to or later than the specified abs_time, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking lock.lock() before the call to wait returns. The lock is also reacquired by invoking lock.lock() if the function exits with an exception.
- Returns:** cv_status::timeout if the call is returning because the time specified by abs_time was reached, cv_status::no_timeout otherwise.
- Postcondition:** lock is locked by the current thread.
- Throws:** boost::thread_resource_error if an error occurs. boost::thread_interrupted if the wait was interrupted by a call to interrupt() on the boost::thread object associated with the current thread of execution.

```

template <class Rep, class Period> cv_status wait_for(boost::unique_lock<boost::mutex>& lock, const
chrono::duration<Rep, Period>& rel_time)

```

- Precondition:** lock is locked by the current thread, and either no other thread is currently waiting on *this, or the execution of the mutex() member function on the lock objects supplied in the calls to wait or wait_until or wait_for in all the threads currently waiting on *this would return the same value as lock->mutex() for this call to wait.
- Effects:** Atomically call lock.unlock() and blocks the current thread. The thread will unblock when notified by a call to this->notify_one() or this->notify_all(), after the period of time indicated by the rel_time argument has elapsed, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking lock.lock() before the call to wait returns. The lock is also reacquired by invoking lock.lock() if the function exits with an exception.
- Returns:** cv_status::timeout if the call is returning because the time period specified by rel_time has elapsed, cv_status::no_timeout otherwise.
- Postcondition:** lock is locked by the current thread.
- Throws:** boost::thread_resource_error if an error occurs. boost::thread_interrupted if the wait was interrupted by a call to interrupt() on the boost::thread object associated with the current thread of execution.



Note

The duration overload of `timed_wait` is difficult to use correctly. The overload taking a predicate should be preferred in most cases.

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

Effects: As-if

```
while(!pred())
{
    if(!wait_until(lock,abs_time))
    {
        return pred();
    }
}
return true;
```

```
template <class Rep, class Period, class Predicate> bool wait_for(boost::unique_lock<boost::mutex>& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred)
```

Effects: As-if

```
return wait_until(lock, chrono::steady_clock::now() + d, boost::move(pred));
```

Class `condition_variable_any`

```

//#include <boost/thread/condition_variable.hpp>

namespace boost
{
    class condition_variable_any
    {
    public:
        condition_variable_any();
        ~condition_variable_any();

        void notify_one();
        void notify_all();

        template<typename lock_type>
        void wait(lock_type& lock);

        template<typename lock_type, typename predicate_type>
        void wait(lock_type& lock, predicate_type predicate);

        template <class lock_type, class Clock, class Duration>
        cv_status wait_until(
            lock_type& lock,
            const chrono::time_point<Clock, Duration>& t);

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

        template <class lock_type, class Rep, class Period>
        cv_status wait_for(
            lock_type& lock,
            const chrono::duration<Rep, Period>& d);

        template <class lock_type, class Rep, class Period, class Predicate>
        bool wait_for(
            lock_type& lock,
            const chrono::duration<Rep, Period>& d,
            Predicate pred);

        #if defined BOOST_THREAD_USES_DATETIME
        template<typename lock_type>
        bool timed_wait(lock_type& lock, boost::system_time const& abs_time);
        template<typename lock_type, typename duration_type>
        bool timed_wait(lock_type& lock, duration_type const& rel_time);
        template<typename lock_type, typename predicate_type>
        bool timed_wait(lock_type& lock, boost::system_time const& abs_time, predicate_type predicate);
        template<typename lock_type, typename duration_type, typename predicate_type>
        bool timed_wait(lock_type& lock, duration_type const& rel_time, predicate_type predicate);
        template<typename lock_type>
        bool timed_wait(lock_type& lock, boost::xtime const& abs_time);
        template<typename lock_type, typename predicate_type>
        bool timed_wait(lock_type& lock, boost::xtime const& abs_time, predicate_type predicate);
        #endif
    };
}

```

condition_variable_any()

Effects: Constructs an object of class `condition_variable_any`.

Throws: `boost::thread_resource_error` if an error occurs.

~condition_variable_any()

Precondition: All threads waiting on `*this` have been notified by a call to `notify_one` or `notify_all` (though the respective calls to `wait` or `timed_wait` need not have returned).

Effects: Destroys the object.

Throws: Nothing.

void notify_one()

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks one of those threads.

Throws: Nothing.

void notify_all()

Effects: If any threads are currently *blocked* waiting on `*this` in a call to `wait` or `timed_wait`, unblocks all of those threads.

Throws: Nothing.

template<typename lock_type> void wait(lock_type& lock)

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

template<typename lock_type,typename predicate_type> void wait(lock_type& lock, predicate_type pred)

Effects: As-if

```
while(!pred())
{
    wait(lock);
}
```

template<typename lock_type> bool timed_wait(lock_type& lock,boost::system_time const& abs_time)

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, when the time as reported by `boost::get_system_time()` would be equal to or later than the specified `abs_time`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the

call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `false` if the call is returning because the time specified by `abs_time` was reached, `true` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

```
template<typename lock_type,typename duration_type> bool timed_wait(lock_type& lock,duration_type
const& rel_time)
```

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, after the period of time indicated by the `rel_time` argument has elapsed, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `false` if the call is returning because the time period specified by `rel_time` has elapsed, `true` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.



Note

The duration overload of `timed_wait` is difficult to use correctly. The overload taking a predicate should be preferred in most cases.

```
template<typename lock_type,typename predicate_type> bool timed_wait(lock_type& lock, boost::sys-
tem_time const& abs_time, predicate_type pred)
```

Effects: As-if

```
while(!pred())
{
    if(!timed_wait(lock,abs_time))
    {
        return pred();
    }
}
return true;
```

```
template <class lock_type, class Clock, class Duration> cv_status wait_until(lock_type& lock, const
chrono::time_point<Clock, Duration>& abs_time)
```

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, when the time as reported by `Clock::now()` would be equal to or later than the specified `abs_time`, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `cv_status::timeout` if the call is returning because the time specified by `abs_time` was reached, `cv_status::no_timeout` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

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

Effects: Atomically call `lock.unlock()` and blocks the current thread. The thread will unblock when notified by a call to `this->notify_one()` or `this->notify_all()`, after the period of time indicated by the `rel_time` argument has elapsed, or spuriously. When the thread is unblocked (for whatever reason), the lock is reacquired by invoking `lock.lock()` before the call to `wait` returns. The lock is also reacquired by invoking `lock.lock()` if the function exits with an exception.

Returns: `cv_status::timeout` if the call is returning because the time specified by `abs_time` was reached, `cv_status::no_timeout` otherwise.

Postcondition: `lock` is locked by the current thread.

Throws: `boost::thread_resource_error` if an error occurs. `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.



Note

The duration overload of `timed_wait` is difficult to use correctly. The overload taking a predicate should be preferred in most cases.

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

Effects: As-if

```
while(!pred())
{
    if(!wait_until(lock,abs_time))
    {
        return pred();
    }
}
return true;
```

```
template <class lock_type, class Rep, class Period, class Predicate> bool wait_for(lock_type& lock,
const chrono::duration<Rep, Period>& rel_time, Predicate pred)
```

Effects: As-if

```
return wait_until(lock, chrono::steady_clock::now() + d, boost::move(pred));
```

Typedef `condition` DEPRECATED V3

```
// #include <boost/thread/condition.hpp>
namespace boost
{
    typedef condition_variable_any condition;
}
```

The typedef `condition` is provided for backwards compatibility with previous boost releases.

Non-member Function `notify_all_at_thread_exit()`

```
// #include <boost/thread/condition_variable.hpp>
namespace boost
{
    void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
}
```

Requires: `lk` is locked by the calling thread and either no other thread is waiting on `cond`, or `lk.mutex()` returns the same value for each of the lock arguments supplied by all concurrently waiting (via `wait`, `wait_for`, or `wait_until`) threads.

Effects: transfers ownership of the lock associated with `lk` into internal storage and schedules `cond` to be notified when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed. This notification shall be as if

```
lk.unlock();
cond.notify_all();
```

One-time Initialization

```
#include <boost/thread/once.hpp>
namespace boost
{
    struct once_flag;
    template<typename Function, class ...ArgTypes>
    inline void call_once(once_flag& flag, Function&& f, ArgTypes&&... args);

    #if defined BOOST_THREAD_PROVIDES_DEPRECATED_FEATURES_SINCE_V3_0_0
    void call_once(void (*func)(), once_flag& flag);
    #endif
}
```



Warning

the variadic prototype is provided only on C++11 compilers supporting variadic templates, otherwise the interface is limited up to 3 parameters.



Warning

the move semantics is ensured only on C++11 compilers supporting SFINAE expression, decltype N3276 and auto. Waiting for a boost::bind that is move aware.

boost::call_once provides a mechanism for ensuring that an initialization routine is run exactly once without data races or deadlocks.

Typedef `once_flag`

```
#ifdef BOOST_THREAD_PROVIDES_ONCE_CXX11
struct once_flag
{
    constexpr once_flag() noexcept;
    once_flag(const once_flag&) = delete;
    once_flag& operator=(const once_flag&) = delete;
};
#else
typedef platform-specific-type once_flag;
#define BOOST_ONCE_INIT platform-specific-initializer
#endif
```

Objects of type boost::once_flag shall be initialized with BOOST_ONCE_INIT if BOOST_THREAD_PROVIDES_ONCE_CXX11 is not defined

```
boost::once_flag f=BOOST_ONCE_INIT;
```

Non-member function `call_once`

```
template<typename Function, class ...ArgTypes>
inline void call_once(once_flag& flag, Function&& f, ArgTypes&&... args);
```

- | | |
|------------------|---|
| Requires: | Function and each of the ArgTypes are MoveConstructible and invoke(decay_copy(boost::forward<Function>(f)), decay_copy(boost::forward<ArgTypes>(args))...) shall be well formed. |
| Effects: | Calls to call_once on the same once_flag object are serialized. If there has been no prior effective call_once on the same once_flag object, the argument func is called as-if by invoking invoke(decay_copy(boost::forward<Function>(f)), decay_copy(boost::forward<ArgTypes>(args))...), and the invocation of call_once is effective if and only if invoke(decay_copy(boost::forward<Function>(f)), decay_copy(boost::forward<ArgTypes>(args))...) returns without exception. If an exception is thrown, the exception is propagated to the caller. If there has been a prior effective call_once on the same once_flag object, the call_once returns without invoking func. |
| Synchronization: | The completion of an effective call_once invocation on a once_flag object, synchronizes with all subsequent call_once invocations on the same once_flag object. |
| Throws: | thread_resource_error when the effects cannot be achieved or any exception propagated from func. |
| Note: | The function passed to call_once must not also call call_once passing the same once_flag object. This may cause deadlock, or invoking the passed function a second time. The alternative is to allow the second call to return immediately, but that assumes the code knows it has been called recursively, and can proceed even though the call to call_once didn't actually call the function, in which case it could also avoid calling call_once recursively. |

Note: On some compilers this function has some restrictions, e.g. if variadic templates are not supported the number of arguments is limited to 3; .

```
void call_once(void (*func)(), once_flag& flag);
```

This second overload is provided for backwards compatibility and is deprecated. The effects of `call_once(func, flag)` shall be the same as those of `call_once(flag, func)`.

Barriers -- EXTENSION

A barrier is a simple concept. Also known as a *rendezvous*, it is a synchronization point between multiple threads. The barrier is configured for a particular number of threads (*n*), and as threads reach the barrier they must wait until all *n* threads have arrived. Once the *n*-th thread has reached the barrier, all the waiting threads can proceed, and the barrier is reset.

Class `barrier`

```
#include <boost/thread/barrier.hpp>

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

    barrier(unsigned int count);
    template <typename F>
    barrier(unsigned int count, F&&);

    ~barrier();

    bool wait();
    void count_down_and_wait();
};
```

Instances of `boost::barrier` are not copyable or movable.

Constructor `barrier(unsigned int)`

```
barrier(unsigned int count);
```

Effects: Construct a barrier for `count` threads.

Throws: `boost::thread_resource_error` if an error occurs.

Constructor `barrier(unsigned int, F&&)`

```
barrier(unsigned int count, F&& completion);
```

Requires: The result type of the completion function call `completion()` is `void` or `unsigned int`.

Effects: Construct a barrier for `count` threads and a completion function `completion`.

Throws: `boost::thread_resource_error` if an error occurs.

Destructor `~barrier()`

```
~barrier();
```

Precondition: No threads are waiting on `*this`.

Effects: Destroys `*this`.

Throws: Nothing.

Member Function `wait()`

```
bool wait();
```

Effects: Block until `count` threads have called `wait` or `count_down_and_wait` on `*this`. When the `count`-th thread calls `wait`, the barrier is reset and all waiting threads are unblocked. The reset depends on whether the barrier was constructed with a completion function or not. If there is no completion function or if the completion function result is void, the reset consists in restoring the original count. Otherwise the rest consist in assigning the result of the completion function (which must not be 0).

Returns: `true` for exactly one thread from each batch of waiting threads, `false` otherwise.

Throws: `- boost::thread_resource_error` if an error occurs.

`- boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `wait()` is an *interruption point*.

Member Function `count_down_and_wait()`

```
void count_down_and_wait();
```

Effects: Block until `count` threads have called `wait` or `count_down_and_wait` on `*this`. When the `count`-th thread calls `wait`, the barrier is reset and all waiting threads are unblocked. The reset depends on whether the barrier was constructed with a completion function or not. If there is no completion function or if the completion function result is void, the reset consists in restoring the original count. Otherwise the rest consist in assigning the result of the completion function (which must not be 0).

Throws: `- boost::thread_resource_error` if an error occurs.

`- boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `count_down_and_wait()` is an *interruption point*.

Latches -- EXPERIMENTAL

Introduction

Latches are a thread co-ordination mechanism that allow one or more threads to block until one or more threads have reached a point.

Examples

Sample use cases for the latch include:

- Setting multiple threads to perform a task, and then waiting until all threads have reached a common point.
- Creating multiple threads, which wait for a signal before advancing beyond a common point.

An example of the first use case would be as follows:

```
void DoWork(thread_pool* pool) {
    latch completion_latch(NTASKS);
    for (int i = 0; i < NTASKS; ++i) {
        pool->submit([&] {
            // perform work
            ...
            completion_latch.count_down();
        });
    }
    // Block until work is done
    completion_latch.wait();
}
```

An example of the second use case is shown below. We need to load data and then process it using a number of threads. Loading the data is I/O bound, whereas starting threads and creating data structures is CPU bound. By running these in parallel, throughput can be increased.

```
void DoWork() {
    latch start_latch(1);
    vector<thread*> workers;
    for (int i = 0; i < NTHREADS; ++i) {
        workers.push_back(new thread([&] {
            // Initialize data structures. This is CPU bound.
            ...
            start_latch.wait();
            // perform work
            ...
        }));
    }
    // Load input data. This is I/O bound.
    ...
    // Threads can now start processing
    start_latch.count_down();
}
```

Class `latch`

```
#include <boost/thread/latch.hpp>

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

    latch(std::size_t count);
    ~latch();

    void wait();
    bool try_wait();
    template <class Rep, class Period>
    cv_status wait_for(const chrono::duration<Rep, Period>& rel_time);
    template <class lock_type, class Clock, class Duration>
    cv_status wait_until(const chrono::time_point<Clock, Duration>& abs_time);
    void count_down();
    void count_down_and_wait();

};
```

A latch maintains an internal counter that is initialized when the latch is created. One or more threads may block waiting until the counter is decremented to 0.

Instances of `latch` are not copyable or movable.

Constructor `latch(std::size_t)`

```
latch(std::size_t count);
```

Effects: Construct a latch with its initial value for the internal counter.

Note: The counter could be zero.

Throws: Nothing.

Destructor `~latch()`

```
~latch();
```

Precondition: No threads are waiting or invoking `count_down` on `*this`.

Effects: Destroys `*this` latch.

Throws: Nothing.

Member Function `wait()`

```
void wait();
```

Effects: Block the calling thread until the internal count reaches the value zero. Then all waiting threads are unblocked.

Throws: - `boost::thread_resource_error` if an error occurs.

- `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `wait()` is an *interruption point*.

Member Function `try_wait()`

```
bool try_wait();
```

Returns: Returns true if the internal count is 0, and false otherwise. Does not block the calling thread.

Throws: - `boost::thread_resource_error` if an error occurs.

Member Function `wait_for()`

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

Effects: Block the calling thread until the internal count reaches the value zero or the duration has been elapsed. If no timeout, all waiting threads are unblocked.

Returns: `cv_status::no_timeout` if the internal count is 0, and `cv_status::timeout` if duration has been elapsed.

Throws: - `boost::thread_resource_error` if an error occurs.

- `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `wait_for()` is an *interruption point*.

Member Function `wait_until()`

```
template <class lock_type, class Clock, class Duration>
cv_status wait_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Effects: Block the calling thread until the internal count reaches the value zero or the `time_point` has been reached. If no timeout, all waiting threads are unblocked.

Returns: `cv_status::no_timeout` if the internal count is 0, and `cv_status::timeout` if `time_point` has been reached.

Throws: - `boost::thread_resource_error` if an error occurs.

- `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `wait_until()` is an *interruption point*.

Member Function `count_down()`

```
void count_down();
```

Requires: The internal counter is non zero.

Effects: Decrements the internal count by 1, and returns. If the count reaches 0, any threads blocked in `wait()` will be released.

Throws: - `boost::thread_resource_error` if an error occurs.

- `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `count_down()` is an *interruption point*.

Member Function `count_down_and_wait()`

```
void count_down_and_wait();
```

Requires: The internal counter is non zero.

Effects: Decrements the internal count by 1. If the resulting count is not 0, blocks the calling thread until the internal count is decremented to 0 by one or more other threads calling `count_down()` or `count_down_and_wait()`.

Throws: - `boost::thread_resource_error` if an error occurs.

- `boost::thread_interrupted` if the wait was interrupted by a call to `interrupt()` on the `boost::thread` object associated with the current thread of execution.

Notes: `count_down_and_wait()` is an *interruption point*.

[

Member Function `reset()`

```
reset( size_t );
```

Requires: This function may only be invoked when there are no other threads currently inside the waiting functions.

Returns: Resets the latch with a new value for the initial thread count.

Throws: - `boost::thread_resource_error` if an error occurs.

]

Executors and Schedulers -- EXPERIMENTAL



Warning

These features are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(



Note

These features are based on the [N3785 - Executors and Schedulers revision 3](#) C++1y proposal from Chris Mysisen, Niklas Gustafsson, Matt Austern, Jeffrey Yasskin. The text that follows has been adapted from tis paper to show the differences.

Executors are objects that can execute units of work packaged as function objects. Boost.Thread differs from N3785 mainly in the an Executor doesn't needs to inherit from an abstract class Executor. Static polymorphism is used instead and type erasure is used internally.

Introduction

Multithreaded programs often involve discrete (sometimes small) units of work that are executed asynchronously. This often involves passing work units to some component that manages execution. We already have `boost::async`, which potentially executes a function asynchronously and eventually returns its result in a future. (“As if” by launching a new thread.)

If there is a regular stream of small work items then we almost certainly don’t want to launch a new thread for each, and it’s likely that we want at least some control over which thread(s) execute which items. It is often convenient to represent that control as multiple executor objects. This allows programs to start executors when necessary, switch from one executor to another to control execution policy, and use multiple executors to prevent interference and thread exhaustion. Several possible implementations exist of the executor class and in practice there are a number of main groups of executors which have been found to be useful in real-world code (more implementations exist, this is simply a high level classification of them). These differ along a couple main dimensions, how many execution contexts will be used, how they are selected, and how they are prioritized.

1. Thread Pools

- a. Simple unbounded thread pool, which can queue up an unbounded amount of work and maintains a dedicated set of threads (up to some maximum) which dequeue and execute work as available.
- b. Bounded thread pools, which can be implemented as a specialization of the previous ones with a bounded queue or semaphore, which limits the amount of queuing in an attempt to bound the time spent waiting to execute and/or limit resource utilization for work tasks which hold state which is expensive to hold.
- c. Thread-spawning executors, in which each work always executes in a new thread.
- d. Prioritized thread pools, which have works which are not equally prioritized such that work can move to the front of the execution queue if necessary. This requires a special comparator or prioritization function to allow for work ordering and normally is implemented as a blocking priority queue in front of the pool instead of a blocking queue. This has many uses but is a somewhat specialized in nature and would unnecessarily clutter the initial interface.
- e. Work stealing thread pools, this is a specialized use case and is encapsulated in the `ForkJoinPool` in java, which allows lightweight work to be created by tasks in the pool and either run by the same thread for invocation efficiency or stolen by another thread without additional work. These have been left out until there is a more concrete fork-join proposal or until there is a more clear need as these can be complicated to implement

2. Mutual exclusion executors

- a. Serial executors, which guarantee all work to be executed such that no two works will execute concurrently. This allows for a sequence of operations to be queued in sequence and that sequential order is maintained and work can be queued on a separate thread but with no mutual exclusion required.
- b. Loop executor, in which one thread donates itself to the executor to execute all queued work. This is related to the serial executor in that it guarantees mutual exclusion, but instead guarantees a particular thread will execute the work. These are particularly useful for testing purposes where code assumes an executor but testing code desires control over execution.
- c. GUI thread executor, where a GUI framework can expose an executor interface to allow other threads to queue up work to be executed as part of the GUI thread. This behaves similarly to a loop executor, but must be implemented as a custom interface as part of the framework.

3. Inline executors, which execute inline to the thread which calls `submit()`. This has no queuing and behaves like a normal executor, but always uses the caller’s thread to execute. This allows parallel execution of works, though. This type of executor is often useful when there is an executor required by an interface, but when for performance reasons it’s better not to queue work or switch threads. This is often very useful as an optimization for work continuations which should execute immediately or quickly and can also be useful for optimizations when an interface requires an executor but the work tasks are too small to justify the overhead of a full thread pool.

A question arises of which of these executors (or others) be included in this library. There are use cases for these and many other executors. Often it is useful to have more than one implemented executor (e.g. the thread pool) to have more precise control of where the work is executed due to the existence of a GUI thread, or for testing purposes. A few core executors are frequently useful and these have been outlined here as the core of what should be in this library, if common use cases arise for alternative executor imple-

mentations, they can be added in the future. The current set provided here are: a basic thread pool `basic_thread_pool`, a serial executor `serial_executor`, a loop executor `loop_executor`, an inline executor `inline_executor` and a thread-spawning executor `thread_executor`.

Examples

Parallel Quick Sort

```
#include <boost/thread/executors/basic_thread_pool.hpp>
#include <boost/thread/future.hpp>
#include <numeric>
#include <algorithm>
#include <functional>
#include <iostream>
#include <list>

template<typename T>
struct sorter
{
    boost::basic_thread_pool pool;
    typedef std::list<T> return_type;

    std::list<T> do_sort(std::list<T> chunk_data)
    {
        if(chunk_data.empty()) {
            return chunk_data;
        }

        std::list<T> result;
        result.splice(result.begin(), chunk_data, chunk_data.begin());
        T const& partition_val=*result.begin();

        typename std::list<T>::iterator divide_point =
            std::partition(chunk_data.begin(), chunk_data.end(),
                          [&](T const& val){return val<partition_val;});

        std::list<T> new_lower_chunk;
        new_lower_chunk.splice(new_lower_chunk.end(), chunk_data,
                              chunk_data.begin(), divide_point);
        boost::future<std::list<T>> new_lower =
            boost::async(pool, &sorter::do_sort, this, std::move(new_lower_chunk));
        std::list<T> new_higher(do_sort(chunk_data));
        result.splice(result.end(), new_higher);
        while(!new_lower.is_ready()) {
            pool.schedule_one_or_yield();
        }
        result.splice(result.begin(), new_lower.get());
        return result;
    }
};

template<typename T>
std::list<T> parallel_quick_sort(std::list<T>& input) {
    if(input.empty()) {
        return input;
    }
    sorter<T> s;
    return s.do_sort(input);
}
```

Design Rationale

The authors of Boost.Thread have taken a different approach respect to N3785. Instead of basing all the design on a abstract executor class we make executor concepts. We believe that this is the good direction as a static polymorphic executor can be seen as a dynamic polymorphic executor using a simple adaptor. We believe also that it would make the library more usable, and more convenient for users.

The major design decisions concern deciding what a unit of work is, how to manage with units of work and time related functions in a polymorphic way.

An Executor is an object that schedules the closures that have been submitted to it, usually asynchronously. There could be multiple models of the Executor class. Some specific design notes:

- Thread pools are well know models of the Executor concept, and this library does indeed include a `basic_thread_pool` class, but other implementations also exist, including the ability to schedule work on GUI threads, scheduling work on a donor thread, as well as several specializations of thread pools.
- The choice of which executor to use is explicit. This is important for reasons described in the Motivation section. In particular, consider the common case of an asynchronous operation that itself spawns asynchronous operations. If both operations ran on the same executor, and if that executor had a bounded number of worker threads, then we could get deadlock. Programs often deal with such issues by splitting different kinds of work between different executors.
- Even if there could be a strong value in having a default executor, that can be used when detailed control is unnecessary, the authors don't know how to implement it in a portable a robust way.
- The library provides Executors based on static and dynamic polymorphism. The static polymorphism interface is intended to be used on contexts that need to have the best performances. The dynamic polymorphism interface has the advantage to been able to change the executor a function is suing without making it a template and it possible to pass executors across a binary interface. For some applications, the cost of an additional virtual dispatch could be almost certainly negligible compared to the other operations involved.
- Conceptually, an executor puts closures on a queue and at some point executes them. The queue is always unbounded, so adding a closure to an executor never blocks. (Defining "never blocks" formally is challenging, but informally we just mean that `submit()` is an ordinary function that executes something and returns, rather than waiting for the completion of some potentially long running operation in another thread.)

Closure

One important question is just what a closure is. This library has a very simple answer: a closure is a `Callable` with no parameters and returning `void`.

N3785 choose the more specific `std::function<void()>` as it provides only dynamic polymorphism and states that in practice the implementation of a template based approach or another approach is impractical. The authors of this library think that the template based approach is compatible with a dynamic based approach. They give some arguments:

The first one is that a virtual function can not be a template. This is true but it is also true that the executor interface can provide the template functions that call to the virtual public functions. Another reason they give is that "a template parameter would complicate the interface without adding any real generality. In the end an executor class is going to need some kind of type erasure to handle all the different kinds of function objects with `void()` signature, and that's exactly what `std::function` already does". We think that it is up to the executor to manage with this implementation details, not to the user.

We share all the argument they give related to the `void()` interface of the work unit. A work unit is a closure that takes no arguments and returns no value. This is indeed a limitation on user code, but combined with `boost::async` taking executors as parameters the user has all what is needs.

The third one is related to performance. They assert that "any mechanism for storing closures on an executor's queue will have to use some form of type erasure. There's no reason to believe that a custom closure mechanism, written just for `std::executor` and used nowhere else within the standard library, would be better in that respect than `std::function<void()>`". We believe that the implementation can do better that storing the closure on a `std::function<void()>`. e.g. the implementation can use intrusive data to store the closure and the pointers to other nodes needed to store the closures in a given order.

In addition `std::function<void()>` can not be constructed by moving the closure, so e.g. `std::packaged_task` could not be a Closure.

Not Handled Exceptions

As in N3785 and based on the same design decision than `std/boost::thread` if a user closure throws an exception, the executor must call the `std::terminate` function. Note that when we combine `boost::async` and `Executors`, the exception will be caught by the closure associated to the returned future, so that the exception is stored on the returned future, as for the other `async` overloads.

At thread entry

It is common idiom to set some thread local variable at the beginning of a thread. As `Executors` could instantiate threads internally these `Executors` shall have the ability to call a user specific function at thread entry on the executor constructor.

For executors that don't instantiate any thread and that would use the current thread this function shall be called only for the thread calling the `at_thread_entry` member function.

Cancelation

The library does not provision yet for the ability to cancel/interrupt work, though this is a commonly requested feature.

This could be managed externally by an additional cancelation object that can be shared between the creator of the unit of work and the unit of work.

We can think also of a cancelable closure that could be used in a more transparent way.

An alternative is to make `async` return a `cancelable_task` but this will need also a cancelable closure.

Current executor

The library does not provision for the ability to get the current executor, though having access to it could simplify a lot the user code.

The reason is that the user can always use a `thread_local` variable and reset it using the `at_thread_entry` member function.

```
thread_local current_executor_state_type current_executor_state;
executor* current_executor() { return current_executor_state.current_executor(); }
basic_thread_pool pool(
    // at_thread_entry
    [](basic_thread_pool& pool) {
        current_executor_state.set_current_executor(pool);
    }
);
```

Default executor

The library authors share some of the concerns of the C++ standard committee (introduction of a new single shared resource, a singleton, could make it difficult to make it portable to all the environments) and that this library doesn't need to provide a default executor for the time being.

The user can always define his default executor himself and use the `at_thread_entry` member function to set the default constructor.

```

thread_local default_executor_state_type default_executor_state;
executor* default_executor() { return default_executor_state.default_executor(); }

// in main
MyDefaultExecutor myDefaultExecutor(
    // at_thread_entry
    [] (MyDefaultExecutor& ex) {
        default_executor_state.set_default_executor(ex);
    }
);

basic_thread_pool pool(
    // at_thread_entry
    [&myDefaultExecutor] (basic_thread_pool& pool) {
        default_executor_state.set_default_executor(myDefaultExecutor);
    }
);

```

Reference

Concept Closure

A type `E` meets the Closure requirements if is a model of `Callable(void())` and a model of `CopyConstructible/MoveConstructible`.

Concept Executor

The `Executor` concept models the common operations of all the executors.

A type `E` meets the `Executor` requirements if the following expressions are well-formed and have the specified semantics

- `E::work`
- `e.submit(lw);`
- `e.submit(rw);`
- `e.submit(lc);`
- `e.submit(rc);`
- `e.close();`
- `b = e.closed();`
- `e.try_executing_one();`
- `e.reschedule_until(p);`

where

- `e` denotes a value of type `E`,
- `lw` denotes a lvalue referece of type `E::work`,
- `rc` denotes a rvalue referece of type `E::work`
- `lc` denotes a lvalue referece of type `Closure`,
- `rc` denotes a rvalue referece of type `Closure`

- `p` denotes a value of type `Predicate`

`e.submit(lw);`

Effects:	The specified closure will be scheduled for execution at some point in the future. If invoked closure throws an exception the executor will call <code>std::terminate</code> , as is the case with threads.
Synchronization:	completion of closure on a particular thread happens before destruction of thread's thread local variables.
Return type:	<code>void</code> .
Throws:	<code>sync_queue_is_closed</code> if the thread pool is closed. Whatever exception that can be throw while storing the closure.
Exception safety:	If an exception is thrown then the executor state is unmodified.

`e.submit(rw);`

Effects:	The specified closure will be scheduled for execution at some point in the future. If invoked closure throws an exception the executor will call <code>std::terminate</code> , as is the case with threads.
Synchronization:	completion of closure on a particular thread happens before destruction of thread's thread local variables.
Return type:	<code>void</code> .
Throws:	<code>sync_queue_is_closed</code> if the thread pool is closed. Whatever exception that can be throw while storing the closure.
Exception safety:	If an exception is thrown then the executor state is unmodified.

`e.submit(lc);`

Effects:	The specified closure will be scheduled for execution at some point in the future. If invoked closure throws an exception the executor will call <code>std::terminate</code> , as is the case with threads.
Synchronization:	completion of closure on a particular thread happens before destruction of thread's thread local variables.
Return type:	<code>void</code> .
Throws:	<code>sync_queue_is_closed</code> if the thread pool is closed. Whatever exception that can be throw while storing the closure.
Exception safety:	If an exception is thrown then the executor state is unmodified.

`e.submit(lc);`

Effects:	The specified closure will be scheduled for execution at some point in the future. If invoked closure throws an exception the executor will call <code>std::terminate</code> , as is the case with threads.
Synchronization:	completion of closure on a particular thread happens before destruction of thread's thread local variables.
Return type:	<code>void</code> .
Throws:	<code>sync_queue_is_closed</code> if the thread pool is closed. Whatever exception that can be throw while storing the closure.
Exception safety:	If an exception is thrown then the executor state is unmodified.

`e.close();`

Effects:	close the executor <code>e</code> for submissions.
----------	--

Remark: The worker threads will work until there is no more closures to run.

Return type: `void`.

Throws: Whatever exception that can be throw while ensuring the thread safety.

Exception safety: If an exception is thrown then the executor state is unmodified.

`b = e.close();`

Return type: `bool`.

Return: `void`.

Throws: whether the pool is closed for submissions.

`e.try_executing_one();`

Effects: try to execute one work.

Remark: whether a work has been executed.

Return type: `bool`.

Return: Whether a work has been executed.

Throws: whatever the current work constructor throws or the `work()` throws.

`e.reschedule_until(p);`

Requires: This must be called from an scheduled work

Effects: reschedule works until `p()`.

Return type: `bool`.

Return: Whether a work has been executed.

Throws: whatever the current work constructor throws or the `work()` throws.

Class `work`

```
#include <boost/thread/work.hpp>
namespace boost {
    typedef 'implementation_defined' work;
}
```

Requires: `work` is a model of 'Closure'

Class `executor`

Executor abstract base class.

```

#include <boost/thread/executor.hpp>
namespace boost {
    class executor
    {
    public:
        typedef boost::work work;

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

        executor();
        virtual ~executor() {};

        virtual void close() = 0;
        virtual bool closed() = 0;

        virtual void submit(work&& closure) = 0;
        template <typename Closure>
        void submit(Closure&& closure);

        virtual bool try_executing_one() = 0;
        template <typename Pred>
        bool reschedule_until(Pred const& pred);
    };
}

```

Constructor `executor()`

```
executor();
```

Effects: Constructs a executor.

Throws: Nothing.

Destructor `~executor()`

```
virtual ~executor();
```

Effects: Destroys the executor.

Synchronization: The completion of all the closures happen before the completion of the executor destructor.

Template Class `executor_adaptor`

Polymorphic adaptor of a model of Executor to an executor.

```

#include <boost/thread/executor.hpp>
namespace boost {
    template <typename Executor>
    class executor_adaptor : public executor
    {
        Executor ex; // for exposition only
    public:
        typedef executor::work work;

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

        template <typename ...Args>
        executor_adaptor(Args&& ... args);

        Executor& underlying_executor();

        void close();
        bool closed();

        void submit(work&& closure);

        bool try_executing_one();

    };
}

```

Constructor `executor_adaptor(Args&& ...)`

```

template <typename ...Args>
executor_adaptor(Args&& ... args);

```

Effects: Constructs a `executor_adaptor`.

Throws: Nothing.

Destructor `~executor_adaptor()`

```

virtual ~ executor_adaptor();

```

Effects: Destroys the `executor_adaptor`.

Synchronization: The completion of all the closures happen before the completion of the executor destructor.

Function member `underlying_executor()`

```

Executor& underlying_executor();

```

Return: The underlying executor instance.

Throws: Nothing.

Template Class `serial_executor`

A serial executor ensuring that there are no two work units that executes concurrently.

```

#include <boost/thread/serial_executor.hpp>
namespace boost {
    template <class Executor>
    class serial_executor
    {
    public:
        typedef    executors::work work;

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

        serial_executor(Executor& ex);

        Executor& underlying_executor();

        void close();
        bool closed();

        void submit(work&& closure);
        template <typename Closure>
        void submit(Closure&& closure);

        bool try_executing_one();
        template <typename Pred>
        bool reschedule_until(Pred const& pred);

    };
}

```

Constructor `serial_executor(Executor&, chrono::duration<Rep, Period>)`

```
serial_executor(Executor& ex);
```

Effects: Constructs a serial_executor.

Throws: Nothing.

Destructor `~serial_executor()`

```
~serial_executor();
```

Effects: Destroys the serial_executor.

Synchronization: The completion of all the closures happen before the completion of the executor destructor.

Function member `underlying_executor()`

```
Executor& underlying_executor();
```

Return: The underlying executor instance.

Throws: Nothing.

Class `basic_thread_pool`

A thread pool with up to a fixed number of threads.

```

#include <boost/thread/work.hpp>
namespace boost {
    class basic_thread_pool
    {
    public:
        typedef boost::work work;

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

        basic_thread_pool(unsigned const thread_count = thread::hardware_concurrency());
        template <class AtThreadEntry>
        basic_thread_pool( unsigned const thread_count, AtThreadEntry at_thread_entry);
        ~basic_thread_pool();

        void close();
        bool closed();

        template <typename Closure>
        void submit(Closure&& closure);

        bool try_executing_one();

        template <typename Pred>
        bool reschedule_until(Pred const& pred);

    };
}

```

Constructor `basic_thread_pool(unsigned const)`

Effects: creates a thread pool that runs closures on `thread_count` threads.

Throws: Whatever exception is thrown while initializing the needed resources.

Destructor `~basic_thread_pool()`

```
virtual ~basic_thread_pool();
```

Effects: Destroys the thread pool.

Synchronization: The completion of all the closures happen before the completion of the executor destructor.

Class `loop_executor`

A user scheduled executor.

```

#include <boost/thread/loop_executor.hpp>
namespace boost {
    class loop_executor
    {
    public:
        typedef thread_detail::work work;

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

        loop_executor();
        ~loop_executor();

        void close();
        bool closed();

        template <typename Closure>
        void submit(Closure&& closure);

        bool try_executing_one();
        template <typename Pred>
        bool reschedule_until(Pred const& pred);

        void loop();
        void run_queued_closures();
    };
}

```

Constructor `loop_executor()`

```
loop_executor();
```

Effects: creates a executor that runs closures using one of its closure-executing methods.

Throws: Whatever exception is thrown while initializing the needed resources.

Destructor `~loop_executor()`

```
virtual ~loop_executor();
```

Effects: Destroys the thread pool.

Synchronization: The completion of all the closures happen before the completion of the executor destructor.

Function member `loop()`

```
void loop();
```

Return: reschedule works until `closed()` or empty.

Throws: whatever the current work constructor throws or the `work()` throws.

Function member `run_queued_closures()`

```
void run_queued_closures();
```

Return: reschedule the enqueued works.

Throws: whatever the current work constructor throws or the `work()` throws.

Futures

Overview

The futures library provides a means of handling synchronous future values, whether those values are generated by another thread, or on a single thread in response to external stimuli, or on-demand.

This is done through the provision of four class templates: `future` and `boost::shared_future` which are used to retrieve the asynchronous results, and `boost::promise` and `boost::packaged_task` which are used to generate the asynchronous results.

An instance of `future` holds the one and only reference to a result. Ownership can be transferred between instances using the move constructor or move-assignment operator, but at most one instance holds a reference to a given asynchronous result. When the result is ready, it is returned from `boost::future<R>::get()` by rvalue-reference to allow the result to be moved or copied as appropriate for the type.

On the other hand, many instances of `boost::shared_future` may reference the same result. Instances can be freely copied and assigned, and `boost::shared_future<R>::get()` returns a non `const` reference so that multiple calls to `boost::shared_future<R>::get()` are safe. You can move an instance of `future` into an instance of `boost::shared_future`, thus transferring ownership of the associated asynchronous result, but not vice-versa.

`boost::async` is a simple way of running asynchronous tasks. A call to `boost::async` returns a `future` that will contain the result of the task.

You can wait for futures either individually or with one of the `boost::wait_for_any()` and `boost::wait_for_all()` functions.

Creating asynchronous values

You can set the value in a future with either a `boost::promise` or a `boost::packaged_task`. A `boost::packaged_task` is a callable object that wraps a function or callable object. When the packaged task is invoked, it invokes the contained function in turn, and populates a future with the return value. This is an answer to the perennial question: "how do I return a value from a thread?": package the function you wish to run as a `boost::packaged_task` and pass the packaged task to the thread constructor. The future retrieved from the packaged task can then be used to obtain the return value. If the function throws an exception, that is stored in the future in place of the return value.

```
int calculate_the_answer_to_life_the_universe_and_everything()
{
    return 42;
}

boost::packaged_task<int> pt(calculate_the_answer_to_life_the_universe_and_everything);
boost::future<int> fi=pt.get_future();

boost::thread task(boost::move(pt)); // launch task on a thread

fi.wait(); // wait for it to finish

assert(fi.is_ready());
assert(fi.has_value());
assert(!fi.has_exception());
assert(fi.get_state()==boost::future_state::ready);
assert(fi.get()==42);
```

A `boost::promise` is a bit more low level: it just provides explicit functions to store a value or an exception in the associated future. A promise can therefore be used where the value may come from more than one possible source, or where a single operation may produce multiple values.

```

boost::promise<int> pi;
boost::future<int> fi;
fi=pi.get_future();

pi.set_value(42);

assert(fi.is_ready());
assert(fi.has_value());
assert(!fi.has_exception());
assert(fi.get_state()==boost::future_state::ready);
assert(fi.get()==42);

```

Wait Callbacks and Lazy Futures

Both `boost::promise` and `boost::packaged_task` support *wait callbacks* that are invoked when a thread blocks in a call to `wait()` or `timed_wait()` on a future that is waiting for the result from the `boost::promise` or `boost::packaged_task`, in the thread that is doing the waiting. These can be set using the `set_wait_callback()` member function on the `boost::promise` or `boost::packaged_task` in question.

This allows *lazy futures* where the result is not actually computed until it is needed by some thread. In the example below, the call to `f.get()` invokes the callback `invoke_lazy_task`, which runs the task to set the value. If you remove the call to `f.get()`, the task is not ever run.

```

int calculate_the_answer_to_life_the_universe_and_everything()
{
    return 42;
}

void invoke_lazy_task(boost::packaged_task<int>& task)
{
    try
    {
        task();
    }
    catch(boost::task_already_started&)
    {}
}

int main()
{
    boost::packaged_task<int> task(calculate_the_answer_to_life_the_universe_and_everything);
    task.set_wait_callback(invoke_lazy_task);
    boost::future<int> f(task.get_future());

    assert(f.get()==42);
}

```

Handling Detached Threads and Thread Specific Variables

Detached threads pose a problem for objects with thread storage duration. If we use a mechanism other than `thread::__join` to wait for a `thread` to complete its work - such as waiting for a future to be ready - then the destructors of thread specific variables will still be running after the waiting thread has resumed. This section explain how the standard mechanism can be used to make such synchronization safe by ensuring that the objects with thread storage duration are destroyed prior to the future being made ready. e.g.


```

int find_the_answer(); // uses thread specific objects
void thread_func(boost::promise<int>&& p)
{
    p.set_value_at_thread_exit(find_the_answer());
}

int main()
{
    boost::promise<int> p;
    boost::thread t(thread_func, boost::move(p));
    t.detach(); // we're going to wait on the future
    std::cout<<p.get_future().get()<<std::endl;
}

```

When the call to `get()` returns, we know that not only is the future value ready, but the thread specific variables on the other thread have also been destroyed.

Such mechanisms are provided for `boost::condition_variable`, `boost::promise` and `boost::packaged_task`. e.g.

```

void task_executor(boost::packaged_task<void(int)> task, int param)
{
    task.make_ready_at_thread_exit(param); // execute stored task
} // destroy thread specific and wake threads waiting on futures from task

```

Other threads can wait on a future obtained from the task without having to worry about races due to the execution of destructors of the thread specific objects from the task's thread.

```

boost::condition_variable cv;
boost::mutex m;
complex_type the_data;
bool data_ready;

void thread_func()
{
    boost::unique_lock<std::mutex> lk(m);
    the_data=find_the_answer();
    data_ready=true;
    boost::notify_all_at_thread_exit(cv, boost::move(lk));
} // destroy thread specific objects, notify cv, unlock mutex

void waiting_thread()
{
    boost::unique_lock<std::mutex> lk(m);
    while(!data_ready)
    {
        cv.wait(lk);
    }
    process(the_data);
}

```

The waiting thread is guaranteed that the thread specific objects used by `thread_func()` have been destroyed by the time `process(the_data)` is called. If the lock on `m` is released and re-acquired after setting `data_ready` and before calling `boost::notify_all_at_thread_exit()` then this does NOT hold, since the thread may return from the wait due to a spurious wake-up.

Executing asynchronously

`boost::async` is a simple way of running asynchronous tasks to make use of the available hardware concurrency. A call to `boost::async` returns a `boost::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.

A launch policy of either `boost::launch::async`, which asks the runtime to create an asynchronous thread, or `boost::launch::deferred`, which indicates you simply want to defer the function call until a later time (lazy evaluation). This argument is optional - if you omit it your function will use the default policy.

For example, consider computing the sum of a very large array. The first task is to not compute asynchronously when the overhead would be significant. The second task is to split the work into two pieces, one executed by the host thread and one executed asynchronously.

```
int parallel_sum(int* data, int size)
{
    int sum = 0;
    if ( size < 1000 )
        for ( int i = 0; i < size; ++i )
            sum += data[i];
    else {
        auto handle = boost::async(parallel_sum, data+size/2, size-size/2);
        sum += parallel_sum(data, size/2);
        sum += handle.get();
    }
    return sum;
}
```

Shared Futures

`shared_future` is designed to be shared between threads, that is to allow multiple concurrent get operations.

Multiple get

The second `get()` call in the following example is undefined.

```
void bad_second_use( type arg ) {

    auto ftr = async( [=]{ return work( arg ); } );
    if ( cond1 )
    {
        use1( ftr.get() );
    } else
    {
        use2( ftr.get() );
    }
    use3( ftr.get() ); // second use is undefined
}
```

Using a `shared_future` solves the issue

```
void good_second_use( type arg ) {

    shared_future<type> ftr = async( [=]{ return work( arg ); } );
    if ( cond1 )
    {
        use1( ftr.get() );
    } else
    {
        use2( ftr.get() );
    }
    use3( ftr.get() ); // second use is defined
}
```

share()

Namming the return type when declaring the `shared_future` is needed; `auto` is not available within template argument lists. Here `share()` could be used to simplify the code

```
void better_second_use( type arg ) {
    auto ftr = async( [=]{ return work( arg ); } ).share();
    if ( cond1 )
    {
        use1( ftr.get() );
    } else
    {
        use2( ftr.get() );
    }
    use3( ftr.get() ); // second use is defined
}
```

Writing on get()

The user can either read or write the future variable.

```
void write_to_get( type arg ) {
    auto ftr = async( [=]{ return work( arg ); } ).share();
    if ( cond1 )
    {
        use1( ftr.get() );
    } else
    {
        if ( cond2 )
            use2( ftr.get() );
        else
            ftr.get() = something(); // assign to non-const reference.
    }
    use3( ftr.get() ); // second use is defined
}
```

This works because the `shared_future<>::get()` function returns a non-const reference to the appropriate storage. Of course the access to this storage must be ensured by the user. The library doesn't ensure the access to the internal storage is thread safe.

There has been some work by the C++ standard committee on an `atomic_future` that behaves as an `atomic` variable, that is thread-safe, and a `shared_future` that can be shared between several threads, but there were not enough consensus and time to get it ready for C++11.

Making immediate futures easier

Some functions may know the value at the point of construction. In these cases the value is immediately available, but needs to be returned as a future or `shared_future`. By using `make_ready_future` a future can be created which holds a pre-computed result in its shared state.

Without these features it is non-trivial to create a future directly from a value. First a promise must be created, then the promise is set, and lastly the future is retrieved from the promise. This can now be done with one operation.

make_ready_future

This function creates a future for a given value. If no value is given then a `future<void>` is returned. This function is primarily useful in cases where sometimes, the return value is immediately available, but sometimes it is not. The example below illustrates, that in an error path the value is known immediately, however in other paths the function must return an eventual value represented as a future.

```

boost::future<int> compute(int x)
{
    if (x == 0) return boost::make_ready_future(0);
    if (x < 0) return boost::make_ready_future<int>(std::logic_error("Error"));
    boost::future<int> f1 = boost::async([]() { return x+1; });
    return f1;
}

```

There are two variations of this function. The first takes a value of any type, and returns a future of that type. The input value is passed to the shared state of the returned future. The second version takes no input and returns a `future<void>`.

Associating future continuations

In asynchronous programming, it is very common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. The current C++ standard does not allow one to register a continuation to a future. With `.then`, instead of waiting for the result, a continuation is "attached" to the asynchronous operation, which is invoked when the result is ready. Continuations registered using the `.then` function will help to avoid blocking waits or wasting threads on polling, greatly improving the responsiveness and scalability of an application.

`future.then()` provides the ability to sequentially compose two futures by declaring one to be the continuation of another. With `.then()` the antecedent future is ready (has a value or exception stored in the shared state) before the continuation starts as instructed by the lambda function.

In the example below the `future<string> f2` is registered to be a continuation of `future<int> f1` using the `.then()` member function. This operation takes a lambda function which describes how `f2` should proceed after `f1` is ready.

```

#include <boost/thread/future.hpp>
using namespace boost;
int main()
{
    future<int> f1 = async([]() { return 123; });
    future<string> f2 = f1.then([](future<int> f) { return f.get().to_string(); // here .get() ↴
won't block });
}

```

One key feature of this function is the ability to chain multiple asynchronous operations. In asynchronous programming, it's common to define a sequence of operations, in which each continuation executes only when the previous one completes. In some cases, the antecedent future produces a value that the continuation accepts as input. By using `future.then()`, creating a chain of continuations becomes straightforward and intuitive:

```
myFuture.then(...).then(...).then(...).
```

Some points to note are:

- Each continuation will not begin until the preceding has completed.
- If an exception is thrown, the following continuation can handle it in a try-catch block

Input Parameters:

- Lambda function: One option which can be considered is to take two functions, one for success and one for error handling. However this option has not been retained for the moment. The lambda function takes a future as its input which carries the exception through. This makes propagating exceptions straightforward. This approach also simplifies the chaining of continuations.
- Scheduler: Providing an overload to `.then`, to take a scheduler reference places great flexibility over the execution of the future in the programmer's hand. As described above, often taking a launch policy is not sufficient for powerful asynchronous operations. The lifetime of the scheduler must outlive the continuation.

- Launch policy: if the additional flexibility that the scheduler provides is not required.

Return values: The decision to return a future was based primarily on the ability to chain multiple continuations using `.then()`. This benefit of composability gives the programmer incredible control and flexibility over their code. Returning a `future` object rather than a `shared_future` is also a much cheaper operation thereby improving performance. A `shared_future` object is not necessary to take advantage of the chaining feature. It is also easy to go from a `future` to a `shared_future` when needed using `future::share()`.

Futures Reference

```
//#include <boost/thread/future.hpp>

namespace boost
{
    namespace future_state // EXTENSION
    {
        enum state {uninitialized, waiting, ready, moved};
    }

    enum class future_errc
    {
        broken_promise,
        future_already_retrieved,
        promise_already_satisfied,
        no_state
    };

    enum class launch
    {
        async = unspecified,
        deferred = unspecified,
        executor = unspecified,
        any = async | deferred
    };

    enum class future_status {
        ready, timeout, deferred
    };

    namespace system
    {
        template <>
        struct is_error_code_enum<future_errc> : public true_type {};

        error_code make_error_code(future_errc e);

        error_condition make_error_condition(future_errc e);
    }

    const system::error_category& future_category();

    class future_error;

    class exceptional_ptr;

    template <typename R>
    class promise;

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

    namespace container {
```

```

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

template <typename R>
class future;

template <typename R>
class shared_future;

template <typename S>
class packaged_task;
template <class S> void swap(packaged_task<S>&, packaged_task<S>&) noexcept;

template <class S, class Alloc>
struct uses_allocator<packaged_task <S>, Alloc>;

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

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

template<typename Iterator>
void wait_for_all(Iterator begin,Iterator end); // EXTENSION
template<typename F1,typename... FS>
void wait_for_all(F1& f1,FS&... fs); // EXTENSION

template<typename Iterator>
Iterator wait_for_any(Iterator begin,Iterator end); // EXTENSION
template<typename F1,typename... FS>
unsigned wait_for_any(F1& f1,FS&... fs); // EXTENSION

template <typename T>
future<typename decay<T>::type> make_future(T&& value); // DEPRECATED
future<void> make_future(); // DEPRECATED

template <typename T>
future<typename decay<T>::type> make_ready_future(T&& value); // EXTENSION
future<void> make_ready_future(); // EXTENSION
//template <typename T>
//future<T> make_ready_future(exception_ptr ex); // DEPRECATED
//template <typename T, typename E>
//future<T> make_ready_future(E ex); // DEPRECATED

exceptional_ptr make_exceptional(exception_ptr ex); // EXTENSION
template <typename E>
exceptional_ptr make_exceptional(E ex); // EXTENSION

```

```

exceptional_ptr make_exceptional(); // EXTENSION

template <typename T>
shared_future<typename decay<T>::type> make_shared_future(T&& value); // DEPRECATED
shared_future<void> make_shared_future(); // DEPRECATED

```

Enumeration `state`

```

namespace future_state
{
    enum state {uninitialized, waiting, ready, moved};
}

```

Enumeration `future_errc`

```

enum class future_errc
{
    broken_promise = implementation defined,
    future_already_retrieved = implementation defined,
    promise_already_satisfied = implementation defined,
    no_state = implementation defined
}

```

The `enum` values of `future_errc` are distinct and not zero.

Enumeration `launch`

```

enum class launch
{
    async = unspecified,
    deferred = unspecified,
    executor = unspecified,
    any = async | deferred
};

```

The `enum` type `launch` is a bitmask type with `launch::async` and `launch::deferred` denoting individual bits.

Specialization `is_error_code_enum<future_errc>`

```

namespace system
{
    template <>
    struct is_error_code_enum<future_errc> : public true_type {};
}

```

Non-member function `make_error_code()`

```

namespace system
{
    error_code make_error_code(future_errc e);
}

```

Returns: `error_code(static_cast<int>(e), future_category())`.

Non-member function `make_error_condition()`

```
namespace system
{
    error_condition make_error_condition(future_errc e);
}
```

Returns: `error_condition(static_cast<int>(e), future_category())`.

Non-member function `future_category()`

```
const system::error_category& future_category();
```

Returns: A reference to an object of a type derived from class `error_category`.

Notes: The object's `default_error_condition` and equivalent virtual functions behave as specified for the class `system::error_category`. The object's `name` virtual function returns a pointer to the string "future".

Class `future_error`

```
class future_error
    : public std::logic_error
{
public:
    future_error(system::error_code ec);

    const system::error_code& code() const no_except;
};
```

Constructor

```
future_error(system::error_code ec);
```

Effects: Constructs a `future_error`.

Postconditions: `code() == ec`

Throws: Nothing.

Member function `code()`

```
const system::error_code& code() const no_except;
```

Returns: The value of `ec` that was passed to the object's constructor.

Enumeration `future_status`

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


Class `exceptional_ptr` EXPERIMENTAL

```
class exceptional_ptr
{
public:
    exceptional_ptr();
    explicit exceptional_ptr(exception_ptr ex);
    template <class E>
    explicit exceptional_ptr(E&& ex);
};
```

Constructor

```
exceptional_ptr();
explicit exceptional_ptr(exception_ptr ex);
template <class E>
explicit exceptional_ptr(E&& ex);
```

- Effects:** The exception that is passed in to the constructor or the current exception if no parameter is moved into the constructed `exceptional_ptr` if it is an rvalue. Otherwise the exception is copied into the constructed `exceptional_ptr`.
- Postconditions:** `valid() == true && is_ready() = true && has_value() = false`
- Throws:** Nothing.

future class template

```

template <typename R>
class future
{
public:
    typedef R value_type; // EXTENSION
    future( future const& rhs) = delete;
    future& operator=( future const& rhs) = delete;

    future() noexcept;
    ~ future();

    // move support
    future( future && other) noexcept;
    future( future< future<R>>&& rhs); // EXTENSION
    future& operator=( future && other) noexcept;

    // factories
    shared_future<R> share();

    template<typename F>
        future<typename boost::result_of<F( future&)>::type>
    then(F&& func); // EXTENSION
    template<typename S, typename F>
        future<typename boost::result_of<F( future&)>::type>
    then(S& scheduler, F&& func); // EXTENSION NOT_YET_IMPLEMENTED
    template<typename F>
        future<typename boost::result_of<F( future&)>::type>
    then(launch policy, F&& func); // EXTENSION

    see below unwrap(); // EXTENSION
    future fallback_to(); // EXTENSION

    void swap( future& other) noexcept;

    // retrieving the value
    see below get();
    see below get_or(see below); // EXTENSION

    exception_ptr get_exception_ptr(); // EXTENSION

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

    // waiting for the result to be ready
    void wait() const;
    template <class Rep>
        future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
        future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;

#if defined BOOST_THREAD_USES_DATE_TIME
    template<typename Duration>

```

```

bool timed_wait(Duration const& rel_time) const; // DEPRECATED SINCE V3.0.0
bool timed_wait_until(boost::system_time const& abs_time) const; // DEPRECATED SINCE V3.0.0
#endif
typedef future_state::state state; // EXTENSION
state get_state() const; // EXTENSION
};

```

Default Constructor

```
future();
```

Effects: Constructs an uninitialized `future`.

Postconditions: `this->is_ready` returns false. `this->get_state()` returns `boost::future_state::uninitialized`.

Throws: Nothing.

Destructor

```
~ future();
```

Effects: Destroys `*this`.

Throws: Nothing.

Move Constructor

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

Effects: Constructs a new `future`, and transfers ownership of the shared state associated with `other` to `*this`.

Postconditions: `this->get_state()` returns the value of `other->get_state()` prior to the call. `other->get_state()` returns `boost::future_state::uninitialized`. If `other` was associated with a shared state, that result is now associated with `*this`. `other` is not associated with any shared state.

Throws: Nothing.

Notes: If the compiler does not support rvalue-references, this is implemented using the `boost.thread` move emulation.

Unwrap Move Constructor - EXTENSION

```
future( future< future<R>>&& other); // EXTENSION
```



Warning

This constructor is experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(

Requires: `other.valid()`.

[Effects:

Constructs a new `future`, and transfers ownership of the shared state associated with `other` and unwrapping the inner future (see `unwrap()`).

Postconditions:	<code>this->get_state()</code> returns the value of <code>other->get_state()</code> prior to the call. <code>other->get_state()</code> returns <code>boost::future_state::uninitialized</code> . The associated shared state is now unwrapped and the inner future shared state is associated with <code>*this</code> . <code>other</code> is not associated with any shared state, ! <code>other.valid()</code> .
Throws:	Nothing.
Notes:	If the compiler does not support rvalue-references, this is implemented using the <code>boost.thread</code> move emulation.

Move Assignment Operator

```
future& operator=( future && other );
```

Effects:	Transfers ownership of the shared state associated with <code>other</code> to <code>*this</code> .
Postconditions:	<code>this->get_state()</code> returns the value of <code>other->get_state()</code> prior to the call. <code>other->get_state()</code> returns <code>boost::future_state::uninitialized</code> . If <code>other</code> was associated with a shared state, that result is now associated with <code>*this</code> . <code>other</code> is not associated with any shared state. If <code>*this</code> was associated with an asynchronous result prior to the call, that result no longer has an associated <code>future</code> instance.
Throws:	Nothing.
Notes:	If the compiler does not support rvalue-references, this is implemented using the <code>boost.thread</code> move emulation.

Member function `swap()`

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

Effects:	Swaps ownership of the shared states associated with <code>other</code> and <code>*this</code> .
Postconditions:	<code>this->get_state()</code> returns the value of <code>other->get_state()</code> prior to the call. <code>other->get_state()</code> returns the value of <code>this->get_state()</code> prior to the call. If <code>other</code> was associated with a shared state, that result is now associated with <code>*this</code> , otherwise <code>*this</code> has no associated result. If <code>*this</code> was associated with a shared state, that result is now associated with <code>other</code> , otherwise <code>other</code> has no associated result.
Throws:	Nothing.

Member function `get()`

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

Effects:	If <code>*this</code> is associated with a shared state, waits until the result is ready as-if by a call to <code>boost::future<R>::wait()</code> , and retrieves the result (whether that is a value or an exception).
Returns:	<ul style="list-style-type: none"> - <code>future<R&>::get()</code> return the stored reference. - <code>future<void>::get()</code>, there is no return value. - <code>future<R>::get()</code> returns an rvalue-reference to the value stored in the shared state.
Postconditions:	<code>this->is_ready()</code> returns true. <code>this->get_state()</code> returns <code>boost::future_state::ready</code> .
Throws:	<ul style="list-style-type: none"> - <code>boost::future_uninitialized</code> if <code>*this</code> is not associated with a shared state. - <code>boost::thread_interrupted</code> if the result associated with <code>*this</code> is not ready at the point of the call, and the current thread is interrupted.

- Any exception stored in the shared state in place of a value.

Notes: `get()` is an *interruption point*.

Member function `get_or()` - EXTENSION

```
R get_or(R&& v); // EXTENSION
R get_or(R const& v); // EXTENSION
R& future<R>::get_or(R& v); // EXTENSION
void future<void>::get_or(); // EXTENSION
```



Warning

These functions are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(

Effects: If `*this` is associated with a shared state, waits until the result is ready as-if by a call to `boost::future<R>::wait()`, and depending on whether the shared state `has_value()` the retrieves the result.

Returns:

- `future<R>::get_or(v)` return the stored reference if `has_value()` and the passes parameter otherwise.
- `future<void>::get_or()`, there is no return value, but the function doesn't throws even if the shared state contained an exception.
- `future<R>::get_or(v)` returns an rvalue-reference to the value stored in the shared state if `has_value()` and an rvalue-reference build with the parameter `v`.

Postconditions: `this->is_ready()` returns true. `this->get_state()` returns `boost::future_state::ready`.

Throws: - `boost::future_uninitialized` if `*this` is not associated with a shared state.

Notes: `get_or()` is an *interruption point*.

Member function `wait()`

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

Effects: If `*this` is associated with a shared state, waits until the result is ready. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.

Throws:

- `boost::future_uninitialized` if `*this` is not associated with a shared state.
- `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.
- Any exception thrown by the *wait callback* if such a callback is called.

Postconditions: `this->is_ready()` returns true. `this->get_state()` returns `boost::future_state::ready`.

Notes: `wait()` is an *interruption point*.

Member function `timed_wait()` DEPRECATED SINCE V3.0.0

```
template<typename Duration>
bool timed_wait(Duration const& wait_duration);
```

**Warning**

DEPRECATED since 3.00.

Use instead [wait_for](#).

Effects:	If <code>*this</code> is associated with a shared state, waits until the result is ready, or the time specified by <code>wait_duration</code> has elapsed. If the result is not ready on entry, and the result has a <i>wait callback</i> set, that callback is invoked prior to waiting.
Returns:	<code>true</code> if <code>*this</code> is associated with a shared state, and that result is ready before the specified time has elapsed, <code>false</code> otherwise.
Throws:	<ul style="list-style-type: none"> - <code>boost::future_uninitialized</code> if <code>*this</code> is not associated with a shared state. - <code>boost::thread_interrupted</code> if the result associated with <code>*this</code> is not ready at the point of the call, and the current thread is interrupted. - Any exception thrown by the <i>wait callback</i> if such a callback is called.
Postconditions:	If this call returned <code>true</code> , then <code>this->is_ready()</code> returns <code>true</code> and <code>this->get_state()</code> returns <code>boost::future_state::ready</code> .
Notes:	<code>timed_wait()</code> is an <i>interruption point</i> . Duration must be a type that meets the <code>Boost.DateTime</code> time duration requirements.

Member function `timed_wait()` DEPRECATED SINCE V3.0.0

```
bool timed_wait(boost::system_time const& wait_timeout);
```

**Warning**

DEPRECATED since 3.00.

Use instead [wait_until](#).

Effects:	If <code>*this</code> is associated with a shared state, waits until the result is ready, or the time point specified by <code>wait_timeout</code> has passed. If the result is not ready on entry, and the result has a <i>wait callback</i> set, that callback is invoked prior to waiting.
Returns:	<code>true</code> if <code>*this</code> is associated with a shared state, and that result is ready before the specified time has passed, <code>false</code> otherwise.
Throws:	<ul style="list-style-type: none"> - <code>boost::future_uninitialized</code> if <code>*this</code> is not associated with a shared state. - <code>boost::thread_interrupted</code> if the result associated with <code>*this</code> is not ready at the point of the call, and the current thread is interrupted. - Any exception thrown by the <i>wait callback</i> if such a callback is called.
Postconditions:	If this call returned <code>true</code> , then <code>this->is_ready()</code> returns <code>true</code> and <code>this->get_state()</code> returns <code>boost::future_state::ready</code> .
Notes:	<code>timed_wait()</code> is an <i>interruption point</i> .

Member function `wait_for()`

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

- Effects:** If `*this` is associated with a shared state, waits until the result is ready, or the time specified by `wait_duration` has elapsed. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.
- Returns:**
- `future_status::deferred` if the shared state contains a deferred function. (Not implemented yet)
 - `future_status::ready` if the shared state is ready.
 - `future_status::timeout` if the function is returning because the relative timeout specified by `rel_time` has expired.
- Throws:**
- `boost::future_uninitialized` if `*this` is not associated with a shared state.
 - `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.
 - Any exception thrown by the *wait callback* if such a callback is called.
- Postconditions:** If this call returned true, then `this->is_ready()` returns true and `this->get_state()` returns `boost::future_state::ready`.
- Notes:** `wait_for()` is an *interruption point*. Duration must be a type that meets the Boost.DateTime time duration requirements.

Member function `wait_until()`

```
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

- Effects:** If `*this` is associated with a shared state, waits until the result is ready, or the time point specified by `wait_timeout` has passed. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.
- Returns:**
- `future_status::deferred` if the shared state contains a deferred function. (Not implemented yet)
 - `future_status::ready` if the shared state is ready.
 - `future_status::timeout` if the function is returning because the absolute timeout specified by `absl_time` has reached.
- Throws:**
- `boost::future_uninitialized` if `*this` is not associated with a shared state.
 - `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.
 - Any exception thrown by the *wait callback* if such a callback is called.
- Postconditions:** If this call returned true, then `this->is_ready()` returns true and `this->get_state()` returns `boost::future_state::ready`.
- Notes:** `wait_until()` is an *interruption point*.

Member function `valid()`

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

Returns: true if `*this` is associated with a shared state, false otherwise.

Throws: Nothing.

Member function `is_ready()` EXTENSION

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

Returns: true if `*this` is associated with a shared state and that result is ready for retrieval, false otherwise.

Throws: Nothing.

Member function `has_value()` EXTENSION

```
bool has_value() const;
```

Returns: true if `*this` is associated with a shared state, that result is ready for retrieval, and the result is a stored value, false otherwise.

Throws: Nothing.

Member function `has_exception()` EXTENSION

```
bool has_exception() const;
```

Returns: true if `*this` is associated with a shared state, that result is ready for retrieval, and the result is a stored exception, false otherwise.

Throws: Nothing.

Member function `get_exception_ptr()` EXTENSION

```
exception_ptr get_exception_ptr();
```

Effects: If `*this` is associated with a shared state, waits until the result is ready. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.

Returns: a `exception_ptr`, storing or not an exception.

Throws: Whatever `mutex::lock()`/`mutex::unlock()` can throw.

Member function `get_state()` EXTENSION

```
future_state::state get_state();
```

Effects: Determine the state of the shared state associated with `*this`, if any.

Returns: `boost::future_state::uninitialized` if `*this` is not associated with a shared state. `boost::future_state::ready` if the shared state associated with `*this` is ready for retrieval, `boost::future_state::waiting` otherwise.

Throws: Nothing.

Member function `share()`

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

Returns: `shared_future<R>(boost::move(*this)).`

Postconditions: `this->valid() == false.`

Member function `then()` - EXTENSION

```
template<typename F>
    future<typename boost::result_of<F( future&)>::type>
    then(F&& func); // EXTENSION
template<typename S, typename F>
    future<typename boost::result_of<F( future&)>::type>
    then(S& scheduler, F&& func); // EXTENSION NOT_YET_IMPLEMENTED
template<typename F>
    future<typename boost::result_of<F( future&)>::type>
    then(launch policy, F&& func); // EXTENSION
```

**Warning**

These functions are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(

**Note**

These functions are based on the [N3634 - Improvements to `std::future<T>` and related APIs](#) C++1y proposal by N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani.

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 a scheduler 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.

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 scheduler.
- When the scheduler or launch policy is not provided the continuation inherits the parent's launch policy or scheduler.
- If the parent was created with `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<typename boost::result_of<F(future)>` that refers to the shared state created by the continuation.

Postconditions:

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

Member function `unwrap()` EXTENSION

```
template <typename R2>
    future<R2>  future< future<R2>>::unwrap();  // EXTENSION
template <typename R2>
    boost::shared_future<R2>  future< boost::shared_future<R2>>::unwrap();  // EXTENSION
```

**Warning**

These functions are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(

**Note**

These functions are based on the [N3634 - Improvements to std::future<T> and related APIs](#) C++1y proposal by N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani.

Notes:	Removes the outermost future and returns a future with the associated state been a proxy of inner future.
Effects:	- Returns a future that becomes ready when the shared state of the inner future is ready.
Returns:	An object of type future with the associated state been a proxy of inner future.
Postconditions:	- The returned future has <code>valid() == true</code> regardless of the validity of the inner future.

shared_future class template

```

template <typename R>
class shared_future
{
public:
    typedef future_state::state state; // EXTENSION
    typedef R value_type; // EXTENSION

    shared_future() noexcept;
    ~shared_future();

    // copy support
    shared_future(shared_future const& other);
    shared_future& operator=(shared_future const& other);

    // move support
    shared_future(shared_future && other) noexcept;
    shared_future( future<R> && other) noexcept;
    shared_future& operator=(shared_future && other) noexcept;
    shared_future& operator=( future<R> && other) noexcept;

    // factories
    template<typename F>
        future<typename boost::result_of<F(shared_future&)>::type>
        then(F&& func); // EXTENSION
    template<typename S, typename F>
        future<typename boost::result_of<F(shared_future&)>::type>
        then(S& scheduler, F&& func); // EXTENSION NOT_YET_IMPLEMENTED
    template<typename F>
        future<typename boost::result_of<F(shared_future&)>::type>
        then(launch policy, F&& func); // EXTENSION

    void swap(shared_future& other);

    // retrieving the value
    see below get();

    exception_ptr get_exception_ptr(); // EXTENSION

    // functions to check state, and wait for ready
    bool valid() const noexcept;
    bool is_ready() const noexcept; // EXTENSION
    bool has_exception() const noexcept; // EXTENSION
    bool has_value() const noexcept; // EXTENSION

    // waiting for the result to be ready
    void wait() const;
    template <class Rep, class Period>
        future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
        future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;

#if defined BOOST_THREAD_USES_DATE_TIME || defined BOOST_THREAD_DONT_USE_CHRONO
    template<typename Duration>
        bool timed_wait(Duration const& rel_time) const; // DEPRECATED SINCE V3.0.0
        bool timed_wait_until(boost::system_time const& abs_time) const; // DEPRECATED SINCE V3.0.0
#endif
    state get_state() const noexcept; // EXTENSION
};

```

Default Constructor

```
shared_future();
```

Effects: Constructs an uninitialized `shared_future`.

Postconditions: `this->is_ready` returns false. `this->get_state()` returns `boost::future_state::uninitialized`.

Throws: Nothing.

Member function `get()`

```
const R& get();
R& get();
void get();
```

Effects: If `*this` is associated with a shared state, waits until the result is ready as-if by a call to `boost::shared_future<R>::wait()`, and returns a const reference to the result.

Returns:

- `shared_future<R>::get()` return the stored reference.
- `shared_future<void>::get()`, there is no return value.
- `shared_future<R>::get()` returns a const reference to the value stored in the shared state.

Throws:

- `boost::future_uninitialized` if `*this` is not associated with a shared state.
- `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.

Notes: `get()` is an *interruption point*.

Member function `wait()`

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

Effects: If `*this` is associated with a shared state, waits until the result is ready. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.

Throws:

- `boost::future_uninitialized` if `*this` is not associated with a shared state.
- `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.
- Any exception thrown by the *wait callback* if such a callback is called.

Postconditions: `this->is_ready()` returns true. `this->get_state()` returns `boost::future_state::ready`.

Notes: `wait()` is an *interruption point*.

Member function `timed_wait()`

```
template<typename Duration>
bool timed_wait(Duration const& wait_duration);
```

- Effects:** If `*this` is associated with a shared state, waits until the result is ready, or the time specified by `wait_duration` has elapsed. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.
- Returns:** true if `*this` is associated with a shared state, and that result is ready before the specified time has elapsed, false otherwise.
- Throws:**
- `boost::future_uninitialized` if `*this` is not associated with a shared state.
 - `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.
 - Any exception thrown by the *wait callback* if such a callback is called.
- Postconditions:** If this call returned true, then `this->is_ready()` returns true and `this->get_state()` returns `boost::future_state::ready`.
- Notes:** `timed_wait()` is an *interruption point*. `Duration` must be a type that meets the `Boost.DateTime` time duration requirements.

Member function `timed_wait()`

```
bool timed_wait(boost::system_time const& wait_timeout);
```

- Effects:** If `*this` is associated with a shared state, waits until the result is ready, or the time point specified by `wait_timeout` has passed. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.
- Returns:** true if `*this` is associated with a shared state, and that result is ready before the specified time has passed, false otherwise.
- Throws:**
- `boost::future_uninitialized` if `*this` is not associated with a shared state.
 - `boost::thread_interrupted` if the result associated with `*this` is not ready at the point of the call, and the current thread is interrupted.
 - Any exception thrown by the *wait callback* if such a callback is called.
- Postconditions:** If this call returned true, then `this->is_ready()` returns true and `this->get_state()` returns `boost::future_state::ready`.
- Notes:** `timed_wait()` is an *interruption point*.

Member function `wait_for()`

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

- Effects:** If `*this` is associated with a shared state, waits until the result is ready, or the time specified by `wait_duration` has elapsed. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.
- Returns:**
- `future_status::deferred` if the shared state contains a deferred function. (Not implemented yet)

- `future_status::ready` if the shared state is ready.
- `future_status::timeout` if the function is returning because the relative timeout specified by `rel_time` has expired.

Throws:	<ul style="list-style-type: none"> - <code>boost::future_uninitialized</code> if <code>*this</code> is not associated with a shared state. - <code>boost::thread_interrupted</code> if the result associated with <code>*this</code> is not ready at the point of the call, and the current thread is interrupted. - Any exception thrown by the <i>wait callback</i> if such a callback is called.
Postconditions:	If this call returned <code>true</code> , then <code>this->is_ready()</code> returns <code>true</code> and <code>this->get_state()</code> returns <code>boost::future_state::ready</code> .
Notes:	<code>timed_wait()</code> is an <i>interruption point</i> . Duration must be a type that meets the <code>Boost.DateTime</code> time duration requirements.

Member function `wait_until()`

```
template <class Clock, class Duration>
future_status wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

Effects:	If <code>*this</code> is associated with a shared state, waits until the result is ready, or the time point specified by <code>wait_timeout</code> has passed. If the result is not ready on entry, and the result has a <i>wait callback</i> set, that callback is invoked prior to waiting.
Returns:	<ul style="list-style-type: none"> - <code>future_status::deferred</code> if the shared state contains a deferred function. (Not implemented yet) - <code>future_status::ready</code> if the shared state is ready. - <code>future_status::timeout</code> if the function is returning because the absolute timeout specified by <code>absl_time</code> has reached.
Throws:	<ul style="list-style-type: none"> - <code>boost::future_uninitialized</code> if <code>*this</code> is not associated with a shared state. - <code>boost::thread_interrupted</code> if the result associated with <code>*this</code> is not ready at the point of the call, and the current thread is interrupted. - Any exception thrown by the <i>wait callback</i> if such a callback is called.
Postconditions:	If this call returned <code>true</code> , then <code>this->is_ready()</code> returns <code>true</code> and <code>this->get_state()</code> returns <code>boost::future_state::ready</code> .
Notes:	<code>timed_wait()</code> is an <i>interruption point</i> .

Member function `valid()`

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

Returns:	<code>true</code> if <code>*this</code> is associated with a shared state, <code>false</code> otherwise.
Throws:	Nothing.

Member function `is_ready()` EXTENSION

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

Returns:	<code>true</code> if <code>*this</code> is associated with a shared state, and that result is ready for retrieval, <code>false</code> otherwise.
----------	--

Throws: Whatever `mutex::lock()`/`mutex::unlock()` can throw.

Member function `has_value()` EXTENSION

```
bool has_value() const;
```

Returns: true if `*this` is associated with a shared state, that result is ready for retrieval, and the result is a stored value, false otherwise.

Throws: Whatever `mutex::lock()`/`mutex::unlock()` can throw.

Member function `has_exception()` EXTENSION

```
bool has_exception() const;
```

Returns: true if `*this` is associated with a shared state, that result is ready for retrieval, and the result is a stored exception, false otherwise.

Throws: Whatever `mutex::lock()`/`mutex::unlock()` can throw.

Member function `get_exception_ptr()` EXTENSION

```
exception_ptr get_exception_ptr();
```

Effects: If `*this` is associated with a shared state, waits until the result is ready. If the result is not ready on entry, and the result has a *wait callback* set, that callback is invoked prior to waiting.

Returns: a `exception_ptr`, storing or not an exception.

Throws: Whatever `mutex::lock()`/`mutex::unlock()` can throw.

Member function `get_state()` EXTENSION

```
future_state::state get_state();
```

Effects: Determine the state of the shared state associated with `*this`, if any.

Returns: `boost::future_state::uninitialized` if `*this` is not associated with a shared state. `boost::future_state::ready` if the shared state associated with `*this` is ready for retrieval, `boost::future_state::waiting` otherwise.

Throws: Whatever `mutex::lock()`/`mutex::unlock()` can throw.

Member function `then()` EXTENSION

```
template<typename F>
    future<typename boost::result_of<F(shared_future&)>::type>
    then(F&& func); // EXTENSION
template<typename S, typename F>
    future<typename boost::result_of<F(shared_future&)>::type>
    then(S& scheduler, F&& func); // EXTENSION NOT_YET_IMPLEMENTED
template<typename F>
    future<typename boost::result_of<F(shared_future&)>::type>
    then(launch policy, F&& func); // EXTENSION
```



Warning

These functions are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(



Note

These functions are based on the [N3634 - Improvements to std::future<T> and related APIs](#) C++1y proposal by N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani.

- 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 a scheduler 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.
- 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 scheduler.
 - When the scheduler or launch policy is not provided the continuation inherits the parent's launch policy or scheduler.
 - If the parent was created with `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<typename boost::result_of<F(shared_future)>` 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.

promise class template

```
template <typename R>
class promise
{
public:
    typedef R value_type;    // EXTENSION

    promise();
    template <class Allocator>
    promise(allocator_arg_t, Allocator a);
    promise & operator=(promise const& rhs) = delete;
    promise(promise const& rhs) = delete;
    ~promise();

    // Move support
    promise(promise && rhs) noexcept;
    promise & operator=(promise&& rhs) noexcept;

    void swap(promise& other) noexcept;
    // Result retrieval
    future<R> get_future();

    // Set the value
    void set_value(see below);
    void set_exception(boost::exception_ptr e);
    template <typename E>
    void set_exception(E e);    // EXTENSION

    // setting the result with deferred notification
    void set_value_at_thread_exit(see below);
    void set_exception_at_thread_exit(exception_ptr p);
    template <typename E>
    void set_exception_at_thread_exit(E p);    // EXTENSION

    template<typename F>
    void set_wait_callback(F f);    // EXTENSION
};
```

Default Constructor

```
promise();
```

Effects: Constructs a new `boost::promise` with no associated result.

Throws: Nothing.

Allocator Constructor

```
template <class Allocator>
promise(allocator_arg_t, Allocator a);
```

Effects: Constructs a new `boost::promise` with no associated result using the allocator `a`.

Throws: Nothing.

Notes: Available only if `BOOST_THREAD_FUTURE_USES_ALLOCATORS` is defined.

Move Constructor

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

Effects: Constructs a new `boost::promise`, and transfers ownership of the result associated with `other` to `*this`, leaving `other` with no associated result.

Throws: Nothing.

Notes: If the compiler does not support rvalue-references, this is implemented using the `boost.thread` move emulation.

Move Assignment Operator

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

Effects: Transfers ownership of the result associated with `other` to `*this`, leaving `other` with no associated result. If there was already a result associated with `*this`, and that result was not *ready*, sets any futures associated with that result to *ready* with a `boost::broken_promise` exception as the result.

Throws: Nothing.

Notes: If the compiler does not support rvalue-references, this is implemented using the `boost.thread` move emulation.

Destructor

```
~promise();
```

Effects: Destroys `*this`. If there was a result associated with `*this`, and that result is not *ready*, sets any futures associated with that task to *ready* with a `boost::broken_promise` exception as the result.

Throws: Nothing.

Member Function `get_future()`

```
future<R> get_future();
```

Effects: If `*this` was not associated with a result, allocate storage for a new shared state and associate it with `*this`. Returns a `future` associated with the result associated with `*this`.

Throws: `boost::future_already_retrieved` if the future associated with the task has already been retrieved. `std::bad_alloc` if any memory necessary could not be allocated.

Member Function `set_value()`

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

Effects: - If `BOOST_THREAD_PROVIDES_PROMISE_LAZY` is defined and if `*this` was not associated with a result, allocate storage for a new shared state and associate it with `*this`.

- Store the value `r` in the shared state associated with `*this`. Any threads blocked waiting for the asynchronous result are woken.

Postconditions: All futures waiting on the shared state are *ready* and `boost::future<R>::has_value()` or `boost::shared_future<R>::has_value()` for those futures shall return *true*.

Throws:

- `boost::promise_already_satisfied` if the result associated with **this* is already *ready*.
- `boost::broken_promise` if **this* has no shared state.
- `std::bad_alloc` if the memory required for storage of the result cannot be allocated.
- Any exception thrown by the copy or move-constructor of *R*.

Member Function `set_exception()`

```
void set_exception(boost::exception_ptr e);
template <typename E>
void set_exception(E e); // EXTENSION
```

Effects:

- If `BOOST_THREAD_PROVIDES_PROMISE_LAZY` is defined and if **this* was not associated with a result, allocate storage for a new shared state and associate it with **this*.
- Store the exception *e* in the shared state associated with **this*. Any threads blocked waiting for the asynchronous result are woken.

Postconditions: All futures waiting on the shared state are *ready* and `boost::future<R>::has_exception()` or `boost::shared_future<R>::has_exception()` for those futures shall return *true*.

Throws:

- `boost::promise_already_satisfied` if the result associated with **this* is already *ready*.
- `boost::broken_promise` if **this* has no shared state.
- `std::bad_alloc` if the memory required for storage of the result cannot be allocated.

Member Function `set_value_at_thread_exit()`

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

Effects: Stores the value *r* in the shared state without making that state *ready* immediately. Schedules that state to be made *ready* when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws:

- `boost::promise_already_satisfied` if the result associated with **this* is already *ready*.
- `boost::broken_promise` if **this* has no shared state.
- `std::bad_alloc` if the memory required for storage of the result cannot be allocated.
- Any exception thrown by the copy or move-constructor of *R*.

Member Function `set_exception_at_thread_exit()`

```
void set_exception_at_thread_exit(boost::exception_ptr e);
template <typename E>
void set_exception_at_thread_exit(E p); // EXTENSION
```

- Effects:** Stores the exception pointer `p` in the shared state without making that state ready immediately. Schedules that state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.
- Postconditions:** All futures waiting on the shared state are *ready* and `boost::future<R>::has_exception()` or `boost::shared_future<R>::has_exception()` for those futures shall return *true*.
- Throws:**
- `boost::promise_already_satisfied` if the result associated with `*this` is already *ready*.
 - `boost::broken_promise` if `*this` has no shared state.
 - `std::bad_alloc` if the memory required for storage of the result cannot be allocated.

Member Function `set_wait_callback()` EXTENSION

```
template<typename F>
void set_wait_callback(F f);
```

- Preconditions:** The expression `f(t)` where `t` is a lvalue of type `boost::promise` shall be well-formed. Invoking a copy of `f` shall have the same effect as invoking `f`.
- Effects:** Store a copy of `f` with the shared state associated with `*this` as a *wait callback*. This will replace any existing wait callback store alongside that result. If a thread subsequently calls one of the wait functions on a `future` or `boost::shared_future` associated with this result, and the result is not *ready*, `f(*this)` shall be invoked.
- Throws:** `std::bad_alloc` if memory cannot be allocated for the required storage.

packaged_task class template

```

template<typename S>
class packaged_task;
template<typename R
    , class... ArgTypes
>
class packaged_task<R(ArgTypes)>
{
public:
    packaged_task(packaged_task const&) = delete;
    packaged_task& operator=(packaged_task const&) = delete;

    // construction and destruction
    packaged_task() noexcept;

    explicit packaged_task(R(*f)(ArgTypes...));

    template <class F>
    explicit packaged_task(F&& f);

    template <class Allocator>
    packaged_task(allocator_arg_t, Allocator a, R(*f)(ArgTypes...));
    template <class F, class Allocator>
    packaged_task(allocator_arg_t, Allocator a, F&& f);

    ~packaged_task()
    {}

    // move support
    packaged_task(packaged_task&& other) noexcept;
    packaged_task& operator=(packaged_task&& other) noexcept;

    void swap(packaged_task& other) noexcept;

    bool valid() const noexcept;
    // result retrieval
    future<R> get_future();

    // execution
    void operator()(ArgTypes... );
    void make_ready_at_thread_exit(ArgTypes...);

    void reset();
    template<typename F>
    void set_wait_callback(F f); // EXTENSION
};

```

Task Constructor

```

packaged_task(R(*f)(ArgTypes...));

template<typename F>
packaged_task(F&&f);

```

- Preconditions:** `f()` is a valid expression with a return type convertible to `R`. Invoking a copy of `f` must behave the same as invoking `f`.
- Effects:** Constructs a new `boost::packaged_task` with `boost::forward<F>(f)` stored as the associated task.
- Throws:** - Any exceptions thrown by the copy (or move) constructor of `f`.

- `std::bad_alloc` if memory for the internal data structures could not be allocated.

Notes: The `R(*f)(ArgTypes...)` overload to allow passing a function without needing to use `&`.

Remark: This constructor doesn't participate in overload resolution if `decay<F>::type` is the same type as `boost::packaged_task<R>`.

Allocator Constructor

```
template <class Allocator>
packaged_task(allocator_arg_t, Allocator a, R(*f)(ArgTypes...));
template <class F, class Allocator>
packaged_task(allocator_arg_t, Allocator a, F&& f);
```

Preconditions: `f()` is a valid expression with a return type convertible to `R`. Invoking a copy of `f` shall behave the same as invoking `f`.

Effects: Constructs a new `boost::packaged_task` with `boost::forward<F>(f)` stored as the associated task using the allocator `a`.

Throws: Any exceptions thrown by the copy (or move) constructor of `f`. `std::bad_alloc` if memory for the internal data structures could not be allocated.

Notes: Available only if `BOOST_THREAD_FUTURE_USES_ALLOCATORS` is defined.

Notes: The `R(*f)(ArgTypes...)` overload to allow passing a function without needing to use `&`.

Move Constructor

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

Effects: Constructs a new `boost::packaged_task`, and transfers ownership of the task associated with `other` to `*this`, leaving `other` with no associated task.

Throws: Nothing.

Notes: If the compiler does not support rvalue-references, this is implemented using the `boost.thread` move emulation.

Move Assignment Operator

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

Effects: Transfers ownership of the task associated with `other` to `*this`, leaving `other` with no associated task. If there was already a task associated with `*this`, and that task has not been invoked, sets any futures associated with that task to *ready* with a `boost::broken_promise` exception as the result.

Throws: Nothing.

Notes: If the compiler does not support rvalue-references, this is implemented using the `boost.thread` move emulation.

Destructor

```
~packaged_task();
```

Effects: Destroys `*this`. If there was a task associated with `*this`, and that task has not been invoked, sets any futures associated with that task to *ready* with a `boost::broken_promise` exception as the result.

Throws: Nothing.

Member Function `get_future()`

```
future<R> get_future();
```

Effects: Returns a `future` associated with the result of the task associated with `*this`.

Throws: `boost::task_moved` if ownership of the task associated with `*this` has been moved to another instance of `boost::packaged_task`. `boost::future_already_retrieved` if the future associated with the task has already been retrieved.

Member Function `operator()()`

```
void operator()();
```

Effects: Invoke the task associated with `*this` and store the result in the corresponding future. If the task returns normally, the return value is stored as the shared state, otherwise the exception thrown is stored. Any threads blocked waiting for the shared state associated with this task are woken.

Postconditions: All futures waiting on the shared state are *ready*

Throws:

- `boost::task_moved` if ownership of the task associated with `*this` has been moved to another instance of `boost::packaged_task`.
- `boost::task_already_started` if the task has already been invoked.

Member Function `make_ready_at_thread_exit()`

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

Effects: Invoke the task associated with `*this` and store the result in the corresponding future. If the task returns normally, the return value is stored as the shared state, otherwise the exception thrown is stored. In either case, this is done without making that state ready immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws:

- `boost::task_moved` if ownership of the task associated with `*this` has been moved to another instance of `boost::packaged_task`.
- `boost::task_already_started` if the task has already been invoked.

Member Function `reset()`

```
void reset();
```

Effects: Reset the state of the `packaged_task` so that it can be called again.

Throws: `boost::task_moved` if ownership of the task associated with `*this` has been moved to another instance of `boost::packaged_task`.

Member Function `set_wait_callback()` EXTENSION

```
template<typename F>
void set_wait_callback(F f);
```

Preconditions: The expression `f(t)` where `t` is a lvalue of type `boost::packaged_task` shall be well-formed. Invoking a copy of `f` shall have the same effect as invoking `f`

Effects: Store a copy of `f` with the task associated with `*this` as a *wait callback*. This will replace any existing wait callback store alongside that task. If a thread subsequently calls one of the wait functions on a `future` or `boost::shared_future` associated with this task, and the result of the task is not *ready*, `f(*this)` shall be invoked.

Throws: `boost::task_moved` if ownership of the task associated with `*this` has been moved to another instance of `boost::packaged_task`.

Non-member function `decay_copy()`

```
template <class T>
typename decay<T>::type decay_copy(T&& v)
{
    return boost::forward<T>(v);
}
```

Non-member function `async()`

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.



Warning

`async(launch::deferred, F)` is NOT YET IMPLEMENTED!

Non-Variadic variant

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

Requires:

```
decay_copy(boost::forward<F>(f))()
```

shall be a valid expression.

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`.

The second function 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 `decay_copy(boost::forward<F>(f))()` 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 `decay_copy(boost::forward<F>(f))()` 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(boost::forward<F>(f))` in the shared state. This copy of `f` constitute a deferred function. Invocation of the deferred function evaluates `boost::move(g)()` where `g` is the stored value of `decay_copy(boost::forward<F>(f))`. The shared state is not made ready until the function has completed. The first call to a non-timed waiting function 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 `boost::move(g)()` 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.)
- if no valid launch policy is provided the behaviour is undefined.

Returns: An object of type `future<typename result_of<typename decay<F>::type()>::type>` that refers to the shared state created by this call to `async`.

Synchronization: Regardless of the provided policy argument,

- the invocation of `async` synchronizes with the invocation of `f`. (Note: This statement applies even when the corresponding future object is moved to another thread.); and
- the completion of the function `f` is sequenced before the shared state is made ready. (Note: `f` might not be called at all, so its completion might never happen.)

If the implementation chooses the `launch::async` policy,

- a call to a non-timed waiting function on an asynchronous return object that shares the shared state created by this `async` call shall block until the associated thread has completed, as if joined, or else time out;
- the associated thread completion synchronizes with the return from the first function that successfully detects the ready status of the shared state or with the return from the last function that releases the shared state, whichever happens first.

Throws: `system_error` if policy is `launch::async` and the implementation is unable to start a new thread.

Error conditions: - `resource_unavailable_try_again` - if policy is `launch::async` and the system is unable to start a new thread.

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

Variadic variant

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



Warning

the variadic prototype is provided only on C++11 compilers supporting rvalue references, variadic templates, decltype and a standard library providing `<tuple>` (waiting for a `boost::tuple` that is move aware), and `BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK` is defined.

Requires: `F` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements.

invoke (decay_copy (boost::forward<F>(f)), decay_copy (boost::forward<Args>(args))...)

shall be a valid expression.

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 function 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(forward<F>(f)), decay_copy (forward<Args>(args))...)` 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(boost::forward<F>(f)), decay_copy (boost::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(forward<F>(f))` and `decay_copy (forward<Args>(args))...` in the shared state. These copies of `f` and `args` constitute a deferred function. Invocation of the deferred function evaluates `invoke(move(g), move(xyz))` where `g` is the stored value of `decay_copy(forward<F>(f))` and `xyz` is the stored copy of `decay_copy (forward<Args>(args))...`. The shared state is not made ready until the function has completed. The first call to a non-timed waiting function 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(move(g), move(xyz))` begins, the function is no longer considered deferred.

- if no valid launch policy is provided the behaviour is undefined.

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.

Returns:

An object of type `future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>` that refers to the shared state created by this call to `async`.

Synchronization:

Regardless of the provided policy argument,

- the invocation of `async` synchronizes with the invocation of `f`. (Note: This statement applies even when the corresponding future object is moved to another thread.); and

- the completion of the function `f` is sequenced before the shared state is made ready. (Note: `f` might not be called at all, so its completion might never happen.)

If the implementation chooses the `launch::async` policy,

- a call to a waiting function on an asynchronous return object that shares the shared state created by this `async` call shall block until the associated thread has completed, as if joined, or else time out;

- the associated thread completion synchronizes with the return from the first function that successfully detects the ready status of the shared state or with the return from the last function that releases the shared state, whichever happens first.

Throws:

`system_error` if policy is `launch::async` and the implementation is unable to start a new thread.

Error conditions:

- `resource_unavailable_try_again` - if policy is `launch::async` and the system is unable to start a new thread.

Remarks:

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

Non-member function `wait_for_any()` - EXTENSION

```
template<typename Iterator>
Iterator wait_for_any(Iterator begin,Iterator end); // EXTENSION

template<typename F1,typename F2>
unsigned wait_for_any(F1& f1,F2& f2); // EXTENSION

template<typename F1,typename F2,typename F3>
unsigned wait_for_any(F1& f1,F2& f2,F3& f3); // EXTENSION

template<typename F1,typename F2,typename F3,typename F4>
unsigned wait_for_any(F1& f1,F2& f2,F3& f3,F4& f4); // EXTENSION

template<typename F1,typename F2,typename F3,typename F4,typename F5>
unsigned wait_for_any(F1& f1,F2& f2,F3& f3,F4& f4,F5& f5); // EXTENSION
```

- Preconditions:** The types `Fn` shall be specializations of `future` or `boost::shared_future`, and `Iterator` shall be a forward iterator with a `value_type` which is a specialization of `future` or `boost::shared_future`.
- Effects:** Waits until at least one of the specified futures is *ready*.
- Returns:** The range-based overload returns an `Iterator` identifying the first future in the range that was detected as *ready*. The remaining overloads return the zero-based index of the first future that was detected as *ready* (first parameter => 0, second parameter => 1, etc.).
- Throws:** `boost::thread_interrupted` if the current thread is interrupted. Any exception thrown by the *wait callback* associated with any of the futures being waited for. `std::bad_alloc` if memory could not be allocated for the internal wait structures.
- Notes:** `wait_for_any()` is an *interruption point*.

Non-member function `wait_for_all()` - EXTENSION

```
template<typename Iterator>
void wait_for_all(Iterator begin,Iterator end); // EXTENSION

template<typename F1,typename F2>
void wait_for_all(F1& f1,F2& f2); // EXTENSION

template<typename F1,typename F2,typename F3>
void wait_for_all(F1& f1,F2& f2,F3& f3); // EXTENSION

template<typename F1,typename F2,typename F3,typename F4>
void wait_for_all(F1& f1,F2& f2,F3& f3,F4& f4); // EXTENSION

template<typename F1,typename F2,typename F3,typename F4,typename F5>
void wait_for_all(F1& f1,F2& f2,F3& f3,F4& f4,F5& f5); // EXTENSION
```

- Preconditions:** The types `Fn` shall be specializations of `future` or `boost::shared_future`, and `Iterator` shall be a forward iterator with a `value_type` which is a specialization of `future` or `boost::shared_future`.
- Effects:** Waits until all of the specified futures are *ready*.
- Throws:** Any exceptions thrown by a call to `wait()` on the specified futures.
- Notes:** `wait_for_all()` is an *interruption point*.

Non-member function `make_ready_future()` EXTENSION

```
template <typename T>
future<typename decay<T>::type> make_ready_future(T&& value); // EXTENSION
future<void> make_ready_future(); // EXTENSION
template <typename T>
future<T> make_ready_future(exception_ptr ex); // DEPRECATED
template <typename T, typename E>
future<T> make_ready_future(E ex); // DEPRECATED
```

- Effects:
- value prototype: The value that is passed into 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.
 - exception: The exception that is passed into the function is copied to the shared state of the returned function.
 - .
- Returns:
- a ready future with the value set with value
 - a ready future with the exception set with ex
 - a ready future<void> with the value set (void).
- Postcondition:
- Returned future, `valid() == true`
 - Returned future, `is_ready() = true`
 - Returned future, `has_value() = true` or `has_exception()` depending on the prototype.

Non-member function `make_exceptional()` EXTENSION

```
exceptional_ptr make_exceptional(exception_ptr ex); // EXTENSION
template <typename E>
exceptional_ptr make_exceptional(E ex); // EXTENSION
exceptional_ptr make_exceptional(); // EXTENSION
```

- Effects:
- The exception that is passed in to the function or the current exception if no parameter is given is moved into the returned `exceptional_ptr` if it is an rvalue. Otherwise the exception is copied into the returned `exceptional_ptr`.
- Returns:
- An `exceptional_ptr` instance implicitly convertible to a `future<T>`

Non-member function `make_future()` DEPRECATED

```
template <typename T>
future<typename decay<T>::type> make_future(T&& value); // DEPRECATED
future<void> make_future(); // DEPRECATED
```

- Effects:
- The value that is passed into 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`
- See:
- `make_ready_future()`

Non-member function `make_shared_future()` DEPRECATED

```
template <typename T>
shared_future<typename decay<T>::type> make_shared_future(T&& value); // DEPRECATED
shared_future<void> make_shared_future(); // DEPRECATED
```

- 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:**
- `shared_future<T>`, if function is given a value of type `T`
 - `shared_future<void>`, if the function is not given any inputs.
- Postcondition:**
- Returned `shared_future<T>`, `valid() == true`
 - Returned `shared_future<T>`, `is_ready() = true`
- See:** `make_ready_future()` and `future<>::share()`

Thread Local Storage

Synopsis

Thread local storage allows multi-threaded applications to have a separate instance of a given data item for each thread. Where a single-threaded application would use static or global data, this could lead to contention, deadlock or data corruption in a multi-threaded application. One example is the C `errno` variable, used for storing the error code related to functions from the Standard C library. It is common practice (and required by POSIX) for compilers that support multi-threaded applications to provide a separate instance of `errno` for each thread, in order to avoid different threads competing to read or update the value.

Though compilers often provide this facility in the form of extensions to the declaration syntax (such as `__declspec(thread)` or `thread` annotations on `static` or namespace-scope variable declarations), such support is non-portable, and is often limited in some way, such as only supporting POD types.

Portable thread-local storage with `boost::thread_specific_ptr`

`boost::thread_specific_ptr` provides a portable mechanism for thread-local storage that works on all compilers supported by **Boost.Thread**. Each instance of `boost::thread_specific_ptr` represents a pointer to an object (such as `errno`) where each thread must have a distinct value. The value for the current thread can be obtained using the `get()` member function, or by using the `*` and `->` pointer dereference operators. Initially the pointer has a value of `NULL` in each thread, but the value for the current thread can be set using the `reset()` member function.

If the value of the pointer for the current thread is changed using `reset()`, then the previous value is destroyed by calling the cleanup routine. Alternatively, the stored value can be reset to `NULL` and the prior value returned by calling the `release()` member function, allowing the application to take back responsibility for destroying the object.

Cleanup at thread exit

When a thread exits, the objects associated with each `boost::thread_specific_ptr` instance are destroyed. By default, the object pointed to by a pointer `p` is destroyed by invoking `delete p`, but this can be overridden for a specific instance of `boost::thread_specific_ptr` by providing a cleanup routine to the constructor. In this case, the object is destroyed by invoking `func(p)` where `func` is the cleanup routine supplied to the constructor. The cleanup functions are called in an unspecified order. If a cleanup routine sets the value of associated with an instance of `boost::thread_specific_ptr` that has already been cleaned up, that value is added to the cleanup list. Cleanup finishes when there are no outstanding instances of `boost::thread_specific_ptr` with values.

Note: on some platforms, cleanup of thread-specific data is not performed for threads created with the platform's native API. On those platforms such cleanup is only done for threads that are started with `boost::thread` unless `boost::on_thread_exit()` is called manually from that thread.

Rationale about the nature of the key

Boost.Thread uses the address of the `thread_specific_ptr` instance as key of the thread specific pointers. This avoids to create/destroy a key which will need a lock to protect from race conditions. This has a little performance liability, as the access must be done using an associative container.

Class `thread_specific_ptr`

```
// #include <boost/thread/tss.hpp>

namespace boost
{
    template <typename T>
    class thread_specific_ptr
    {
    public:
        thread_specific_ptr();
        explicit thread_specific_ptr(void (*cleanup_function)(T*));
        ~thread_specific_ptr();

        T* get() const;
        T* operator->() const;
        T& operator*() const;

        T* release();
        void reset(T* new_value=0);
    };
}
```

`thread_specific_ptr();`

Requires: `delete this->get()` is well-formed.

Effects: Construct a `thread_specific_ptr` object for storing a pointer to an object of type `T` specific to each thread. The default delete-based cleanup function will be used to destroy any thread-local objects when `reset()` is called, or the thread exits.

Throws: `boost::thread_resource_error` if an error occurs.

`explicit thread_specific_ptr(void (*cleanup_function)(T*));`

Requires: `cleanup_function(this->get())` does not throw any exceptions.

Effects: Construct a `thread_specific_ptr` object for storing a pointer to an object of type `T` specific to each thread. The supplied `cleanup_function` will be used to destroy any thread-local objects when `reset()` is called, or the thread exits.

Throws: `boost::thread_resource_error` if an error occurs.

`~thread_specific_ptr();`

Requires: All the thread specific instances associated to this `thread_specific_ptr` (except maybe the one associated to this thread) must be null.

Effects: Calls `this->reset()` to clean up the associated value for the current thread, and destroys `*this`.

Throws: Nothing.

Remarks: The requirement is due to the fact that in order to delete all these instances, the implementation should be forced to maintain a list of all the threads having an associated specific ptr, which is against the goal of thread specific data.

**Note**

Care needs to be taken to ensure that any threads still running after an instance of `boost::thread_specific_ptr` has been destroyed do not call any member functions on that instance.

```
T* get() const;
```

Returns: The pointer associated with the current thread.

Throws: Nothing.

**Note**

The initial value associated with an instance of `boost::thread_specific_ptr` is `NULL` for each thread.

```
T* operator->() const;
```

Returns: `this->get()`

Throws: Nothing.

```
T& operator*() const;
```

Requires: `this->get` is not `NULL`.

Returns: `*(this->get())`

Throws: Nothing.

```
void reset(T* new_value=0);
```

Effects: If `this->get() != new_value` and `this->get()` is non-`NULL`, invoke `delete this->get()` or `cleanup_function(this->get())` as appropriate. Store `new_value` as the pointer associated with the current thread.

Postcondition: `this->get() == new_value`

Throws: `boost::thread_resource_error` if an error occurs.

```
T* release();
```

Effects: Return `this->get()` and store `NULL` as the pointer associated with the current thread without invoking the cleanup function.

Postcondition: `this->get() == 0`

Throws: Nothing.

Synchronized Data Structures

Synchronized Values - EXPERIMENTAL



Warning

These features are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(

Tutorial



Note

This tutorial is an adaptation of the paper of Anthony Williams "Enforcing Correct Mutex Usage with Synchronized Values" to the Boost library.

The Problem with Mutexes

The key problem with protecting shared data with a mutex is that there is no easy way to associate the mutex with the data. It is thus relatively easy to accidentally write code that fails to lock the right mutex - or even locks the wrong mutex - and the compiler will not help you.

```
std::mutex m1;
int value1;
std::mutex m2;
int value2;

int readValue1()
{
    boost::lock_guard<boost::mutex> lk(m1);
    return value1;
}
int readValue2()
{
    boost::lock_guard<boost::mutex> lk(m1); // oops: wrong mutex
    return value2;
}
```

Moreover, managing the mutex lock also clutters the source code, making it harder to see what is really going on.

The use of `synchronized_value` solves both these problems - the mutex is intimately tied to the value, so you cannot access it without a lock, and yet access semantics are still straightforward. For simple accesses, `synchronized_value` behaves like a pointer-to-T; for example:

```

boost::synchronized_value<std::string> value3;
std::string readValue3()
{
    return *value3;
}
void setValue3(std::string const& newVal)
{
    *value3=newVal;
}
void appendToValue3(std::string const& extra)
{
    value3->append(extra);
}

```

Both forms of pointer dereference return a proxy object rather than a real reference, to ensure that the lock on the mutex is held across the assignment or method call, but this is transparent to the user.

Beyond Simple Accesses

The pointer-like semantics work very well for simple accesses such as assignment and calls to member functions. However, sometimes you need to perform an operation that requires multiple accesses under protection of the same lock, and that's what the `synchronize()` method provides.

By calling `synchronize()` you obtain an `strict_lock_ptr` object that holds a lock on the mutex protecting the data, and which can be used to access the protected data. The lock is held until the `strict_lock_ptr` object is destroyed, so you can safely perform multi-part operations. The `strict_lock_ptr` object also acts as a pointer-to-T, just like `synchronized_value` does, but this time the lock is already held. For example, the following function adds a trailing slash to a path held in a `synchronized_value`. The use of the `strict_lock_ptr` object ensures that the string hasn't changed in between the query and the update.

```

void addTrailingSlashIfMissing(boost::synchronized_value<std::string> & path)
{
    boost::strict_lock_ptr<std::string> u=path.synchronize();

    if(u->empty() || (*u->rbegin() != '/'))
    {
        *u += '/';
    }
}

```

Operations Across Multiple Objects

Though `synchronized_value` works very well for protecting a single object of type T, nothing that we've seen so far solves the problem of operations that require atomic access to multiple objects unless those objects can be combined within a single structure protected by a single mutex.

One way to protect access to two `synchronized_value` objects is to construct a `strict_lock_ptr` for each object and use those to access the respective protected values; for instance:

```

synchronized_value<std::queue<MessageType> > q1,q2;
void transferMessage()
{
    strict_lock_ptr<std::queue<MessageType> > u1 = q1.synchronize();
    strict_lock_ptr<std::queue<MessageType> > u2 = q2.synchronize();

    if(!u1->empty())
    {
        u2->push_back(u1->front());
        u1->pop_front();
    }
}

```

This works well in some scenarios, but not all -- if the same two objects are updated together in different sections of code then you need to take care to ensure that the `strict_lock_ptr` objects are constructed in the same sequence in all cases, otherwise you have the potential for deadlock. This is just the same as when acquiring any two mutexes.

In order to be able to use the dead-lock free lock algorithms we need to use instead `unique_lock_ptr`, which is `Lockable`.

```

synchronized_value<std::queue<MessageType> > q1,q2;
void transferMessage()
{
    unique_lock_ptr<std::queue<MessageType> > u1 = q1.unique_synchronize(boost::defer_lock);
    unique_lock_ptr<std::queue<MessageType> > u2 = q2.unique_synchronize(boost::defer_lock);
    boost::lock(u1,u2); // dead-lock free algorithm

    if(!u1->empty())
    {
        u2->push_back(u1->front());
        u1->pop_front();
    }
}

```

While the preceding takes care of dead-lock, the access to the `synchronized_value` via `unique_lock_ptr` requires a lock that is not forced by the interface. An alternative on compilers providing a standard library that supports movable `std::tuple` is to use the free `synchronize` function, which will lock all the mutexes associated to the synchronized values and return a tuple of `strict_lock_ptr`.

```

synchronized_value<std::queue<MessageType> > q1,q2;
void transferMessage()
{
    auto lks = synchronize(u1,u2); // dead-lock free algorithm

    if(!std::get<1>(lks)->empty())
    {
        std::get<2>(lks)->push_back(u1->front());
        std::get<1>(lks)->pop_front();
    }
}

```

Value semantics

`synchronized_value` has value semantics even if the syntax lets is close to a pointer (this is just because we are unable to define smart references).

Reference

```
#include <boost/thread/synchronized_value.hpp>
namespace boost
{

    template<typename T, typename Lockable = mutex>
    class synchronized_value;

    // Specialized swap algorithm
    template <typename T, typename L>
    void swap(synchronized_value<T,L> & lhs, synchronized_value<T,L> & rhs);
    template <typename T, typename L>
    void swap(synchronized_value<T,L> & lhs, T & rhs);
    template <typename T, typename L>
    void swap(T & lhs, synchronized_value<T,L> & rhs);

    // Hash support
    template<typename T, typename L>
    struct hash<synchronized_value<T,L> >;

    // Comparison
    template <typename T, typename L>
    bool operator==(synchronized_value<T,L> const&lhs, synchronized_value<T,L> const& rhs)
    template <typename T, typename L>
    bool operator!=(synchronized_value<T,L> const&lhs, synchronized_value<T,L> const& rhs)
    template <typename T, typename L>
    bool operator<(synchronized_value<T,L> const&lhs, synchronized_value<T,L> const& rhs)
    template <typename T, typename L>
    bool operator<=(synchronized_value<T,L> const&lhs, synchronized_value<T,L> const& rhs)
    template <typename T, typename L>
    bool operator>(synchronized_value<T,L> const&lhs, synchronized_value<T,L> const& rhs)
    template <typename T, typename L>
    bool operator>=(synchronized_value<T,L> const&lhs, synchronized_value<T,L> const& rhs)

    // Comparison with T
    template <typename T, typename L>
    bool operator==(T const& lhs, synchronized_value<T,L> const&rhs);
    template <typename T, typename L>
    bool operator!=(T const& lhs, synchronized_value<T,L> const&rhs);
    template <typename T, typename L>
    bool operator<(T const& lhs, synchronized_value<T,L> const&rhs);
    template <typename T, typename L>
    bool operator<=(T const& lhs, synchronized_value<T,L> const&rhs);
    template <typename T, typename L>
    bool operator>(T const& lhs, synchronized_value<T,L> const&rhs);
    template <typename T, typename L>
    bool operator>=(T const& lhs, synchronized_value<T,L> const&rhs);

    template <typename T, typename L>
    bool operator==(synchronized_value<T,L> const& lhs, T const& rhs);
    template <typename T, typename L>
    bool operator!=(synchronized_value<T,L> const& lhs, T const& rhs);
    template <typename T, typename L>
    bool operator<(synchronized_value<T,L> const& lhs, T const& rhs);
    template <typename T, typename L>
    bool operator<=(synchronized_value<T,L> const& lhs, T const& rhs);
    template <typename T, typename L>
    bool operator>(synchronized_value<T,L> const& lhs, T const& rhs);
    template <typename T, typename L>
    bool operator>=(synchronized_value<T,L> const& lhs, T const& rhs);

}
```

```

#if ! defined(BOOST_THREAD_NO_SYNCHRONIZE)
    template <typename ...SV>
    std::tuple<typename synchronized_value_strict_lock_ptr<SV>::type ...> synchronize(SV& ...sv);
#endif
}

```

Class `synchronized_value`

```

#include <boost/thread/synchronized_value.hpp>

namespace boost
{
    template<typename T, typename Lockable = mutex>
    class synchronized_value
    {
    public:
        typedef T value_type;
        typedef Lockable mutex_type;

        synchronized_value() noexcept(is_nothrow_default_constructible<T>::value);
        synchronized_value(T const& other) noexcept(is_nothrow_copy_constructible<T>::value);
        synchronized_value(T&& other) noexcept(is_nothrow_move_constructible<T>::value);
        synchronized_value(synchronized_value const& rhs);
        synchronized_value(synchronized_value&& other);

        // mutation
        synchronized_value& operator=(synchronized_value const& rhs);
        synchronized_value& operator=(value_type const& val);
        void swap(synchronized_value & rhs);
        void swap(value_type & rhs);

        //observers
        T get() const;
#if ! defined(BOOST_NO_CXX11_EXPLICIT_CONVERSION_OPERATORS)
        explicit operator T() const;
#endif

        strict_lock_ptr<T,Lockable> operator->();
        const_strict_lock_ptr<T,Lockable> operator->() const;
        strict_lock_ptr<T,Lockable> synchronize();
        const_strict_lock_ptr<T,Lockable> synchronize() const;

        deref_value operator*();
        const_deref_value operator*() const;

    private:
        T value_; // for exposition only
        mutable mutex_type mtx_; // for exposition only
    };
}

```

Requires: Lockable is Lockable.

`synchronized_value()`

```
synchronized_value() noexcept(is_nothrow_default_constructible<T>::value);
```

Requires: T is DefaultConstructible.

Effects: Default constructs the cloaked `value_type`

Throws: Any exception thrown by `value_type()`.

`synchronized_value(T const&)`

```
synchronized_value(T const& other) noexcept(is_nothrow_copy_constructible<T>::value);
```

Requires: `T` is CopyConstructible.

Effects: Copy constructs the cloaked `value_type` using the parameter `other`

Throws: Any exception thrown by `value_type(other)`.

`synchronized_value(synchronized_value const&)`

```
synchronized_value(synchronized_value const& rhs);
```

Requires: `T` is DefaultConstructible and Assignable.

Effects: Assigns the value on a scope protected by the mutex of the rhs. The mutex is not copied.

Throws: Any exception thrown by `value_type()` or `value_type& operator=(value_type&)` or `mtx_.lock()`.

`synchronized_value(T&&)`

```
synchronized_value(T&& other) noexcept(is_nothrow_move_constructible<T>::value);
```

Requires: `T` is CopyMovable .

Effects: Move constructs the cloaked `value_type`

Throws: Any exception thrown by `value_type(value_type&&)`.

`synchronized_value(synchronized_value&&)`

```
synchronized_value(synchronized_value&& other);
```

Requires: `T` is CopyMovable .

Effects: Move constructs the cloaked `value_type`

Throws: Any exception thrown by `value_type(value_type&&)` or `mtx_.lock()`.

`operator=(synchronized_value const&)`

```
synchronized_value& operator=(synchronized_value const& rhs);
```

Requires: `T` is Assignable.

Effects: Copies the underlying value on a scope protected by the two mutexes. The mutex is not copied. The locks are acquired avoiding deadlock. For example, there is no problem if one thread assigns `a = b` and the other assigns `b = a`.

Return: `*this`

Throws: Any exception thrown by `value_type& operator(value_type const&)` or `mtx_.lock()`.

operator=(T const&)

```
synchronized_value& operator=(value_type const& val);
```

Requires: T is Assignable.

Effects: Copies the value on a scope protected by the mutex.

Return: *this

Throws: Any exception thrown by value_type& operator(value_type const&) or mtx_.lock().

get() const

```
T get() const;
```

Requires: T is CopyConstructible.

Return: A copy of the protected value obtained on a scope protected by the mutex.

Throws: Any exception thrown by value_type(value_type const&) or mtx_.lock().

operator T() const

```
#if ! defined(BOOST_NO_CXX11_EXPLICIT_CONVERSION_OPERATORS)
    explicit operator T() const;
#endif
```

Requires: T is CopyConstructible.

Return: A copy of the protected value obtained on a scope protected by the mutex.

Throws: Any exception thrown by value_type(value_type const&) or mtx_.lock().

swap(synchronized_value&)

```
void swap(synchronized_value & rhs);
```

Requires: T is Assignable.

Effects: Swaps the data on a scope protected by both mutex. Both mutex are acquired to avoid dead-lock. The mutexes are not swapped.

Throws: Any exception thrown by swap(value_, rhs.value) or mtx_.lock() or rhs_.mtx_.lock().

swap(synchronized_value&)

```
void swap(value_type & rhs);
```

Requires: T is Swappable.

Effects: Swaps the data on a scope protected by both mutex. Both mutex are acquired to avoid dead-lock. The mutexes are not swapped.

Throws: Any exception thrown by swap(value_, rhs) or mtx_.lock().

operator->()

```
strict_lock_ptr<T,Lockable> operator->();
```

Essentially calling a method `obj->foo(x, y, z)` calls the method `foo(x, y, z)` inside a critical section as long-lived as the call itself.

Return: A `strict_lock_ptr<>`.

Throws: Nothing.

operator->() const

```
const_strict_lock_ptr<T,Lockable> operator->() const;
```

If the `synchronized_value` object involved is `const`-qualified, then you'll only be able to call `const` methods through `operator->`. So, for example, `vec->push_back("xyz")` won't work if `vec` were `const`-qualified. The locking mechanism capitalizes on the assumption that `const` methods don't modify their underlying data.

Return: A `const_strict_lock_ptr <>`.

Throws: Nothing.

synchronize()

```
strict_lock_ptr<T,Lockable> synchronize();
```

The `synchronize()` factory make easier to lock on a scope. As discussed, `operator->` can only lock over the duration of a call, so it is insufficient for complex operations. With `synchronize()` you get to lock the object in a scoped and to directly access the object inside that scope.

Example:

```
void fun(synchronized_value<vector<int>> & vec) {
    auto vec2=vec.synchronize();
    vec2.push_back(42);
    assert(vec2.back() == 42);
}
```

Return: A `strict_lock_ptr <>`.

Throws: Nothing.

synchronize() const

```
const_strict_lock_ptr<T,Lockable> synchronize() const;
```

Return: A `const_strict_lock_ptr <>`.

Throws: Nothing.

operator*()

```
deref_value operator*();
```

Return: A an instance of a class that locks the mutex on construction and unlocks it on destruction and provides implicit conversion to a reference to the protected value.

Throws: Nothing.

operator*() const

```
const_deref_value operator*() const;
```

Return: A an instance of a class that locks the mutex on construction and unlocks it on destruction and provides implicit conversion to a constant reference to the protected value.

Throws: Nothing.

Non-Member Function synchronize

```
#include <boost/thread/synchronized_value.hpp>
namespace boost
{
    #if ! defined(BOOST_THREAD_NO_SYNCHRONIZE)
        template <typename ...SV>
        std::tuple<typename synchronized_value_strict_lock_ptr<SV>::type ...> synchronize(SV& ...sv);
    #endif
}
```

Synchronized Queues -- EXPERIMENTAL

**Warning**

These features are experimental and subject to change in future versions. There are not too much tests yet, so it is possible that you can find out some trivial bugs :(

**Note**

These features are based on the [N3533 - C++ Concurrent Queues](#) C++1y proposal from Lawrence Crowl and Chris Mysisen and [C++ Concurrency in Action](#) from Anthony Williams.

Introduction

Queues provide a mechanism for communicating data between components of a system.

The existing deque in the standard library is an inherently sequential data structure. Its reference-returning element access operations cannot synchronize access to those elements with other queue operations. So, concurrent pushes and pops on queues require a different interface to the queue structure.

Moreover, concurrency adds a new dimension for performance and semantics. Different queue implementation must trade off uncontended operation cost, contended operation cost, and element order guarantees. Some of these trade-offs will necessarily result in semantics weaker than a serial queue.

Tutorial

Concurrent queues are a well know mechanism for communicating data between different threads.

Concurrent queues have inherently copy/move semantics for the data handling operation. Reference-returning interfaces are forbidden as multiple access to these references can not be thread-safe.

Examples

Reference

Synchronized Queue Model

Bounded-Unbounded Queues

One of the major features of a concurrent queue is whether it has a bounded-unbounded capacity.

Locking/Lock-free Queues

Locking queues can by nature block waiting for the queue to be non-empty or non-full.

Lock-free queues will have some trouble waiting for the queue to be non-empty or non-full queues. These queues can not define operations such as push (and pull for bounded queues). That is, it could have blocking operations (presumably emulated with busy wait) but not waiting operations.

Closed Queue

Threads using a queue for communication need some mechanism to signal when the queue is no longer needed. The usual approach is add an additional out-of-band signal. However, this approach suffers from the flaw that threads waiting on either full or empty queues need to be woken up when the queue is no longer needed. Rather than require an out-of-band signal, we chose to directly support such a signal in the queue itself, which considerably simplifies coding.

To achieve this signal, a thread may close a queue. Once closed, no new elements may be pushed onto the queue. Push operations on a closed queue will either return `queue_op_status::closed` (when they have a `queue_op_status` return type), set the closed parameter if it has one or throw `sync_queue::closed` (when they do not). Elements already on the queue may be pulled off. When a queue is empty and closed, pull operations will either return `queue_op_status::closed` (when they have a status return), set the closed parameter if it has one or throw `sync_queue::closed` (when they do not).

Concurrent Queues Throw specification

Locking

All the functions are defined as if we had in addition to its specific Throw specification the following:

Throws: Any exception thrown by the internal locking.

Allocation

All the functions that allocate a resource are defined as if we had in addition to its specific Throw specification the following:

Throws: Any exception due to allocation errors.

Basic Concurrent Queue Operations

The essential solution to the problem of concurrent queuing is to shift to value-based operations, rather than reference-based operations.

The `BasicConcurrentQueue` concept models the basic operations of a concurrent queue.

A type `Q` meets the `BasicConcurrentQueue` requirements if the following expressions are well-formed and have the specified semantics

- `Q::value_type`
- `Q::size_type`
- `q.push_back(e);`
- `q.push_back(rve);`
- `q.pull_front(lre);`
- `lre = q.pull_front();`
- `b = q.empty();`
- `u = q.size();`

where

- `q` denotes a value of type `Q`,
- `e` denotes a value of type `Q::value_type`,
- `u` denotes a value of type `Q::size_type`,
- `lre` denotes a lvalue reference of type `Q::value_type`,
- `rve` denotes a rvalue reference of type `Q::value_type`:
- `qs` denotes a variable of type `queus_op_status`,

`q.push_back(e);`

Effects:	Waits until the queue is not full (for bounded queues) and then push back <code>e</code> to the queue copying it (this could need an allocation for unbounded queues).
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Postcondition:	<code>! q.empty()</code> .
Return type:	<code>void</code> .
Throws:	If the queue was closed, throws <code>sync_queue_is_closed</code> . Any exception thrown by the copy of <code>e</code> .
Exception safety:	If an exception is thrown then the queue state is unmodified.

`q.push_back(rve);`

Effects:	Waits until the queue is not full (for bounded queues) and then push <code>e</code> to the queue moving it back in the queue (this could need an allocation for unbounded queues).
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Postcondition:	<code>! q.empty()</code> .
Return type:	<code>void</code> .
Throws:	If the queue is closed, throws <code>sync_queue_is_closed</code> . Any exception thrown by the copy of <code>e</code> .
Exception safety:	If an exception is thrown then the queue state is unmodified.

q.pull_front(lve)

Effects:	Waits until the queue is not empty and then pull_front the element from the queue q and moves the pulled element into lve (this could need an allocation for unbounded queues).
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Postcondition:	<code>! q.full()</code> .
Return type:	<code>void</code> .
Throws:	Any exception thrown by the move of e.
Exception safety:	If an exception is thrown then the queue state is unmodified.

e = q.pull_front()

Requires:	<code>Q::value_type</code> is no throw copy movable. This is needed to ensure the exception safety.
Effects:	Waits until the queue is not empty and not closed. If the queue is empty and closed throws <code>sync_queue_is_closed</code> . Otherwise pull the element from the queue q and moves the pulled element.
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Postcondition:	<code>! q.full()</code> .
Return type:	<code>Q::value_type</code> .
Return:	The pulled element.
Throws:	Any exception thrown by the copy of e.
Exception safety:	If an exception is thrown then the queue state is unmodified.

Non-waiting Concurrent Queue Operations

The ConcurrentQueue concept models a queue with .

A type Q meets the ConcurrentQueue requirements if the following expressions are well-formed and have the specified semantics

- `s = q.try_push_back(e);`
- `s = q.try_push_back(rve);`
- `s = q.try_pull_front(lre);`

where

- q denotes a value of type Q,
- e denotes a value of type `Q::value_type`,
- s denotes a value of type `queue_status`,
- u denotes a value of type `Q::size_type`,
- lve denotes a lvalue referece of type `Q::value_type`,
- rve denotes a rvalue referece of type `Q::value_type`:

```
s = q.try_push_back(e);
```

Effects:	If the queue <code>q</code> is not full and not closed, push back the <code>e</code> to the queue copying it.
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation when the operation succeeds.
Return type:	<code>queue_op_status</code> .
Return:	<ul style="list-style-type: none">- If the queue is closed, returns <code>queue_op_status::closed</code>,- otherwise if the queue <code>q</code> is full return <code>queue_op_status::full</code>,- otherwise return <code>queue_op_status::success</code>;
Postcondition:	If the call returns <code>queue_op_status::success</code> , <code>! q.empty()</code> .
Throws:	If the queue is closed, throws <code>sync_queue_is_closed</code> . Any exception thrown by the copy of <code>e</code> .
Exception safety:	If an exception is thrown then the queue state is unmodified.

```
s = q.try_push_back(rve());
```

Effects:	If the queue <code>q</code> is not full and not closed, push back the <code>e</code> onto the queue moving it.
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Return type:	<code>queue_op_status</code> .
Return:	<ul style="list-style-type: none">- If the queue is closed, returns <code>queue_op_status::closed</code>,- otherwise if the queue <code>q</code> is full return <code>queue_op_status::full</code>,- otherwise return <code>queue_op_status::success</code>;
Postcondition:	If the call returns <code>queue_op_status::success</code> , <code>! q.empty()</code> .
Throws:	Any exception thrown by the copy of <code>e</code> .
Exception safety:	If an exception is thrown then the queue state is unmodified.

```
s = q.try_pull_front(lve)
```

Effects:	Waits until the queue is not empty and then pull the element from the queue <code>q</code> and moves the pulled element into <code>lve</code> (this could need an allocation for unbounded queues).
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Postcondition:	<code>! q.full()</code> .
Return type:	<code>bool</code> .
Return:	<ul style="list-style-type: none">- If the queue <code>q</code> is empty return <code>queue_op_status::empty</code>,- otherwise return <code>queue_op_status::success</code>;
Throws:	Any exception thrown by the move of <code>e</code> .
Exception safety:	If an exception is thrown then the queue state is unmodified.

Non-blocking Concurrent Queue Operations

For cases when blocking for mutual exclusion is undesirable, we have non-blocking operations. The interface is the same as the try operations but is allowed to also return `queue_op_status::busy` in case the operation is unable to complete without blocking.

Non-blocking operations are provided only for `BlockingQueues`

- `s = q.nonblocking_push_back(nb, e);`
- `s = q.nonblocking_push_back(nb, rve);`
- `s = q.nonblocking_pull_front(nb, lre);`

where

- `q` denotes a value of type `Q`,
- `e` denotes a value of type `Q::value_type`,
- `s` denotes a value of type `queue_status`,
- `lve` denotes a lvalue reference of type `Q::value_type`,
- `rve` denotes a rvalue reference of type `Q::value_type`:

`s = q.nonblocking_push_back(e);`

Effects:	If the queue <code>q</code> is not full and not closed, push back the <code>e</code> to the queue copying it.
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation when the operation succeeds.
Return type:	<code>queue_op_status</code> .
Return:	<ul style="list-style-type: none"> - If the operation would block, return <code>queue_op_status::busy</code>, - otherwise, if the queue is closed, return <code>queue_op_status::closed</code>, - otherwise, if the queue <code>q</code> is full return <code>queue_op_status::full</code>, - otherwise return <code>queue_op_status::success</code>;
Postcondition:	If the call returns <code>queue_op_status::success</code> , ! <code>q.empty()</code> .
Throws:	If the queue is closed, throws <code>sync_queue_is_closed</code> . Any exception thrown by the copy of <code>e</code> .
Exception safety:	If an exception is thrown then the queue state is unmodified.

`s = q.nonblocking_push_back(rve());`

Effects:	If the queue <code>q</code> is not full and not closed, push back the <code>e</code> onto the queue moving it.
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Return type:	<code>queue_op_status</code> .
Return:	<ul style="list-style-type: none"> - If the operation would block, return <code>queue_op_status::busy</code>, - otherwise if the queue is closed, returns <code>queue_op_status::closed</code>, - otherwise if the queue <code>q</code> is full return <code>queue_op_status::full</code>, - otherwise return <code>queue_op_status::success</code>;

Postcondition: If the call returns `queue_op_status::success`, ! `q.empty()`.

Throws: Any exception thrown by the copy of `e`.

Exception safety: If an exception is thrown then the queue state is unmodified.

`s = q.nonblocking_pull_front(lve)`

Effects: Waits until the queue is not empty and then pull the element from the queue `q` and moves the pulled element into `lve` (this could need an allocation for unbounded queues).

Synchronization: Prior pull-like operations on the same object synchronizes with this operation.

Postcondition: ! `q.full()`.

Return type: `bool`.

Return:

- If the operation would block, return `queue_op_status::busy`,
- otherwise if the queue `q` is empty return `queue_op_status::empty`,
- otherwise return `queue_op_status::success`;

Throws: Any exception thrown by the move of `e`.

Exception safety: If an exception is thrown then the queue state is unmodified.

Bounded Concurrent Queue Operations

Bounded queues add the following valid expressions

- `Q q(u);`
- `b = q.full();`
- `u = q.capacity();`

where

- `q` denotes a value of type `Q`,
- `b` denotes a value of type `bool`,
- `u` denotes a value of type `Q::size_type`,

`b = q.full();`

Return type: `bool`.

Return: Return `true` iff the queue is full.

Remark: Not all queues will have a full state, and these would always return false if the function is provided.

`b = q.capacity();`

Return type: `Q::size_type`.

Return: Return the capacity of queue.

Closed Concurrent Queue Operations

Closed queues add the following valid expressions

- `q.close();`
- `b = q.closed();`
- `s = q.wait_push_back(e);`
- `s = q.wait_push_back(rve);`
- `s = q.wait_pull_front(lre);`

`q.close();`

Effects: Close the queue.

`b = q.closed();`

Return type: `bool`.

Return: Return `true` iff the queue is closed.

`s = q.wait_push_back(e);`

Effects: Waits until the queue is not full (for bounded queues) and then push back `e` to the queue copying it (this could need an allocation for unbounded queues).

Synchronization: Prior pull-like operations on the same object synchronizes with this operation.

Postcondition: `! q.empty()`.

Return type: `queue_op_status`.

Return: - If the queue is closed return `queue_op_status::closed`,
- otherwise, return `queue_op_status::success` if no exception is thrown.

Throws: Any exception thrown by the copy of `e`.

Exception safety: If an exception is thrown then the queue state is unmodified.

`s = q.wait_push_back(rve);`

Effects: Waits until the queue is not full (for bounded queues) and then push `e` to the queue moving it back in the queue (this could need an allocation for unbounded queues).

Synchronization: Prior pull-like operations on the same object synchronizes with this operation.

Postcondition: `! q.empty()`.

Return type: `queue_op_status`.

Return: - If the queue is closed return `queue_op_status::closed`,
- otherwise, return `queue_op_status::success` if no exception is thrown.

.

Throws: Any exception thrown by the copy of `e`.

Exception safety: If an exception is thrown then the queue state is unmodified.

s = q.wait_pull_front(lve)

Effects:	if the queue is not empty and not closed, waits until the queue is not empty and then pull_front the element from the queue q and moves the pulled element into lve.
Synchronization:	Prior pull-like operations on the same object synchronizes with this operation.
Postcondition:	<code>! q.full()</code> .
Return type:	<code>queue_op_status</code> .
Return:	<ul style="list-style-type: none"> - If the queue is empty and closed, return <code>queue_op_status::closed</code>, - otherwise, return <code>queue_op_status::success</code> if no exception is thrown.
Throws:	Any exception thrown by the move of e.
Exception safety:	If an exception is thrown then the queue state is unmodified.

Synchronized Bounded Queue

```
#include <boost/thread/sync_bounded_queue.hpp>

namespace boost
{
    struct sync_queue_is_closed : std::exception {};

    template <typename ValueType>
    class sync_bounded_queue;

    // Stream-like operators
    template <typename ValueType>
    sync_bounded_queue<ValueType>& operator<< (sync_bounded_queue<ValueType>& sbq, ValueType&& elem);
    template <typename ValueType>
    sync_bounded_queue<ValueType>& operator>> (sync_bounded_queue<ValueType>& sbq, ValueType&& elem);
    or<< (sync_bounded_queue<ValueType>& sbq, ValueType const&elem);
    template <typename ValueType>
    sync_bounded_queue<ValueType>& operator>> (sync_bounded_queue<ValueType>& sbq, ValueType &elem);
}
```

Class sync_queue_is_closed

```
#include <boost/thread/sync_bounded_queue.hpp>

namespace boost
{
    struct sync_queue_is_closed : std::exception {};
}
```

Class template `sync_bounded_queue<>`

```

#include <boost/thread/sync_bounded_queue.hpp>
namespace boost
{
    template <typename ValueType>
    class sync_bounded_queue
    {
    public:
        typedef ValueType value_type;
        typedef std::size_t size_type;

        sync_bounded_queue(sync_bounded_queue const&) = delete;
        sync_bounded_queue& operator=(sync_bounded_queue const&) = delete;
        explicit sync_bounded_queue(size_type max_elems);
        template <typename Range>
        sync_bounded_queue(size_type max_elems, Range range);
        ~sync_bounded_queue();

        // Observers
        bool empty() const;
        bool full() const;
        size_type capacity() const;
        size_type size() const;
        bool closed() const;

        // Modifiers
        void push_back(const value_type& x);
        void push_back(value_type&& x);

        queue_op_status try_push_back(const value_type& x);
        queue_op_status try_push_back(value_type&& x);

        queue_op_status nonblocking_push_back(const value_type& x);
        queue_op_status nonblocking_push_back(value_type&& x);

        void pull_front(value_type&);
        value_type pull_front();

        queue_op_status try_pull_front(value_type&);
        queue_op_status nonblocking_pull_front(value_type&);

        void close();
    };
}

```

Constructor `sync_bounded_queue(size_type)`

```
explicit sync_bounded_queue(size_type max_elems);
```

Effects: Constructs a `sync_bounded_queue` with a maximum number of elements given by `max_elems`.

Throws: any exception that can be throw because of resources unavailable.

Template Constructor `sync_bounded_queue(size_type, Range)`

```
template <typename Range>
sync_bounded_queue(size_type max_elems, Range range);
```

Effects: Constructs a `sync_bounded_queue` with a maximum number of elements given by `max_elems` and push back the elements of the range.

Throws: any exception that can be throw because of resources unavailable.

Non-Member Function `operator<<()`

```
#include <boost/thread/sync_bounded_queue.hpp>
namespace boost
{
    template <typename ValueType>
    sync_bounded_queue<ValueType>& operator<<(sync_bounded_queue<ValueType>& sbq, ValueType&& elem);
    template <typename ValueType>
    sync_bounded_queue<ValueType>& operator<<(sync_bounded_queue<ValueType>& sbq, ValueType const&elem);
}
```

Non-Member Function `operator>>()`

```
#include <boost/thread/sync_bounded_queue.hpp>
namespace boost
{
    template <typename ValueType>
    sync_bounded_queue<ValueType>& operator>>(sync_bounded_queue<ValueType>& sbq, ValueType &elem);
}
```

Synchronized Unbounded Queue

```
#include <boost/thread/sync_queue.hpp>
namespace boost
{
    template <typename ValueType>
    class sync_queue;

    // Stream-like operators
    template <typename ValueType>
    sync_queue<ValueType>& operator<<(sync_queue<ValueType>& sbq, ValueType&& elem);
    template <typename ValueType>
    sync_queue<ValueType>& operator<<(sync_queue<ValueType>& sbq, ValueType const&elem);
    template <typename ValueType>
    sync_queue<ValueType>& operator>>(sync_queue<ValueType>& sbq, ValueType &elem);
}
```

Class template `sync_queue<>`

```

#include <boost/thread/sync_queue.hpp>

namespace boost
{
    template <typename ValueType>
    class sync_queue
    {
    public:
        typedef ValueType value_type;
        typedef csbl::deque<ValueType> underlying_queue_type;
        typedef std::size_t size_type;

        sync_queue(sync_queue const&) = delete;
        sync_queue& operator=(sync_queue const&) = delete;
        sync_queue();
        explicit template <typename Range>
        sync_queue(Range range); // Not yet implemented
        ~sync_queue();

        // Observers
        bool empty() const;
        bool full() const;
        size_type size() const;
        bool closed() const;

        // Modifiers
        void push_back(const value_type& x);
        void push_back(value_type&& x);

        queue_op_status try_push_back(const value_type& x);
        queue_op_status try_push_back(value_type&& x);

        queue_op_status nonblocking_push_back(const value_type& x);
        queue_op_status nonblocking_push_back(value_type&& x);

        void pull_front(value_type&);
        value_type pull_front();

        queue_op_status try_pull_front(value_type&);
        queue_op_status nonblocking_pull_front(value_type&);

        underlying_queue_type underlying_queue() noexcept;

        void close();
    };
}

```

Constructor `sync_bounded_queue(size_type)`

```
explicit sync_queue();
```

Effects: Constructs an empty `sync_queue`.

Throws: any exception that can be throw because of resources unavailable.

Template Constructor `sync_bounded_queue(size_type, Range)`

```
template <typename Range>
sync_bounded_queue(size_type max_elems, Range range);
```

Effects: Constructs an `sync_queue` with all the elements of the range.

Throws: any exception that can be throw because of resources unavailable.

Member Function `full()`

```
bool full() const;
```

Returns: `false`.

Member Function `underlying_queue()`

```
underlying_queue_type underlying_queue() noexcept;
```

Returns: Moves internal queue.

Non-Member Function `operator<<()`

```
#include <boost/thread/sync_queue.hpp>
namespace boost
{
    template <typename ValueType>
    sync_queue<ValueType>& operator<<(sync_queue<ValueType>& sbq, ValueType&& elem);
    template <typename ValueType>
    sync_queue<ValueType>& operator<<(sync_queue<ValueType>& sbq, ValueType const&elem);
}
```

Non-Member Function `operator>>()`

```
#include <boost/thread/sync_queue.hpp>
namespace boost
{
    template <typename ValueType>
    sync_queue<ValueType>& operator>>(sync_queue<ValueType>& sbq, ValueType &elem);
}
```

Time Requirements

As of Boost 1.50.0, the **Boost.Thread** library uses Boost.Chrono library for all operations that require a time out as defined in the standard c++11. These include (but are not limited to):

- `boost::this_thread::sleep_for`
- `boost::this_thread::sleep_until`
- `boost::thread::try_join_for`
- `boost::thread::try_join_until`
- `boost::condition_variable::wait_for`
- `boost::condition_variable::wait_until`
- `boost::condition_variable_any::wait_for`
- `boost::condition_variable_any::wait_until`
- `TimedLockable::try_lock_for`
- `TimedLockable::try_lock_until`

Deprecated

The time related functions introduced in Boost 1.35.0, using the [Boost.Date_Time](#) library are deprecated. These include (but are not limited to):

- `boost::this_thread::sleep()`
- `timed_join()`
- `timed_wait()`
- `timed_lock()`

For the overloads that accept an absolute time parameter, an object of type `boost::system_time` is required. Typically, this will be obtained by adding a duration to the current time, obtained with a call to `boost::get_system_time()`. e.g.

```
boost::system_time const timeout=boost::get_system_time() + boost::posix_time::milliseconds(500);

extern bool done;
extern boost::mutex m;
extern boost::condition_variable cond;

boost::unique_lock<boost::mutex> lk(m);
while(!done)
{
    if(!cond.timed_wait(lk,timeout))
    {
        throw "timed out";
    }
}
```

For the overloads that accept a *TimeDuration* parameter, an object of any type that meets the [Boost.Date_Time Time Duration requirements](#) can be used, e.g.

```
boost::this_thread::sleep(boost::posix_time::milliseconds(25));

boost::mutex m;
if(m.timed_lock(boost::posix_time::nanoseconds(100)))
{
    // ...
}
```

Typedef `system_time`

```
#include <boost/thread/thread_time.hpp>

typedef boost::posix_time::ptime system_time;
```

See the documentation for `boost::posix_time::ptime` in the Boost.Date_Time library.

Non-member function `get_system_time()`

```
#include <boost/thread/thread_time.hpp>

system_time get_system_time();
```

Returns: The current time.

Throws: Nothing.

Emulations

=delete emulation

C++11 allows to delete some implicitly generated functions as constructors and assignment using '= delete' as in

```
public:
    thread(thread const&) = delete;
```

On compilers not supporting this feature, Boost.Thread relays on a partial simulation, it declares the function as private without definition.

```
private:
    thread(thread &);
```

The emulation is partial as the private function can be used for overload resolution for some compilers and prefer it to other overloads that need a conversion. See below the consequences on the move semantic emulation.

Move semantics

In order to implement Movable classes, move parameters and return types Boost.Thread uses the rvalue reference when the compiler support it. On compilers not supporting it Boost.Thread uses either the emulation provided by Boost.Move or the emulation provided by the previous versions of Boost.Thread depending whether `BOOST_THREAD_USES_MOVE` is defined or not. This macros is unset by default when `BOOST_THREAD_VERSION` is 2. Since `BOOST_THREAD_VERSION 3`, `BOOST_THREAD_USES_MOVE` is defined.

Deprecated Version 2 interface

Previous to version 1.50, Boost.Thread make use of its own move semantic emulation which had more limitations than the provided by Boost.Move. In addition, it is of interest of the whole Boost community that Boost.Thread uses Boost.Move so that `boost::thread` can be stored on Movable aware containers.

To preserve backward compatibility at least during some releases, Boost.Thread allows the user to use the deprecated move semantic emulation defining `BOOST_THREAD_DONT_USE_MOVE`.

Many aspects of move semantics can be emulated for compilers not supporting rvalue references and Boost.Thread legacy offers tools for that purpose.

Helpers class and function

Next follows the interface of the legacy move semantic helper class and function.


```
namespace boost
{
    namespace detail
    {
        template<typename T>
        struct thread_move_t
        {
            explicit thread_move_t(T& t_);
            T& operator*() const;
            T* operator->() const;
        private:
            void operator=(thread_move_t&);
        };
    }
    template<typename T>
    boost::detail::thread_move_t<T> move(boost::detail::thread_move_t<T> t);
}
```

Movable emulation

We can write a MovableOnly class as follows. You just need to follow these simple steps:

- Add a conversion to the `detail::thread_move_t<classname>`
- Make the copy constructor private.
- Write a constructor taking the parameter as `detail::thread_move_t<classname>`
- Write an assignment taking the parameter as `detail::thread_move_t<classname>`

For example the thread class defines the following:

```

class thread
{
    // ...
private:
    thread(thread&);
    thread& operator=(thread&);
public:
    detail::thread_move_t<thread> move()
    {
        detail::thread_move_t<thread> x(*this);
        return x;
    }
    operator detail::thread_move_t<thread>()
    {
        return move();
    }
    thread(detail::thread_move_t<thread> x)
    {
        thread_info=x->thread_info;
        x->thread_info.reset();
    }
    thread& operator=(detail::thread_move_t<thread> x)
    {
        thread new_thread(x);
        swap(new_thread);
        return *this;
    }
    // ...
};

```

Portable interface

In order to make the library code portable Boost.Thread uses some macros that will use either the ones provided by Boost.Move or the deprecated move semantics provided by previous versions of Boost.Thread.

See the Boost.Move documentation for a complete description on how to declare new Movable classes and its limitations.

- `BOOST_THREAD_RV_REF(TYPE)` is the equivalent of `BOOST_RV_REF(TYPE)`
- `BOOST_THREAD_RV_REF_BEG` is the equivalent of `BOOST_RV_REF_BEG(TYPE)`
- `BOOST_THREAD_RV_REF_END` is the equivalent of `BOOST_RV_REF_END(TYPE)`
- `BOOST_THREAD_FWD_REF(TYPE)` is the equivalent of `BOOST_FWD_REF(TYPE)`

In addition the following macros are needed to make the code portable:

- `BOOST_THREAD_RV(V)` macro to access the rvalue from a `BOOST_THREAD_RV_REF(TYPE)`,
- `BOOST_THREAD_MAKE_RV_REF(RVALUE)` makes a rvalue.
- `BOOST_THREAD_DCL_MOVABLE(CLASS)` to avoid conflicts with Boost.Move
- `BOOST_THREAD_DCL_MOVABLE_BEG(T1)` and `BOOST_THREAD_DCL_MOVABLE_END` are variant of `BOOST_THREAD_DCL_MOVABLE` when the parameter is a template instantiation.

Other macros are provided and must be included on the public section:

- `BOOST_THREAD_NO_COPYABLE` declares a class no-copyable either deleting the copy constructors and copy assignment or moving them to the private section.

- `BOOST_THREAD_MOVABLE(CLASS)` declares all the implicit conversions to an rvalue-reference.
- `BOOST_THREAD_MOVABLE_ONLY(CLASS)` is the equivalent of `BOOST_MOVABLE_BUT_NOT_COPYABLE(CLASS)`
- `BOOST_THREAD_COPYABLE_AND_MOVABLE(CLASS)` is the equivalent of `BOOST_COPYABLE_AND_MOVABLE(CLASS)`

`BOOST_THREAD_NO_COPYABLE(CLASS)`

This macro marks a class as no copyable, disabling copy construction and assignment.

`BOOST_THREAD_MOVABLE(CLASS)`

This macro marks a class as movable, declaring all the implicit conversions to an rvalue-reference.

`BOOST_THREAD_MOVABLE_ONLY(CLASS)`

This macro marks a type as movable but not copyable, disabling copy construction and assignment. The user will need to write a move constructor/assignment to fully write a movable but not copyable class.

`BOOST_THREAD_COPYABLE_AND_MOVABLE(CLASS)`

This macro marks a type as copyable and movable. The user will need to write a move constructor/assignment and a copy assignment to fully write a copyable and movable class.

`BOOST_THREAD_RV_REF(TYPE)`, `BOOST_THREAD_RV_REF_BEG` and `BOOST_THREAD_RV_REF_END`

This macro is used to achieve portable syntax in move constructors and assignments for classes marked as `BOOST_THREAD_COPYABLE_AND_MOVABLE` or `BOOST_THREAD_MOVABLE_ONLY`.

`BOOST_THREAD_RV_REF_BEG` and `BOOST_THREAD_RV_REF_END` are used when the parameter end with a `>` to avoid the compiler error.

`BOOST_THREAD_RV(V)`

While Boost.Move emulation allows to access an rvalue reference `BOOST_THREAD_RV_REF(TYPE)` using the dot operator, the legacy defines the operator `->`. We need then a macro `BOOST_THREAD_RV` that mask this difference. E.g.

```
thread(BOOST_THREAD_RV_REF(thread) x)
{
    thread_info=BOOST_THREAD_RV(x).thread_info;
    BOOST_THREAD_RV(x).thread_info.reset();
}
```

The use of this macros has reduced considerably the size of the Boost.Thread move related code.

`BOOST_THREAD_MAKE_RV_REF(RVALUE)`

While Boost.Move is the best C++03 move emulation there are some limitations that impact the way the library can be used. For example, with the following declarations

```
class thread {
    // ...
private:
    thread(thread &);
public:
    thread(rv<thread>&);
    // ...
};
```

This could not work on some compilers even if `thread` is convertible to `rv<thread>` because the compiler prefers the private copy constructor.

```
thread mkth()
{
    return thread(f);
}
```

On these compilers we need to use instead an explicit conversion. The library provides a move member function that allows to workaround the issue.

```
thread mkth()
{
    return thread(f).move();
}
```

Note that `::boost::move` can not be used in this case as `thread` is not implicitly convertible to `thread&`.

```
thread mkth()
{
    return ::boost::move(thread(f));
}
```

To make the code portable Boost.Thread the user needs to use a macro `BOOST_THREAD_MAKE_RV_REF` that can be used as in

```
thread mkth()
{
    return BOOST_THREAD_MAKE_RV_REF(thread(f));
}
```

Note that this limitation is shared also by the legacy Boost.Thread move emulation.

`BOOST_THREAD_DCL_MOVABLE`, `BOOST_THREAD_DCL_MOVABLE_BEG(T1)` and `BOOST_THREAD_DCL_MOVABLE_END`

As Boost.Move defines also the `boost::move` function we need to specialize the `has_move_emulation_enabled_aux` metafunction.

```
template <>
struct has_move_emulation_enabled_aux<thread>
    : BOOST_MOVE_BOOST_NS::integral_constant<bool, true>
{
};
```

so that the following Boost.Move overload is disabled

```
template <class T>
inline typename BOOST_MOVE_BOOST_NS::disable_if<has_move_emulation_enabled_aux<T>, T&>::type move(T& x);
```

The macros `BOOST_THREAD_DCL_MOVABLE(CLASS)`, `BOOST_THREAD_DCL_MOVABLE_BEG(T1)` and `BOOST_THREAD_DCL_MOVABLE_END` are used for this purpose. E.g.

```
BOOST_THREAD_DCL_MOVABLE(thread)
```

and

```
BOOST_THREAD_DCL_MOVABLE_BEG(T) promise<T> BOOST_THREAD_DCL_MOVABLE_END
```

Bool explicit conversion

Locks provide an explicit bool conversion operator when the compiler provides them.

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

The library provides an implicit conversion to an undefined type that can be used as a conditional expression.

```
#if defined(BOOST_NO_EXPLICIT_CONVERSION_OPERATORS)
    operator unspecified-bool-type() const;
    bool operator!() const;
#else
    explicit operator bool() const;
#endif
```

The user should use the `lock.owns_lock()` when an explicit conversion is required.

operator *unspecified-bool-type*() const

Returns: If `owns_lock()` would return true, a value that evaluates to true in boolean contexts, otherwise a value that evaluates to false in boolean contexts.

Throws: Nothing.

bool operator!() const

Returns: `! owns_lock()`.

Throws: Nothing.

Scoped Enums

Some of the enumerations defined in the standard library are scoped enums.

On compilers that don't support them, the library uses a class to wrap the underlying type. Instead of

```
enum class future_errc
{
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
};
```

the library declares these types as

```
BOOST_SCOPED_ENUM_DECLARE_BEGIN( future_errc )
{
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied,
    no_state
}
BOOST_SCOPED_ENUM_DECLARE_END( future_errc )
```

These macros allows to use 'future_errc' in almost all the cases as an scoped enum.

There are however some limitations:

- The type is not a C++ enum, so 'is_enum<future_errc>' will be false_type.
- The emulated scoped enum can not be used in switch nor in template arguments. For these cases the user needs to use some macros.

Instead of

```
switch (ev)
{
    case future_errc::broken_promise:
// ...
```

use

```
switch (boost::native_value(ev))
{
    case future_errc::broken_promise:
```

And instead of

```
#ifdef BOOST_NO_SCOPED_ENUMS
template <>
struct BOOST_SYMBOL_VISIBLE is_error_code_enum<future_errc> : public true_type { };
#endif
```

use

```
#ifdef BOOST_NO_SCOPED_ENUMS
template <>
struct BOOST_SYMBOL_VISIBLE is_error_code_enum<future_errc::enum_type> : public true_type { };
#endif
```

Acknowledgments

The original implementation of **Boost.Thread** was written by William Kempf, with contributions from numerous others. This new version initially grew out of an attempt to rewrite **Boost.Thread** to William Kempf's design with fresh code that could be released under the Boost Software License. However, as the C++ Standards committee have been actively discussing standardizing a thread library for C++, this library has evolved to reflect the proposals, whilst retaining as much backwards-compatibility as possible.

Particular thanks must be given to Roland Schwarz, who contributed a lot of time and code to the original **Boost.Thread** library, and who has been actively involved with the rewrite. The scheme for dividing the platform-specific implementations into separate directories was devised by Roland, and his input has contributed greatly to improving the quality of the current implementation.

Thanks also must go to Peter Dimov, Howard Hinnant, Alexander Terekhov, Chris Thomasson and others for their comments on the implementation details of the code.

Conformance and Extension

C++11 standard Thread library



Note

C++11 - Standard for Programming Language C++

]

Table 2. C++11 standard Conformance

Section	Description	Status	Comments	Ticket
30	Thread support library	Yes	-	-
30.1	General	-	-	-
30.2	Requirements	-	-	-
30.2.1	Template parameter names	-	-	-
30.2.2	Exceptions	Yes	-	-
30.2.3	Native handles	Yes	-	-
30.2.4	Timing specifications	Yes	-	-
30.2.5	Requirements for Lockable types	Yes	-	-
30.2.5.1	In general	-	-	-
30.2.5.2	BasicLockable requirements	Yes	-	-
30.2.5.3	Lockable requirements	yes	-	-
30.2.5.4	TimedLockable requirements	Yes	-	-
30.2.6	decay_copy	-	-	-
30.3	Threads	Yes	-	-
30.3.1	Class thread	Yes	-	-
30.3.1.1	Class thread::id	Yes	-	-
30.3.1.2	thread constructors	Partial	-	-
30.3.1.3	thread destructor	Yes	-	-
30.3.1.4	thread assignment	Yes	-	-
30.3.1.5	thread members	Yes	-	-
30.3.1.6	thread static members	Yes	-	-
30.3.1.7	thread specialized algorithms	Yes	-	-
30.3.2	Namespace this_thread	Yes	-	-
30.4	Mutual exclusion	Partial	-	-
30.4.1	Mutex requirements	Yes	-	-

Section	Description	Status	Comments	Ticket
30.4.1.1	In general	Yes	-	-
30.4.1.2	Mutex types	Yes	-	-
30.4.1.2.1	Class mutex	Yes	-	-
30.4.1.2.2	Class recursive_mutex	Yes	-	-
30.4.1.3	Timed mutex types	Yes	-	-
30.4.1.3.1	Class timed_mutex	Yes	-	-
30.4.1.3.1	Class recursive_timed_mutex	Yes	-	-
30.4.2	Locks	Yes	-	-
30.4.2.1	Class template lock_guard	Yes	-	-
30.4.2.2	Class template unique_lock	Yes	-	-
30.4.2.2.1	unique_lock constructors, destructor, and assignment	Yes	-	-
30.4.2.2.2	unique_lock locking	Yes	-	-
30.4.2.2.3	unique_lock modifiers	Yes	-	-
30.4.2.2.4	unique_lock observers	Yes	-	-
30.4.3	Generic locking algorithms	Partial	variadic	#6227
30.4.4	Call once	Yes	-	-
30.4.4.1	Struct once_flag	Yes	-	-
30.4.4.2	Function call_once	Yes	-	-
30.5	Condition variables	Yes	-	-
30.5.1	Class condition_variable	Yes	-	-
30.5.2	Class condition_variable_any	Yes	-	-
30.6	Futures	Yes	-	-
30.6.1	Overview	Partial	-	-
30.6.2	Error handling	Yes	-	-

Section	Description	Status	Comments	Ticket
30.6.3	Class future_error	-	-	-
30.6.4	Shared state	-	-	-
30.6.5	Class template promise	Yes	-	-
30.6.6	Class template future	Yes	-	-
30.6.7	Class template shared_future	Yes	-	-
30.6.8	Function template async	Yes	-	-
30.6.9	Class template pack-aged_task	Yes	-	-

Table 3. Extension

Section	Description	Comments
30.3.1.5.x	interrupt	-
30.3.2.x	Interruption	-
30.3.2.y	at_thread_exit	-
30.4.3.x	Generic locking algorithms begin/end	-

C++14 standard Thread library - accepted changes



Note

[Working Draft, Standard for Programming Language C++](#)

Table 4. [@<http://isocpp.org/files/papers/N3659.html> N3659 Shared locking in C++ revision 2] Conformance

Section	Description	Status	Comments
30.4.1.4	Shared Lockables Types	Yes	-
30.4.1.4.1	shared_mutex class	Yes	-
30.4.2.3	Class template shared_lock	Yes	-

C++1y TS Concurrency - On going proposals

C++ Latches and Barriers



Note

[N3600 C++ Latches and Barriers](#)



Note

[N3817 C++ Latches and Barriers](#)

Table 5. C++ Latches and Barriers Conformance

Section	Description	Status	Comments
X.1	Class latch	Partial	A new class latch has been added. The interface is a super set of the one of the proposal, taking some of the functions of the class barrier.
X.2	Class barrier	No	Even if Boost.Thread has a class <code>boost::barrier</code> it doesn't provides the same kind of services. There is an experimental <code>completion_latch</code> that could be used instead.

C++ Concurrent Queues



Note

[N3533 C++ Concurrent Queues](#)

Table 6. C++ Concurrent Queues Conformance

Section	Description	Status	Comments
X.1	Conceptual interface	Partial	The interface provided has some differences respect to this proposal. All the functions having a queue_op_status are not provided. No lock-free concrete classes
X.1.1	Basic Operations	Partial	-
X.1.1.1	push	yes	renamed push_back.
X.1.1.2	value_pop	no	renamed pull_front with two flavors.
X.1.2	Non-waiting operations	-	-
X.1.2.1	try_push	yes	renamed try_push_back
X.1.2.2	try_pop	yes	renamed try_pull_back
X.1.3	Non-blocking operations	-	-
X.1.3.1	nonblocking_push	Yes	renamed nonblock-ing_push_back
X.1.3.2	nonblocking_pop	Yes	renamed nonblock-ing_pull_front
X.1.4	Push-front operations	No	-
X.1.5	Closed queues	Partial	-
X.1.5.1	close	Yes	-
X.1.5.2	is_closed	Yes	renamed closed
X.1.5.3	wait_push	Yes	renamed wait_push_back
X.1.5.4	wait_pop	Yes	renamed wait_pull_front
X.1.5.5	wait_push_front	no	-
X.1.5.6	wait_pop_back	no	-
X.1.5.6	open	no	-
X.1.6	Empty and Full Queues	Yes	-
X.1.6.1	is_empty	Yes	-
X.1.6.2	is_full	Yes	Added capacity
X.1.7	Queue Names	No	Not considered a must for the time been.

Section	Description	Status	Comments
X.1.8	Element Type Requirements	Yes?	-
X.1.9	Exception Handling	Yes?	-
X.1.10	Queue Ordering	Yes?	-
X.1.11	Lock-Free Implementations	No	waiting to stabilize the lock-based interface. Will use Boost.LockFree once it is Boost.Move aware.
X.2	Concrete queues	Partial	-
X.2.1	Locking Buffer Queue	Partial	classes <code>sync_queue</code> and <code>sync_bounded_queue</code> .
X.2.1	Lock-Free Buffer Queue	No	waiting to stabilize the lock-based interface. Will use Boost.LockFree once it is Boost.Move aware.
X.3	Additional Conceptual Tools	No	-
X.3.1	Fronts and Backs	No	-
X.3.2	Streaming Iterators	No	-
X.3.3	Storage Iterators	No	-
X.3.4	Binary Interfaces	No	-
X.3.4	Managed Indirection	No	-

Asynchronous Executors

While Boost.Thread implementation of executors would not use dynamic polymorphism, it is worth comparing with the current trend on the standard.



Note

[N3785 Executors and Schedulers](#)

Table 7. Asynchronous Executors

Section	Description	Status	Comments
V.1.1	Class executor	Yes	-
V.1.1	add	Yes	renamed with a function template submit
V.1.1	num_of_pendin_closures	No	
V.1.2	Class sceduled_executor	No	-
V.1.2	add_at	No	renamed with a function template submit_at
V.1.2	add_after	No	renamed with a function template submit_after
V.2	Concrete executor classes	No	-
V.2.1	thread_pool	Yes	static version Basic_thread_pool, dynamic one execduler_adaptor<basic_thread_pool>
V.2.2	serial_executor	yes	-
V.2.3	loop_executor	Yes	static version loop_scheduler, dynamic one execduler_adaptor<loop_scheduler>
V.2.4	inline_executor	Yes	static version inline_executor, dynamic one execduler_adaptor<inline_executor>
V.2.5	thread_executor	Yes	static version thread_executor, dynamic one execduler_adaptor<thread_executor>

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



Note

[N3857-Improvements to std::future<T> and Related APIs](#)



Note

These functions are based on [N3634 - Improvements to std::future<T> and related APIs](#) C++1y proposal by N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani.

Table 8. Improvements to `std::future<T>` and related APIs]

Section	Description	Status	Comments
30.6.6	Class template future	Partial	-
30.6.6	unwrap constructor	Yes	-
30.6.6	then	Yes	-
30.6.6	unwrap	Yes	-
30.6.6	ready	Partial	is_ready
30.6.7	Class template shared_future	Partial	-
30.6.6	unwrap constructor	Yes	-
30.6.7	then	Yes	-
30.6.7	unwrap	No	#XXXXX
30.6.7	ready	Partial	is_ready
30.6.X	Function template when_all	Partial	interface not complete #7447
30.6.X	Function template when_any	Partial	interface not complete #7446
30.6.X	Function template when_any_swapped	No	#XXXXX
30.6.X	Function template make_ready_future	Yes	-
30.6.8	Function template async	Yes	-

C++ Stream Mutexes - C++ Stream Guards

While Boost.Thread implementation of stream mutexes differ in the approach, it is worth comparing with the current trend on the standard.



Note

These functions are based on [N3535 - C++ Stream Mutexes](#) by Lawrence Crowl.



Note

This proposal has been replaced already by [N3678 - C++ Stream Guards](#), which has been replaced by [N3665 - Un-interleaved String Output Streaming](#) and [N3750 - C++ Ostream Buffers](#)

Table 9. C++ Stream Mutexes Conformance

Section	Description	Status	Comments
X.1	Class template stream_mutex	Partial	Renamed <code>externally_locked_stream<></code>
X.2.1	constructor	Partial	<code>externally_locked_stream</code> needs a mutex in addition as argument.
X.2.2	lock	yes	-
X.2.3	unlock	yes	-
X.2.4	try_lock	yes	-
X.2.5	hold	Yes	-
X.2.6	bypass	Yes	-
X.2	Class template stream_guard	Yes	-
X.2.1	stream_guard	Yes	-
X.2.2	~stream_guard	Yes	-
X.2.3	bypass	Yes	-
X.3	Stream Operators	Yes	-
X.4	Predefined Objects	No	-