Survey of Multi-Threaded Programming Support in C++11 and Boost

C++ Now 2013

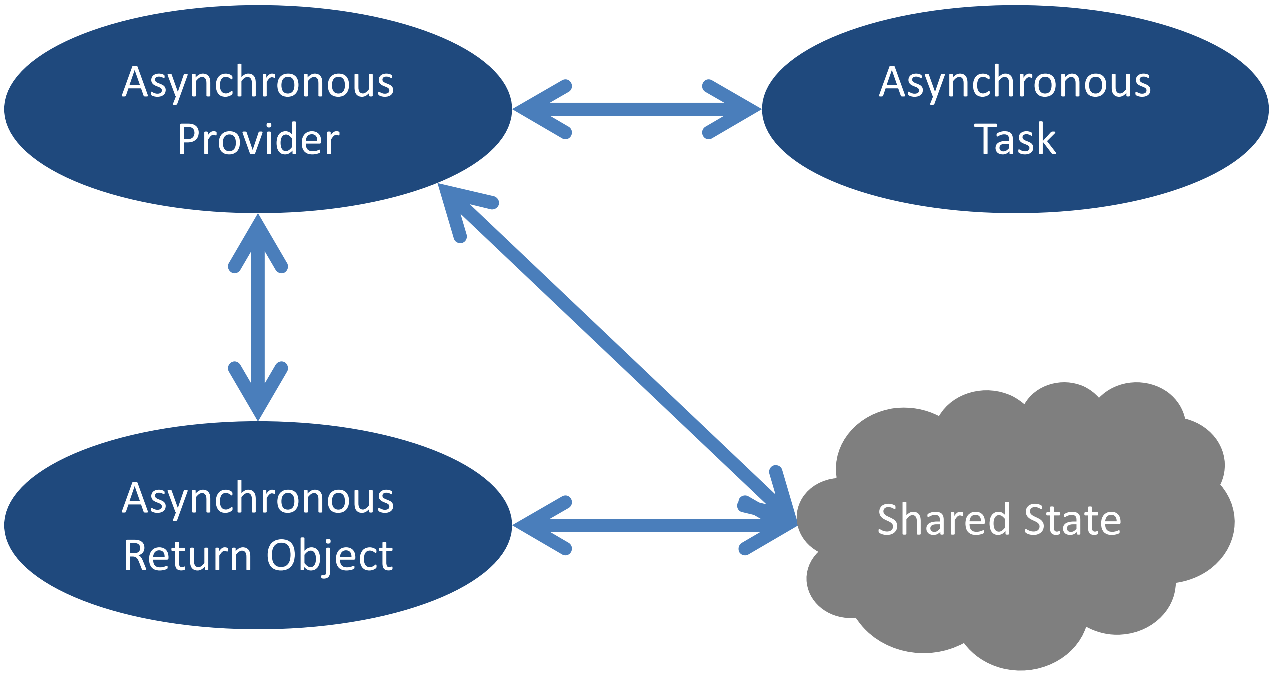
Rob Stewart

Overview

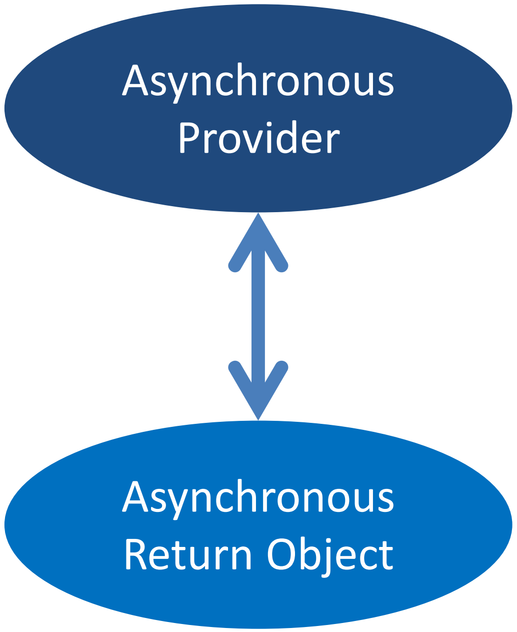
- Asynchronous computations
- Threads
- Synchronization primitives

Asynchronous Computations

Asynchronous Computations

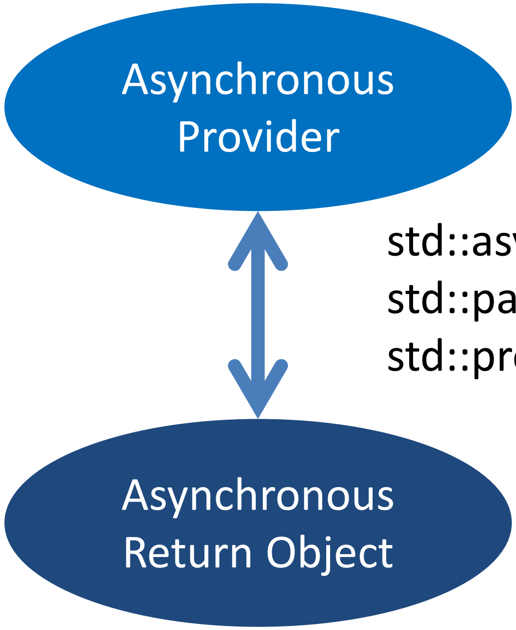


Asynchronous Return Objects



std::future
std::shared_future

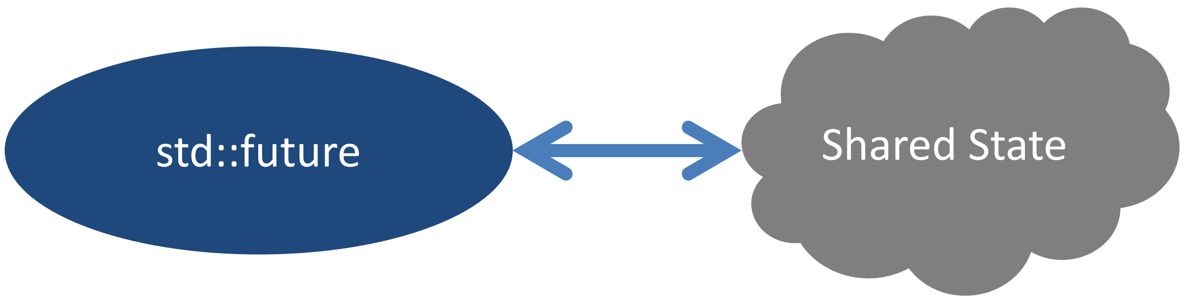
Asynchronous Providers



std::async()
std::packaged_task
std::promise

Asynchronous Return Objects

std::future



std::future

- Gets result from asynchronous task
  - Can block until result ready
  - Can wait a limited time for result
- One-to-one association with
  - Asynchronous provider
  - Shared state
- Once retrieve value, no longer available

std::future Synopsis

```
// #include <future>

template <class T>
struct std::future
{
    future();

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

    future(future &&);
    future & operator =(future &&);
    . . .
```


std::future Synopsis

```
. . .
T get();      // std::future<T>
T & get();    // std::future<T &>
void get();   // std::future<void>

bool valid() const;

std::shared_future<T> share();
. . .
```


std::future Synopsis (cont.)

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

    template <class Repr, class Period>
    std::future_status wait_for(
        std::chrono::duration<Repr,Period> const &) const;

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

std::future_status

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

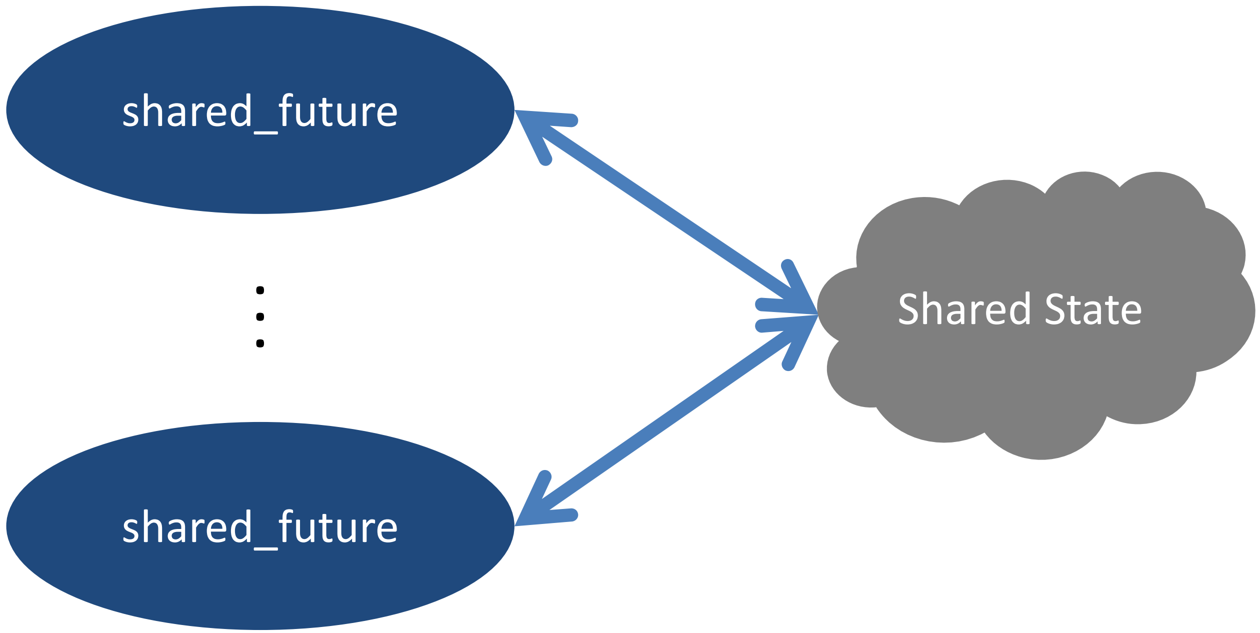
- **deferred**: Callable not yet invoked
- **ready**: Result ready
- **timeout**: Timeout exceeded before result ready

boost::future

- then() member template invokes callable when shared state ready
- get_state() reports shared state's status
  - Unitialized
  - Waiting
  - Ready
  - Moved
- Swappable

std::shared_future

std::shared_future



std::shared_future Differences

- Copyable: Copies retrieve same value
- Can retrieve value repeatedly
- Access to shared state is synchronized
- Members not synchronized (give copy to each thread)

std::shared_future Synopsis

```
// #include <future>

template <class T>
struct std::shared_future
{
    shared_future();
    shared_future(future<T> &&);
    shared_future & operator =(future<T> &&);

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

    shared_future(shared_future &&);
    shared_future & operator =(shared_future &&);
    . . .
```


std::shared_future Synopsis (cont.)

```
    . . .
    // as for std::future:
    get;
    valid;
    wait;
    wait_for;
    wait_until;

    bool is_ready() const;
    bool has_exception() const;
    bool has_value() const;

    void swap(shared_future &);
};
```


boost::shared_future

- get_state() reports shared state's status
  - Uninitialized
  - Waiting
  - Ready
  - Moved

Asynchronous Providers

std::async()

std::async()

- Runs asynchronous task
  - “Immediately,” in separate thread
  - On demand, in calling thread
- Returns result some time in future
- Simplified usage versus std::thread

Using std::async()

```
#include <future>

int f(double);

int main()
{
    std::future<int> retval(std::async(f, 1.0));
    // do other things
    return retval.get();
}
```


std::async() Callable

- Function pointer
- Lambda
- Member function pointer
  - Object pointer passed as argument to std::async()
- (Move-only) function object

std::async() Synopsis

```
// #include <future>

template <class F, class... Args>
std::future<class std::result_of<F(Args...)>::type>
std::async(std::launch, F &&, Args &&...);

template<class F, class... Args>
std::future<class std::result_of<F(Args...)>::type>
std::async(F &&, Args &&...);
```


std::async() Launch Policy

```
enum class std::launch
{
  async, deferred, sync=deferred, any=async|deferred
};
```

- **async**: Spawn new thread to run callable
- **deferred**: Invoke by thread asking for result
- **any**: Implementation defined

Surprising std::async() Side Effect

```
#include <chrono>
#include <future>
#include <iostream>
using namespace std::chrono;

int main()
{
    auto const start(std::chrono::system_clock::now());
    std::async(std::launch::async, []
    {
        std::this_thread::sleep_for(std::chrono::hours(1));
    });
    auto const elapsed(duration_cast<minutes>(
        system_clock::now() - start));
    std::cout << elapsed.count() << std::endl;
}
```

Unsurprising Equivalent

```
int main()
{
    auto const start(std::chrono::system_clock::now());
    {
        std::future<void> retval(std::async(std::launch::async, []
        {
            std::this_thread::sleep_for(std::chrono::hours(1));
        }));
        retval.get();
    }
    auto const elapsed(duration_cast<minutes>(
        system_clock::now() – start));
    std::cout << elapsed.count() << std::endl;
}
```

boost::async() Synopsis

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

template <class F>
boost::future<class boost::result_of<class boost::decay<F>::type()>::type>
boost::async(boost::launch, F &&);

template<class F>
boost::future<class boost::result_of<class boost::decay<F>::type()>::type>
boost::async(F &&);
```


Variadic boost::async()

- For C++11 platform with
  - Variadic templates
  - rvalue references
  - decltype
  - <tuple>
- BOOST_THREAD_PROVIDES_SIGNATURE_PACKAGED_TASK defined

std::packaged_task

std::packaged_task

- Packages a callable for asynchronous invocation
  - std::thread thread procedure
  - Function argument
  - Direct invocation
  - Store in container for later use
- Get future before invocation

std::packaged_task Usage

```
#include <future>

int f(double);

int main()
{
    std::packaged_task<int(double)> task(f);
    std::future<int> retval(task.get_future());
    // 1. hand off task to another thread
    // 2. do things in current thread
    return retval.get();
}
```


std::packaged_task Synopsis

```
// #include <future>

template <class T> struct std::packaged_task; // undefined

template <class ResultType, class... ArgTypes>
struct std::packaged_task<ResultType(ArgTypes...)>
{
    packaged_task();

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

    packaged_task(packaged_task &&);
    packaged_task & operator =(packaged_task &&);
    ...
```

std::packaged_task Synopsis

```
. . .
template <class F>
explicit packaged_task(F);

template <class F, class Alloc>
packaged_task(std::allocator_arg_t, Alloc const &, F);

template <class R, class... Args>
packaged_task(R (*)(Args...));

void swap(packaged_task &);
. . .
```


std::packaged_task Synopsis

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

    std::future<result_type> get_future();

    void operator ()(ArgTypes...);

    void make_ready_at_thread_exit(ArgTypes...);

    void reset();
};
```


boost::packaged_task

- Exact match function pointer constructor
- valid() equivalent to explicit bool conversion operator
- Can add callback to invoke if an associated future waits for the asynchronous result

std::promise

std::promise

- Vends a std::future (asynchronous provider)
- Not unlike std::packaged_task:
  - Get future first
  - Use promise to set value or exception
- Low level provider

std::promise Synopsis

```
// #include <future>

template <class T>
struct std::promise
{
    promise();
    ~promise();

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

    promise(promise &&);
    promise & operator =(promise &&);
    . . .
```


std::promise Synopsis (cont.)

```
. . .
template <class Alloc>
promise(std::allocator_arg_t, Alloc const &);

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

std::promise Synopsis (cont.)

```
  . . .
  std::future<T> get_future();

  void set_value(T const &); // promise<T>
  void set_value(T &&);      // promise<T>
  void set_value(T &);       // promise<T &>
  void set_value();          // promise<void>

  void set_value_at_thread_exit(T const &);    // promise<T>
  void set_value_at_thread_exit(T &&);         // promise<T>
  void set_value_at_thread_exit(T &);          // promise<T &>
  void set_value_at_thread_exit();             // promise<void>
};
```

std::promise Synopsis (cont.)

```
    ...
    void set_exception(T const &);     // promise<T>
    void set_exception(T &&);          // promise<T>
    void set_exception(T &);           // promise<T &>
    void set_exception();              // promise<void>

    void set_exception_at_thread_exit(T const &);   // promise<T>
    void set_exception_at_thread_exit(T &&);        // promise<T>
    void set_exception_at_thread_exit(T &);         // promise<T &>
    void set_exception_at_thread_exit();            // promise<void>
};
```


boost::promise

- Can add callback to invoke if an associated future waits for the asynchronous result

Threads

std::thread

- General way to run asynchronous task
  - Represents a thread of execution (except when default constructed)
  - Assembly language of threading
- Has unique ID
- Can be joined
- Can be detached
- Exposes native handle

std::thread Synopsis

```
// #include <thread>

struct std::thread
{
    thread();
    ~thread();

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

    thread(thread &&);
    thread & operator =(thread &&);
    . . .
```

std::thread Synopsis

```
. . .
template<class F>
explicit thread(F);

template<class F, class Args...>
thread(F, Args...);

void swap(thread &);
. . .
```


std::thread Synopsis (cont.)

```
. . .
    id get_id() const;

    bool joinable() const;

    void join();

    void detach();

    native_handle_type native_handle();

    static unsigned hardware_concurrency();
};
```


std::thread Semantics

- Constructor with extra parameters works like std::bind
  - Arguments copied to internal storage
  - Arguments not converted to parameter type
  - Can create dangling references
  - Can use std::cref() and std::ref()
- Destructor calls std::terminate() if joinable()
- Like main(), unhandled exceptions trigger std::terminate()

boost::thread

- Interruptible
  - Effected by boost::thread_interrupted exceptions
  - Only triggered at *interruption points*
- Supports thread attributes
- Adds time-limited joins
- Destructor behavior controlled by BOOST_THREAD_PROVIDES_THREAD_DESTRUCTOR_CALLS_TERMINATE_IF_JOINABLE
  - Defined: calls std::terminate() if joinable
  - Undefined: calls detach()

Namespace std::this_thread

Namespace std::this_thread Synopsis

```
namespace std::this_thread
{
    thread::id get_id();

    void yield();

    template<class Repr,class Period>
    void sleep_for(std::chrono::duration<Repr,Period> const &);

    template<class Clock,class Duration>
    void sleep_until(std::chrono::time_point<Clock,Duration> const &);
}
```


Namespace boost::this_thread Synopsis

```
namespace boost::this_thread
{
    // as for std::this_thread

    void interruption_point();
    bool interruption_requested() noexcept;
    bool interruption_enabled() noexcept;

    class disable_interruption;
    class restore_interruption;
}
```


Tying It Together: Implementing async()

Implementing async()

```
template <class F, class ...Args>
std::future<typename std::result_of<F(Args...)>::type
async(F _f, Args... _args)
{
    std::promise<std::result_of<F(Args...)>::type> promise;
    auto future(promise.get_future());
    std::thread thread(/*next slide*/);
    thread.detach();
    return std::move(future);
}
```


Implementing async() (cont.)

```
...
std::promise<std::result_of<F(Args...)>::type> promise;
auto future(promise.get_future());
std::thread thread(
    [] (std::promise<return_type> && _promise,
        F _f, Arg &&... _args)
    {
        // next slide
    }, std::move(promise), _f, std::forward<Args>(_args)...);
thread.detach();
...
```

Implementing async() (cont.)

```
...
std::thread thread(
    [] (std::promise<return_type> && _promise,
        F _f, Arg &&... _args)
    {
        try
        {
            _promise.set_value(_f(std::forward<Args>(_args)...));
        }
        catch (...)
        {
            _promise.set_exception(std::current_exception());
        }
    }, std::move(promise), _f, std::forward<Args>(_args)...);
...
```


Synchronization Primitives

Synchronization Primitives

- Lock concepts
- Mutexes
- Guards
- Condition Variables
- One-time Invocation

Lock Concepts

Lockable

```
struct Lockable
{
    void lock();

    bool try_lock();

    void unlock();
};
```

TimedLockable

```
struct TimedLockable
{
    // as for Lockable

    template <class Repr,class Period>
    bool try_lock_for(std::chrono::duration<Repr,Period> const &);

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

Mutexes

Lockable Mutexes

- std::mutex: Unique lock ownership
- std::recursive_mutex
  - lock() and try_lock()
    - First call acquires mutex
    - Increment lock count
  - unlock()
    - Decrements lock count
    - Releases mutex when count is zero

TimedLockable Mutexes

- std::timed_mutex: Unique lock ownership
- std::recursive_timed_mutex
  - lock() and try_lock()
    - First call acquires mutex
    - Increment lock count
  - unlock()
    - Decrements lock count
    - Releases mutex when count is zero

Mutex Semantics

- DefaultConstructable
- **Not** CopyConstructable or CopyAssignable
- **Not** MoveConstructable or MoveAssignable
- **Not** Swappable

Guards

std::lock_guard

std::lock_guard Semantics

- RAII class template
- Constructor locks or adopts Lockable
- Destructor unlocks Lockable

std::lock_guard Synopsis

```
// #include <mutex>

template <class M>
struct std::lock_guard
{
    typedef M mutex_type;

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

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


    ~lock_guard();
};
```


boost::lock_guard

- No differences

std::unique_lock

std::unique_lock

- RAII lock guard
- TimedLockable
- Flexible construction
  - Lock Lockable
  - Try to lock Lockable
  - Assume Lockable is locked
- Destructor unlocks if locked

std::unique_lock Synopsis

```
// #include <mutex>

template <class M>
struct std::unique_lock
{
    typedef M mutex_type;

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

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

    ~unique_lock();
    . . .
```


std::unique_lock Synopsis

```
. . .
explicit unique_lock(mutex_type &);
unique_lock(mutex_type &, adopt_lock_t);
unique_lock(mutex_type &, defer_lock_t);
unique_lock(mutex_type &, try_to_lock_t);

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

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

std::unique_lock Synopsis (cont.)

```
. . .
// Lockable
void lock();
bool try_lock();
void unlock();

// TimedLockable
template<typename Repr, typename Period>
bool try_lock_for(std::chrono::duration<Repr,Period> const &);

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


std::unique_lock Synopsis (cont.)

```
    . . .
    explicit operator bool() const;
    bool owns_lock() const;

    Mutex * release();

    Mutex * mutex() const;
};
```


boost::unique_lock

- No differences

Locking Function Templates

Locking Function Motivation

- Locks must be locked in the same sequence to avoid deadlocks
- If using try_lock(), must unlocking acquired locks, if one fails
- Must account for exceptions

Locking Functions Synopses

```
// #include <mutex>

template <class class Lockable1, class... Lockables>
void std::lock(Lockable1 &, Lockables &...);

template <class class Lockable1, class... Lockables>
int std::try_lock(Lockable1 &, Lockables &...);
```


Boost Locking Functions

- Limited to five locks
- Iterator-based overloads

std::condition_variable

std::condition_variable

- Used to notify threads of state changes
- Changer
  - Acquire mutex
  - Change state
  - Notify one or all waiting threads
- Waiter
  - Acquire mutex
  - Wait on condition variable
  - Examine state

std::condition_variable Synopsis

```
// #include <condition_variable>

struct std::condition_variable
{
    condition_variable();

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

    . . .
```


std::condition_variable Synopsis

```
. . .
void notify_one();

void notify_all();

void wait(std::unique_lock<std::mutex> &);

template <class Pred>
void wait(std::unique_lock<std::mutex> &, Pred);
. . .
```


std::condition_variable Synopsis (cont.)

```
. . .
template <class Repr, class Period>
cv_status wait_for(std::unique_lock<std::mutex> &,
    std::chrono::duration<Repr,Period> const &);

template <class Repr, class Period, class Pred>
bool wait_for(std::unique_lock<std::mutex> &,
    std::chrono::duration<Repr,Period> const &, Pred);
. . .
```

std::cv_status

```
enum class std::cv_status
{
  no_timeout, timeout
};
```

- **timeout**: Timeout exceeded before result ready
- **no_timeout**
  - Result ready before timeout
  - Spuriously awakened

std::condition_variable Synopsis (cont.)

```
  . . .
  template <class Clock, class Duration>
  cv_status wait_until(std::unique_lock<std::mutex> &,
      std::chrono::time_point<Clock, Duration> const &);

  template <class Clock, class Duration, class Pred>
  bool wait_until(std::unique_lock<std::mutex> &,
      std::chrono::time_point<Clock, Duration> const &, Pred);
};
```

boost::condition_variable

- No differences

std::condition_variable_any

std::condition_variable_any

- Like std::condition_variable except…
- All wait functions use arbitrary Lockable type

std::condition_variable_any Synopsis

```
// #include <condition_variable>

struct std::condition_variable_any
{
    // as for std::condition_variable

    template <class Lockable>
    void wait(Lockable &);

    template <class Lockable, class Pred>
    void wait(Lockable &, Pred);
    . . .
```


std::condition_variable_any Synopsis (cont.)

```
. . .
template <class Lockable, class Repr, class Period>
cv_status wait_for(Lockable &,
    std::chrono::duration<Repr,Period> const &);

template <class Lockable, class Repr, class Period, class Pred>
bool wait_for(Lockable &,
    std::chrono::duration<Repr,Period> const &, Pred);
. . .
```


std::condition_variable_any Synopsis (cont.)

```
    ...
    template <class Lockable, class Clock, class Duration>
    cv_status wait_until(Lockable &,
        std::chrono::time_point<Clock,Duration> const &);

    template <class Lockable, class Clock, class Duration, class Pred>
    bool wait_until(Lockable &,
        std::chrono::time_point<Clock,Duration> const &, Pred);
};
```


boost::condition_variable_any

- No differences

Why std::condition_variable and std::condition_variable_any?

- std::condition_variable
  - Uses std::unique_lock<std::mutex> only
  - Allows for optimizations not otherwise possible
- std::condition_variable_any
  - Uses an arbitrary Lockable
  - Can be less efficient
  - Typically implemented in terms of the former

One-time Invocation

One-time Invocation

- Invoke callable exactly once
  - One-time initialization
  - No races
  - No deadlocks
- Strong exception guarantee
  - Callable's exceptions propagated
  - Call considered not done if exception

One-time Invocation Usage

```
#include <mutex>

std::once_flag one_time;

void f(int, char const *);

int main()
{
    std::call_once(one_time, f, 12, "Test");
}
```


std::call_once

```
// #include <mutex>

template<class F, class... Args>
void std::call_once(std::once_flag &, F, Args...);
```


Boost One-time Invocation

- boost::once_flag requires static initialization from BOOST_ONCE_INIT
- boost::call_once() only accepts zero-argument callables

Summary

- Asynchronous computations
- Threads
- Synchronization primitives

Questions?

Resources

- http://www.boost.org/libs/thread/index.html
- http://www.stdthread.co.uk/doc/
- http://en.cppreference.com/w/cpp/thread
- *C++ Concurrency in Action: Practical Multithreading* (Williams)

