C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

Table of Contents

4 Synchronizing concurrent operations

This chapter covers

- Waiting for an event
- Waiting for one-off events with futures
- Waiting with a time limit
- Using synchronization of operations to simplify code

In the last chapter, we looked at various ways of protecting data that's shared between threads. But sometimes you don't just need to protect the data but also to synchronize actions on separate threads. One thread might need to wait for another thread to complete a task before the first thread can complete its own, for example. In general, it's common to want a thread to wait for a specific event to happen or a condition to be `true`. Although it would be possible to do this by periodically checking a "task complete" flag or something similar stored in shared data, this is far from ideal. The need to synchronize operations between threads like this is such a common scenario that the C++ Standard Library provides facilities to handle it, in the form of *condition variables* and *futures*.

In this chapter I'll discuss how to wait for events with condition variables and futures and how to use them to simplify the synchronization of operations.

4.1 Waiting for an event or other condition

Suppose you're traveling on an overnight train. One way to ensure you get off at the right station would be to stay awake all night and pay attention to where the train stops. You wouldn't miss your station, but you'd be tired when you got there. Alternatively, you could look at the timetable to see when the train is supposed to arrive, set your alarm a bit before, and go to sleep. That would be OK; you wouldn't miss your stop, but if the train got delayed, you'd wake up too early. There's also the possibility that your alarm clock's batteries would die, and you'd sleep too long and miss your station. What would be ideal is if you could just go to sleep and have somebody or something wake you up when the train gets to your station, whenever that is.

How does that relate to threads? Well, if one thread is waiting for a second thread to complete a task, it has several options. First, it could just keep checking a flag in shared data (protected by a mutex) and have the second thread set the flag when it completes the task. This is wasteful on two counts: the thread consumes valuable processing time repeatedly checking the flag, and when the mutex is locked by the waiting thread, it can't be locked by any other thread. Both of these work against the thread doing the waiting, because they limit the resources available to the thread being waited for and even prevent it from setting the flag when it's done. This is akin to staying awake all night talking to the train driver: he has to drive the train more slowly because you keep distracting him, so it takes longer to get there. Similarly, the waiting thread is consuming resources that could be used by other threads in the system and may end up waiting longer than necessary.

A second option is to have the waiting thread sleep for small periods between the checks using the `std::this_thread::sleep_for()` function (see section 4.3):

```
bool flag;
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
    }
}
```

❶ Unlock the mutex

❷ Sleep for 100 ms

❸ Relock the mutex

In the loop, the function unlocks the mutex ❶ before the sleep ❷ and locks it again afterward ❸, so another thread gets a chance to acquire it and set the flag.

This is an improvement, because the thread doesn't waste processing time while it's sleeping, but it's hard to get the sleep period right. Too short a sleep in between checks and the thread still wastes processing time checking; too long a sleep and the thread will keep on sleeping even when the task it's waiting for is complete, introducing a delay. It's rare that this oversleeping will have a direct impact on the operation of

the program, but it could mean dropped frames in a fast-paced game or overrunning a time slice in a real-time application.

The third, and preferred, option is to use the facilities from the C++ Standard Library to wait for the event itself. The most basic mechanism for waiting for an event to be triggered by another thread (such as the presence of additional work in the pipeline mentioned previously) is the *condition variable*. Conceptually, a condition variable is associated with some event or other *condition*, and one or more threads can *wait* for that condition to be satisfied. When some thread has determined that the condition is satisfied, it can then *notify* one or more of the threads waiting on the condition variable, in order to wake them up and allow them to continue processing.

4.1.1 Waiting for a condition with condition variables

The Standard C++ Library provides not one but *two* implementations of a condition variable: `std::condition_variable` and `std::condition_variable_any`. Both of these are declared in the `<condition_variable>` library header. In both cases, they need to work with a mutex in order to provide appropriate synchronization; the former is limited to working with `std::mutex`, whereas the latter can work with anything that meets some minimal criteria for being mutex-like, hence the `_any` suffix. Because `std::condition_variable_any` is more general, there's the potential for additional costs in terms of size, performance, or operating system resources, so `std::condition_variable` should be preferred unless the additional flexibility is required.

So, how do you use a `std::condition_variable` to handle the example in the introduction—how do you let the thread that's waiting for work sleep until there's data to process? The following listing shows one way you could do this with a condition variable.

Listing 4.1 Waiting for data to process with a `std::condition_variable`

```
std::mutex mut;
std::queue<data_chunk> data_queue;    <-1
std::condition_variable data_cond;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);              <-2
        data_cond.notify_one();    <-3
    }
}

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);    <-4
```

```
        data_cond.wait(
            lk,[]{return !data_queue.empty();});    <-5
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();        <-6
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

First off, you have a queue ❶ that's used to pass the data between the two threads. When the data is ready, the thread preparing the data locks the mutex protecting the queue using a `std::lock_guard` and pushes the data onto the queue ❷. It then calls the `notify_one()` member function on the `std::condition_variable` instance to notify the waiting thread (if there is one) ❸.

On the other side of the fence, you have the processing thread. This thread first locks the mutex, but this time with a `std::unique_lock` rather than a `std::lock_guard` ❹—you'll see why in a minute. The thread then calls `wait()` on the `std::condition_variable`, passing in the lock object and a lambda function that expresses the condition being waited for ❺. Lambda functions are a new feature in C++11 that allows you to write an anonymous function as part of another expression, and they're ideally suited for specifying predicates for standard library functions such as `wait()`. In this case, the simple lambda function `[]{return !data_queue.empty();}` checks to see if the `data_queue` is not `empty()`—that is, there's some data in the queue ready for processing. Lambda functions are described in more detail in appendix A, section A.5.

The implementation of `wait()` then checks the condition (by calling the supplied lambda function) and returns if it's satisfied (the lambda function returned `true`). If the condition isn't satisfied (the lambda function returned `false`), `wait()` unlocks the mutex and puts the thread in a blocked or waiting state. When the condition variable is notified by a call to `notify_one()` from the data-preparation thread, the thread wakes from its slumber (unblocks it), reacquires the lock on the mutex, and checks the condition again, returning from `wait()` with the mutex still locked if the condition has been satisfied. If the condition hasn't been satisfied, the thread unlocks the mutex and resumes waiting. This is why you need the `std::unique_lock` rather than the `std::lock_guard`—the waiting thread must unlock the mutex while it's waiting and lock it again afterward, and `std::lock_guard` doesn't provide that flexibility. If the mutex remained locked while the thread was sleeping, the data-preparation thread wouldn't be able to lock the mutex to add an item to the queue, and the waiting thread would never be able to see its condition satisfied.

Listing 4.1 uses a simple lambda function for the wait ❺, which checks to see if the queue is not empty, but any function or callable object could be passed. If you already have a function to check the condition (perhaps because it's more complicated than a simple test like this), then this function can be passed in directly; there's no need

to wrap it in a lambda. During a call to `wait()`, a condition variable may check the supplied condition any number of times; however, it always does so with the mutex locked and will return immediately if (and only if) the function provided to test the condition returns `true`. When the waiting thread reacquires the mutex and checks the condition, if it isn't in direct response to a notification from another thread, it's called a *spurious wake*. Because the number and frequency of any such spurious wakes are by definition indeterminate, it isn't advisable to use a function with side effects for the condition check. If you do so, you must be prepared for the side effects to occur multiple times.

The flexibility to unlock a `std::unique_lock` isn't just used for the call to `wait()`; it's also used once you have the data to process but before processing it ❻. Processing data can potentially be a time-consuming operation, and as you saw in chapter 3, it's a bad idea to hold a lock on a mutex for longer than necessary.

Using a queue to transfer data between threads as in listing 4.1 is a common scenario. Done well, the synchronization can be limited to the queue itself, which greatly reduces the possible number of synchronization problems and race conditions. In view of this, let's now work on extracting a generic thread-safe queue from listing 4.1.

4.1.2 Building a thread-safe queue with condition variables

If you're going to be designing a generic queue, it's worth spending a few minutes thinking about the operations that are likely to be required, as you did with the thread-safe stack back in section 3.2.3. Let's look at the C++ Standard Library for inspiration, in the form of the `std::queue<>` container adaptor shown in the following listing.

Listing 4.2 `std::queue` interface

```
template <class T, class Container = std::deque<T> >
class queue {
public:
    explicit queue(const Container&);
    explicit queue(Container&& = Container());

    template <class Alloc> explicit queue(const Alloc&);
    template <class Alloc> queue(const Container&, const Alloc&);
    template <class Alloc> queue(Container&&, const Alloc&);
    template <class Alloc> queue(queue&&, const Alloc&);

    void swap(queue& q);

    bool empty() const;
    size_type size() const;

    T& front();
    const T& front() const;
    T& back();
    const T& back() const;

    void push(const T& x);
    void push(T&& x);
```

```
    void pop();
    template <class... Args> void emplace(Args&&... args);
};
```

If you ignore the construction, assignment and swap operations, you're left with three groups of operations: those that query the state of the whole queue (`empty()` and `size()`), those that query the elements of the queue (`front()` and `back()`), and those that modify the queue (`push()`, `pop()` and `emplace()`). This is the same as you had back in section 3.2.3 for the stack, and therefore you have the same issues regarding race conditions inherent in the interface. Consequently, you need to combine `front()` and `pop()` into a single function call, much as you combined `top()` and `pop()` for the stack. The code from listing 4.1 adds a new nuance, though: when using a queue to pass data between threads, the receiving thread often needs to wait for the data. Let's provide two variants on `pop()`: `try_pop()`, which tries to pop the value from the queue but always returns immediately (with an indication of failure) even if there wasn't a value to retrieve, and `wait_and_pop()`, which will wait until there's a value to retrieve. If you take your lead for the signatures from the stack example, your interface looks like the following.

Listing 4.3 The interface of your `threadsafe_queue`

```
#include <memory>                    <-- For std::shared_ptr

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue&);
    threadsafe_queue& operator=(
        const threadsafe_queue&) = delete;      <-- Disallow assignment for simplicity

    void push(T new_value);

    bool try_pop(T& value);          <-- (1)
    std::shared_ptr<T> try_pop();    <-- (2)

    void wait_and_pop(T& value);
    std::shared_ptr<T> wait_and_pop();

    bool empty() const;
};
```

As you did for the stack, you've cut down on the constructors and eliminated assignment in order to simplify the code. You've also provided two versions of both `try_pop()` and `wait_for_pop()`, as before. The first overload of `try_pop()` (1) stores the retrieved value in the referenced variable, so it can use the return value for status; it returns `true` if it retrieved a value and `false` otherwise (see section A.2). The second overload (2) can't do this, because it returns the retrieved value directly. But the returned pointer can be set to `NULL` if there's no value to retrieve.

So, how does all this relate to listing 4.1? Well, you can extract the code for `push()` and `wait_and_pop()` from there, as shown in the next listing.

Listing 4.4 Extracting `push()` and `wait_and_pop()` from listing 4.1

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
};

threadsafe_queue<data_chunk> data_queue;    <-- 1

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        data_chunk const data=prepare_data();
        data_queue.push(data);                <-- 2
    }
}

void data_processing_thread()
{
    while(true)
    {
        data_chunk data;
        data_queue.wait_and_pop(data);    <-- 3
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

The mutex and condition variable are now contained within the `threadsafe_queue` instance, so separate variables are no longer required ❶, and no external synchronization is required for the call to `push()` ❷. Also, `wait_and_pop()` takes care of the condition variable wait ❸.

The other overload of `wait_and_pop()` is now trivial to write, and the remaining functions can be copied almost verbatim from the stack example in listing 3.5. The final queue implementation is shown here.

Listing 4.5 Full class definition for a thread-safe queue using condition variables

```
#include <queue>
#include <memory>
#include <mutex>
#include <condition_variable>

template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }
```

❶ **The mutex must be mutable**

```
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=data_queue.front();
        data_queue.pop();
        return true;
    }

    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res(std::make_shared<T>(data_queue.front()));
        data_queue.pop();
        return res;
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

Even though `empty()` is a `const` member function, and the `other` parameter to the copy constructor is a `const` reference, other threads may have non-`const` references to the object, and be calling mutating member functions, so we still need to lock the mutex. Since locking a mutex is a mutating operation, the mutex object must be marked `mutable` ❶ so it can be locked in `empty()` and in the copy constructor.

Condition variables are also useful where there's more than one thread waiting for the same event. If the threads are being used to divide the workload, and thus only one thread should respond to a notification, exactly the same structure as shown in listing 4.1 can be used; just run multiple instances of the data—processing thread. When new data is ready, the call to `notify_one()` will trigger one of the threads currently executing `wait()` to check its condition and thus return from `wait()` (because you've just added an item to the `data_queue`). There's no guarantee which thread will be notified or even if there's a thread waiting to be notified; all the processing threads might be still processing data.

Another possibility is that several threads are waiting for the same event, and all of them need to respond. This can happen where shared data is being initialized, and the processing threads can all use the same data but need to wait for it to be initialized (although there are better mechanisms for this; see section 3.3.1 in chapter 3), or where the threads need to wait for an update to shared data, such as a periodic reinitialization. In these cases, the thread preparing the data can call the `notify_all()` member function on the condition variable rather than `notify_one()`. As the name suggests, this causes *all* the threads currently executing `wait()` to check the condition they're waiting for.

If the waiting thread is going to wait only once, so when the condition is `true` it will never wait on this condition variable again, a condition variable might not be the best

choice of synchronization mechanisms. This is especially true if the condition being waited for is the availability of a particular piece of data. In this scenario, a *future* might be more appropriate.

4.2 Waiting for one-off events with futures

Suppose you?re going on vacation abroad by plane. Once you get to the airport and clear the various check-in procedures, you still have to wait for notification that your flight is ready for boarding, possibly for several hours. Yes, you might be able to find some means of passing the time, such as reading a book, surfing the internet, or eating in an overpriced airport café, but fundamentally you?re just waiting for one thing: the signal that it?s time to get on the plane. Not only that, but a given flight goes only once; the next time you?re going on vacation, you?ll be waiting for a different flight.

The C++ Standard Library models this sort of one-off event with something called a *future*. If a thread needs to wait for a specific one-off event, it somehow obtains a future representing this event. The thread can then periodically wait on the future for short periods of time to see if the event has occurred (check the departures board) while performing some other task (eating in the overpriced café) in between checks. Alternatively, it can do another task until it needs the event to have happened before it can proceed and then just wait for the future to become *ready*. A future may have data associated with it (such as which gate your flight is boarding at), or it may not. Once an event has happened (and the future has become *ready*), the future can't be reset.

There are two sorts of futures in the C++ Standard Library, implemented as two class templates declared in the `<future>` library header: *unique futures* (`std::future<>`) and *shared futures* (`std::shared_future<>`). These are modeled after `std::unique_ptr` and `std::shared_ptr`. An instance of `std::future` is the one and only instance that refers to its associated event, whereas multiple instances of `std::shared_future` may refer to the same event. In the latter case, all the instances will become *ready* at the same time, and they may all access any data associated with the event. This associated data is the reason these are templates; just like `std::unique_ptr` and `std::shared_ptr`, the template parameter is the type of the associated data. The `std:future<void>`, `std::shared_future<void>` template specializations should be used where there's no associated data. Although futures are used to communicate between threads, the future objects themselves don't provide synchronized accesses. If multiple threads need to access a single future object, they must protect access via a mutex or other synchronization mechanism, as described in chapter 3. However, as you'll see in section 4.2.5, multiple threads may each access their own copy of a `std::shared_future<>` without further synchronization, even if they all refer to the same asynchronous result.

The most basic of one-off events is the result of a calculation that has been run in the background. Back in chapter 2 you saw that `std::thread` doesn't provide an easy means of returning a value from such a task, and I promised that this would be addressed in chapter 4 with futures—now it's time to see how.

4.2.1 Returning values from background tasks

Suppose you have a long-running calculation that you expect will eventually yield a useful result but for which you don't currently need the value. Maybe you've found a way to determine the answer to Life, the Universe, and Everything, to pinch an example from Douglas Adams.[1] You could start a new thread to perform the calculation, but that means you have to take care of transferring the result back, because `std::thread` doesn't provide a direct mechanism for doing so. This is where the `std::async` function template (also declared in the `<future>` header) comes in.

You use `std::async` to start an *asynchronous task* for which you don't need the result right away. Rather than giving you back a `std::thread` object to wait on, `std::async` returns a `std::future` object, which will eventually hold the return value of the function. When you need the value, you just call `get()` on the future, and the thread blocks until the future is *ready* and then returns the value. The following listing shows a simple example.

Listing 4.6 Using `std::future` to get the return value of an asynchronous task

```
#include <future>
#include <iostream>

int find_the_answer_to_ltuae();
void do_other_stuff();
int main()
{
    std::future<int> the_answer=std::async(find_the_answer_to_ltuae);
    do_other_stuff();
    std::cout<<"The answer is "<<the_answer.get()<<std::endl;
}
```

`std::async` allows you to pass additional arguments to the function by adding extra arguments to the call, in the same way that `std::thread` does. If the first argument is a pointer to a member function, the second argument provides the object on which to apply the member function (either directly, or via a pointer, or wrapped in `std::ref`), and the remaining arguments are passed as arguments to the member function. Otherwise, the second and subsequent arguments are passed as arguments to the function or callable object specified as the first argument. Just as with `std::thread`, if the arguments are rvalues, the copies are created by *moving* the originals. This allows the use of move-only types as both the function object and the arguments. See the following listing.

Listing 4.7 Passing arguments to a function with `std::async`

```
#include <string>
#include <future>
```

[1] In *The Hitchhiker's Guide to the Galaxy*, the computer Deep Thought is built to determine "the answer to Life, the Universe and Everything." The answer is 42.

```
struct X
{
    void foo(int,std::string const&);
    std::string bar(std::string const&);
};
X x;
auto f1=std::async(&X::foo,&x,42,"hello");
auto f2=std::async(&X::bar,x,"goodbye");
struct Y
{
    double operator()(double);
};
Y y;
auto f3=std::async(Y(),3.141);
auto f4=std::async(std::ref(y),2.718);
X baz(X&);
std::async(baz,std::ref(x));
class move_only
{
public:
    move_only();
    move_only(move_only&&)
    move_only(move_only const&) = delete;
    move_only& operator=(move_only&&);
    move_only& operator=(move_only const&) = delete;

    void operator()();
};
auto f5=std::async(move_only());
```

Calls p->foo(42,"hello") where p is &x

Calls tmpx.bar("goodbye") where tmpx is a copy of x

Calls tmpy(3.141) where tmpy is move-constructed from Y()

Calls y(2.718)

Calls baz(x)

Calls tmp() where tmp is constructed from std::move(move_only())

By default, it's up to the implementation whether `std::async` starts a new thread, or whether the task runs synchronously when the future is waited for. In most cases this is what you want, but you can specify which to use with an additional parameter to `std::async` before the function to call. This parameter is of the type `std::launch`, and can either be `std::launch::deferred` to indicate that the function call is to be deferred until either `wait()` or `get()` is called on the future, `std::launch::async` to indicate that the function must be run on its own thread, or `std::launch::deferred | std::launch::async` to indicate that the implementation may choose. This last option is the default. If the function call is deferred, it may never actually run. For example:

```
auto f6=std::async(std::launch::async,Y(),1.2);
auto f7=std::async(std::launch::deferred,baz,std::ref(x));
auto f8=std::async(
   std::launch::deferred | std::launch::async,
   baz,std::ref(x));
auto f9=std::async(baz,std::ref(x));
f7.wait();
```

Run in new thread

Run in wait() or get()

Implementation chooses

Invoke deferred function

As you'll see later in this chapter and again in chapter 8, the use of `std::async` makes it easy to divide algorithms into tasks that can be run concurrently. However, it's not the only way to associate a `std::future` with a task; you can also do it by wrapping the task in an instance of the `std::packaged_task<>` class template or by writing code to

explicitly set the values using the `std::promise<>` class template. `std::packaged_task` is a higher-level abstraction than `std::promise`, so I'll start with that.

4.2.2 Associating a task with a future

`std::packaged_task<>` ties a future to a function or callable object. When the `std::packaged_task<>` object is invoked, it calls the associated function or callable object and makes the future *ready*, with the return value stored as the associated data. This can be used as a building block for thread pools (see chapter 9) or other task management schemes, such as running each task on its own thread, or running them all sequentially on a particular background thread. If a large operation can be divided into self-contained sub-tasks, each of these can be wrapped in a `std::packaged_task<>` instance, and then that instance passed to the task scheduler or thread pool. This abstracts out the details of the tasks; the scheduler just deals with `std::packaged_task<>` instances rather than individual functions.

The template parameter for the `std::packaged_task<>` class template is a function signature, like `void()` for a function taking no parameters with no return value, or `int(std::string&,double*)` for a function that takes a non-`const` reference to a `std::string` and a pointer to a `double` and returns an `int`. When you construct an instance of `std::packaged_task`, you must pass in a function or callable object that can accept the specified parameters and that returns a type convertible to the specified return type. The types don't have to match exactly; you can construct a `std::packaged_task<double(double)>` from a function that takes an `int` and returns a `float` because the types are implicitly convertible.

The return type of the specified function signature identifies the type of the `std::future<>` returned from the `get_future()` member function, whereas the argument list of the function signature is used to specify the signature of the packaged task's function call operator. For example, a partial class definition for `std::packaged_task<std::string(std::vector<char>*,int)>` would be as shown in the following listing.

Listing 4.8 Partial class definition for a specialization of `std::packaged_task< >`

```
template<>
class packaged_task<std::string(std::vector<char>*,int)>
{
public:
    template<typename Callable>
    explicit packaged_task(Callable&& f);
    std::future<std::string> get_future();
    void operator()(std::vector<char>*,int);
};
```

The `std::packaged_task` object is thus a callable object, and it can be wrapped in a `std::function` object, passed to a `std::thread` as the thread function, passed to another function that requires a callable object, or even invoked directly. When the `std::packaged_task` is invoked as a function object, the arguments supplied

to the function call operator are passed on to the contained function, and the return value is stored as the asynchronous result in the `std::future` obtained from `get_future()`. You can thus wrap a task in a `std::packaged_task` and retrieve the future before passing the `std::packaged_task` object elsewhere to be invoked in due course. When you need the result, you can wait for the future to become ready. The following example shows this in action.

PASSING TASKS BETWEEN THREADS

Many GUI frameworks require that updates to the GUI be done from specific threads, so if another thread needs to update the GUI, it must send a message to the right thread in order to do so. `std:packaged_task` provides one way of doing this without requiring a custom message for each and every GUI-related activity, as shown here.

Listing 4.9 Running code on a GUI thread using `std::packaged_task`

```
#include <deque>
#include <mutex>
#include <future>
#include <thread>
#include <utility>

std::mutex m;
std::deque<std::packaged_task<void()> > tasks;

bool gui_shutdown_message_received();
void get_and_process_gui_message();

void gui_thread()                                   <-1
{
    while(!gui_shutdown_message_received())         <-2
    {
        get_and_process_gui_message();              <-3
        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> lk(m);
            if(tasks.empty())                       <-4
                continue;
            task=std::move(tasks.front());          <-5
            tasks.pop_front();
        }
        task();                                     <-6
    }
}

std::thread gui_bg_thread(gui_thread);

template<typename Func>
std::future<void> post_task_for_gui_thread(Func f)
{
    std::packaged_task<void()> task(f);             <-7
    std::future<void> res=task.get_future();        <-8
    std::lock_guard<std::mutex> lk(m);
    tasks.push_back(std::move(task));               <-9
    return res;                                     <-10
}
```

This code is very simple: the GUI thread ❶ loops until a message has been received telling the GUI to shut down ❷, repeatedly polling for GUI messages to handle ❸, such as user clicks, and for tasks on the task queue. If there are no tasks on the queue ❹, it loops again; otherwise, it extracts the task from the queue ❺, releases the lock on the queue, and then runs the task ❻. The future associated with the task will then be made ready when the task completes.

Posting a task on the queue is equally simple: a new packaged task is created from the supplied function ❼, the future is obtained from that task ❽ by calling the `get_future()` member function, and the task is put on the list ❾ before the future is returned to the caller ❿. The code that posted the message to the GUI thread can then wait for the future if it needs to know that the task has been completed, or it can discard the future if it doesn't need to know.

This example uses `std::packaged_task<void()>` for the tasks, which wraps a function or other callable object that takes no parameters and returns `void` (if it returns anything else, the return value is discarded). This is the simplest possible task, but as you saw earlier, `std::packaged_task` can also be used in more complex situations—by specifying a different function signature as the template parameter, you can change the return type (and thus the type of data stored in the future's associated state) and also the argument types of the function call operator. This example could easily be extended to allow for tasks that are to be run on the GUI thread to accept arguments and return a value in the `std::future` rather than just a completion indicator.

What about those tasks that can't be expressed as a simple function call or those tasks where the result may come from more than one place? These cases are dealt with by the third way of creating a future: using a `std::promise` to set the value explicitly.

4.2.3 Making (std::)promises

When you have an application that needs to handle a lot of network connections, it's often tempting to handle each connection on a separate thread, because this can make the network communication easier to think about and easier to program. This works well for low numbers of connections (and thus low numbers of threads). Unfortunately, as the number of connections rises, this becomes less suitable; the large numbers of threads consequently consume large numbers of operating system resources and potentially cause a lot of context switching (when the number of threads exceeds the available hardware concurrency), impacting performance. In the extreme case, the operating system may run out of resources for running new threads before its capacity for network connections is exhausted. In applications with very large numbers of network connections, it's therefore common to have a small number of threads (possibly only one) handling the connections, each thread dealing with multiple connections at once.

Consider one of these threads handling the connections. Data packets will come in from the various connections being handled in essentially random order, and likewise data packets will be queued to be sent in random order. In many cases, other parts of the application will be waiting either for data to be successfully sent or for a new batch of data to be successfully received via a specific network connection.

`std::promise<T>` provides a means of setting a value (of type `T`), which can later be read through an associated `std::future<T>` object. A `std::promise/std::future` pair would provide one possible mechanism for this facility; the waiting thread could block on the future, while the thread providing the data could use the promise half of the pairing to set the associated value and make the future *ready*.

You can obtain the `std::future` object associated with a given `std::promise` by calling the `get_future()` member function, just like with `std::packaged_task`. When the value of the promise is set (using the `set_value()` member function), the future becomes *ready* and can be used to retrieve the stored value. If you destroy the `std::promise` without setting a value, an exception is stored instead. Section 4.2.4 describes how exceptions are transferred across threads.

Listing 4.10 shows some example code for a thread processing connections as just described. In this example, you use a `std::promise<bool>/std::future<bool>` pair to identify the successful transmission of a block of outgoing data; the value associated with the future is a simple success/failure flag. For incoming packets, the data associated with the future is the payload of the data packet.

Listing 4.10 Handling multiple connections from a single thread using promises

```
#include <future>

void process_connections(connection_set& connections)
{
    while(!done(connections))    <-1
    {
        for(connection_iterator       <-2
                connection=connections.begin(),end=connections.end();
            connection!=end;
            ++connection)
        {
            if(connection->has_incoming_data())    <-3
            {
                data_packet data=connection->incoming();
                std::promise<payload_type>& p=
                    connection->get_promise(data.id);    <-4
                p.set_value(data.payload);
            }
            if(connection->has_outgoing_data())    <-5
            {
                outgoing_packet data=
                    connection->top_of_outgoing_queue();
                connection->send(data.payload);
                data.promise.set_value(true);      <-6
            }
        }
    }
}
```

The function `process_connections()` loops until `done()` returns `true` (1). Every time through the loop, it checks each connection in turn (2), retrieving incoming data if

there is any ❸ or sending any queued outgoing data ❺. This assumes that an incoming packet has some ID and a payload with the actual data in it. The ID is mapped to a `std::promise` (perhaps by a lookup in an associative container) ❹, and the value is set to the packet's payload. For outgoing packets, the packet is retrieved from the outgoing queue and actually sent through the connection. Once the send has completed, the promise associated with the outgoing data is set to `true` to indicate successful transmission ❻. Whether this maps nicely to the actual network protocol depends on the protocol; this promise/future style structure may not work for a particular scenario, although it does have a similar structure to the asynchronous I/O support of some operating systems.

All the code up to now has completely disregarded exceptions. Although it might be nice to imagine a world in which everything worked all the time, this isn't actually the case. Sometimes disks fill up, sometimes what you're looking for just isn't there, sometimes the network fails, and sometimes the database goes down. If you were performing the operation in the thread that needed the result, the code could just report an error with an exception, so it would be unnecessarily restrictive to require that everything go well just because you wanted to use a `std::packaged_task` or a `std::promise`. The C++ Standard Library therefore provides a clean way to deal with exceptions in such a scenario and allows them to be saved as part of the associated result.

4.2.4 Saving an exception for the future

Consider the following short snippet of code. If you pass in `-1` to the `square_root()` function, it throws an exception, and this gets seen by the caller:

```
double square_root(double x)
{
    if(x<0)
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}
```

Now suppose that instead of just invoking `square_root()` from the current thread,

```
double y=square_root(-1);
```

you run the call as an asynchronous call:

```
std::future<double> f=std::async(square_root,-1);
double y=f.get();
```

It would be ideal if the behavior was exactly the same; just as `y` gets the result of the function call in either case, it would be great if the thread that called `f.get()` could see the exception too, just as it would in the single-threaded case.

Well, that's exactly what happens: if the function call invoked as part of `std::async` throws an exception, that exception is stored in the future in place of a stored value, the future becomes *ready*, and a call to `get()` rethrows that stored exception.

(Note: the standard leaves it unspecified whether it is the original exception object that's rethrown or a copy; different compilers and libraries make different choices on this matter.) The same happens if you wrap the function in a `std::packaged_task`—when the task is invoked, if the wrapped function throws an exception, that exception is stored in the future in place of the result, ready to be thrown on a call to `get()`.

Naturally, `std::promise` provides the same facility, with an explicit function call. If you wish to store an exception rather than a value, you call the `set_exception()` member function rather than `set_value()`. This would typically be used in a `catch` block for an exception thrown as part of the algorithm, to populate the promise with that exception:

```
extern std::promise<double> some_promise;

try
{
    some_promise.set_value(calculate_value());
}
catch(...)
{
    some_promise.set_exception(std::current_exception());
}
```

This uses `std::current_exception()` to retrieve the thrown exception; the alternative here would be to use `std::copy_exception()` to store a new exception directly without throwing:

```
some_promise.set_exception(std::copy_exception(std::logic_error("foo ")));
```

This is much cleaner than using a `try/catch` block if the type of the exception is known, and it should be used in preference; not only does it simplify the code, but it also provides the compiler with greater opportunity to optimize the code.

Another way to store an exception in a future is to destroy the `std::promise` or `std::packaged_task` associated with the future without calling either of the set functions on the promise or invoking the packaged task. In either case, the destructor of the `std::promise` or `std::packaged_task` will store a `std::future_error` exception with an error code of `std::future_errc::broken_promise` in the associated state if the future isn't already *ready*; by creating a future you make a promise to provide a value or exception, and by destroying the source of that value or exception without providing one, you break that promise. If the compiler didn't store anything in the future in this case, waiting threads could potentially wait forever.

Up until now all the examples have used `std::future`. However, `std::future` has its limitations, not the least of which being that only one thread can wait for the result. If you need to wait for the same event from more than one thread, you need to use `std::shared_future` instead.

4.2.5 Waiting from multiple threads

Although `std::future` handles all the synchronization necessary to transfer data from one thread to another, calls to the member functions of a particular `std::future` instance are not synchronized with each other. If you access a single `std::future` object from multiple threads without additional synchronization, you have a *data race* and undefined behavior. This is by design: `std::future` models unique ownership of the asynchronous result, and the one-shot nature of `get()` makes such concurrent access pointless anyway—only one thread can retrieve the value, because after the first call to `get()` there's no value left to retrieve.

If your fabulous design for your concurrent code requires that multiple threads can wait for the same event, don't despair just yet; `std::shared_future` allows exactly that. Whereas `std::future` is only *moveable*, so ownership can be transferred between instances, but only one instance refers to a particular asynchronous result at a time, `std::shared_future` instances are *copyable*, so you can have multiple objects referring to the same associated state.

Now, with `std::shared_future`, member functions on an individual object are still unsynchronized, so to avoid data races when accessing a single object from multiple threads, you must protect accesses with a lock. The preferred way to use it would be to take a copy of the object instead and have each thread access its own copy. Accesses to the shared asynchronous state from multiple threads are safe if each thread accesses that state through its own `std::shared_future` object. See figure 4.1.

One potential use of `std::shared_future` is for implementing parallel execution of something akin to a complex spreadsheet; each cell has a single final value, which may be used by the formulas in multiple other cells. The formulas for calculating the results of the dependent cells can then use a `std::shared_future` to reference the first cell. If all the formulas for the individual cells are then executed in parallel, those tasks that can proceed to completion will do so, whereas those that depend on others will block until their dependencies are ready. This will thus allow the system to make maximum use of the available hardware concurrency.

Instances of `std::shared_future` that reference some asynchronous state are constructed from instances of `std::future` that reference that state. Since `std::future` objects don't share ownership of the asynchronous state with any other object, the ownership must be transferred into the `std::shared_future` using `std::move`, leaving the `std::future` in an empty state, as if it was default constructed:

```
std::promise<int> p;
std::future<int> f(p.get_future());
assert(f.valid());
std::shared_future<int> sf(std::move(f));
assert(!f.valid());
assert(sf.valid());
```

❶ The future f is valid

❷ f is no longer valid

❸ sf is now valid

Here, the future `f` is initially valid ❶ because it refers to the asynchronous state of the promise `p`, but after transferring the state to `sf`, `f` is no longer valid ❷, whereas `sf` is ❸.

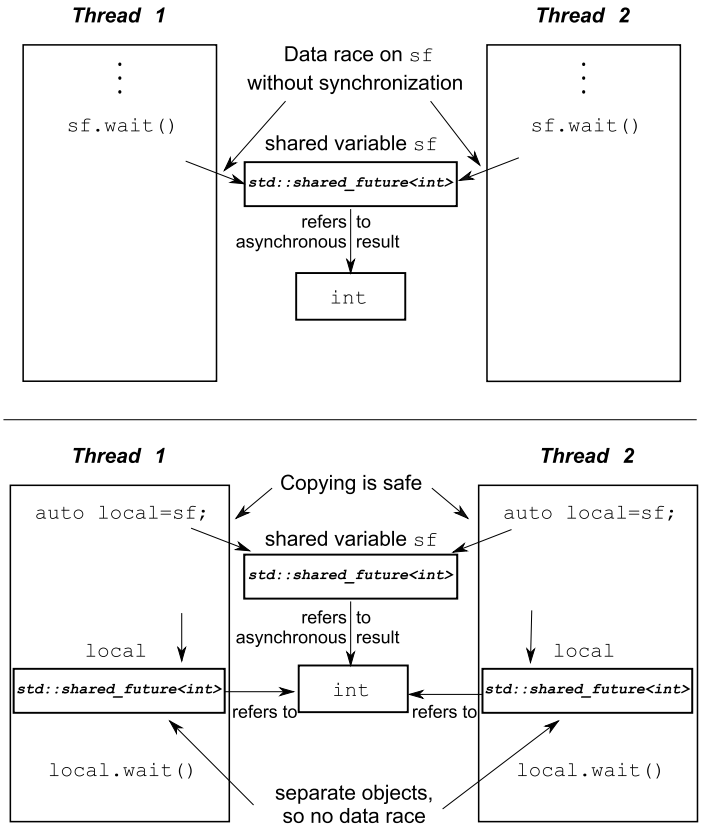

Figure 4.1 Using multiple `std::shared_future` objects to avoid data races

Just as with other movable objects, the transfer of ownership is implicit for rvalues, so you can construct a `std::shared_future` directly from the return value of the `get_future()` member function of a `std::promise` object, for example:

```
std::promise<std::string> p;
std::shared_future<std::string> sf(p.get_future());
```

❶ **Implicit transfer of ownership**

Here, the transfer of ownership is implicit; the `std::shared_future<>` is constructed from an rvalue of type `std::future<std::string>` ❶.

`std::future` also has an additional feature to facilitate the use of `std::shared_future` with the new facility for automatically deducing the type of a variable from its initializer (see appendix A, section A.6). `std::future` has a `share()` member function that creates a new `std::shared_future` and transfers ownership to it directly. This can save a lot of typing and makes code easier to change:

```
std::promise< std::map< SomeIndexType, SomeDataType, SomeComparator,
    SomeAllocator>::iterator> p;
auto sf=p.get_future().share();
```

In this case, the type of `sf` is deduced to be `std::shared_future< std::map< SomeIndexType, SomeDataType, SomeComparator, SomeAllocator>::iterator>`, which is rather a mouthful. If the comparator or allocator is changed, you only need to change the type of the promise; the type of the future is automatically updated to match.

Sometimes you want to limit the amount of time you're waiting for an event, either because you have a hard time limit on how long a particular section of code may take, or because there's other useful work that the thread can be doing if the event isn't going to happen soon. To handle this facility, many of the waiting functions have variants that allow a timeout to be specified.

4.3 Waiting with a time limit

All the blocking calls introduced previously will block for an indefinite period of time, suspending the thread until the event being waited for occurs. In many cases this is fine, but in some cases you want to put a limit on how long you wait. This might be to allow you to send some form of "I'm still alive" message either to an interactive user or another process or indeed to allow you to abort the wait if the user has given up waiting and pressed Cancel.

There are two sorts of timeouts you may wish to specify: a *duration-based* timeout, where you wait for a specific amount of time (for example, 30 milliseconds), or an *absolute* timeout, where you wait until a specific point in time (for example, 17:30:15.045987023 UTC on November 30, 2011). Most of the waiting functions provide variants that handle both forms of timeouts. The variants that handle the duration-based timeouts have a `_for` suffix, and those that handle the absolute timeouts have a `_until` suffix.

So, for example, `std::condition_variable` has two overloads of the `wait_for()` member function and two overloads of the `wait_until()` member function that correspond to the two overloads of `wait()`—one overload that just waits until signaled, or the timeout expires, or a spurious wakeup occurs, and another that will check the supplied predicate when woken and will return only when the supplied predicate is `true` (and the condition variable has been signaled) or the timeout expires.

Before we look at the details of the functions that use the timeouts, let's examine the way that times are specified in C++, starting with clocks.

4.3.1 Clocks

As far as the C++ Standard Library is concerned, a clock is a source of time information. In particular, a clock is a class that provides four distinct pieces of information:

- The time *now*
- The type of the value used to represent the times obtained from the clock
- The tick period of the clock
- Whether or not the clock ticks at a uniform rate and is thus considered to be a *steady* clock

The current time of a clock can be obtained by calling the static member function `now()` for that clock class; for example, `std::chrono::system_clock::now()` will return the current time of the system clock. The type of the time points for a particular clock is specified by the `time_point` member `typedef`, so the return type of `some_clock::now()` is `some_clock::time_point`.

The tick period of the clock is specified as a fractional number of seconds, which is given by the `period` member `typedef` of the clock—a clock that ticks 25 times per second thus has a `period` of `std::ratio<1,25>`, whereas a clock that ticks every 2.5 seconds has a `period` of `std::ratio<5,2>`. If the tick period of a clock can't be known until runtime, or it may vary during a given run of the application, the `period` may be specified as the average tick period, smallest possible tick period, or some other value that the library writer deems appropriate. There's no guarantee that the observed tick period in a given run of the program matches the specified `period` for that clock.

If a clock *ticks at a uniform rate* (whether or not that rate matches the `period`) and *can't be adjusted*, the clock is said to be a *steady* clock. The `is_steady` static data member of the clock class is `true` if the clock is steady and `false` otherwise. Typically, `std::chrono::system_clock` will *not* be steady, because the clock can be adjusted, even if such adjustment is done automatically to take account of local clock drift. Such an adjustment may cause a call to `now()` to return a value earlier than that returned by a prior call to `now()`, which is in violation of the requirement for a uniform tick rate. Steady clocks are important for timeout calculations, as you'll see shortly, so the C++ Standard Library provides one in the form of `std::chrono::steady_clock`. The other clocks provided by the C++ Standard Library are `std::chrono::system_clock` (mentioned above), which represents the "real time" clock of the system and which provides functions for converting its time points to and from `time_t` values, and `std::chrono::high_resolution_clock`, which provides the smallest possible tick period (and thus the highest possible resolution) of all the library-supplied clocks. It may actually be a `typedef` to one of the other clocks. These clocks are defined in the `<chrono>` library header, along with the other time facilities.

We'll look at the representation of time points shortly, but first let's look at how durations are represented.

4.3.2 Durations

Durations are the simplest part of the time support; they're handled by the `std::chrono::duration<>` class template (all the C++ time-handling facilities used by the Thread Library are in the `std::chrono` namespace). The first template parameter is the type of the representation (such as `int`, `long`, or `double`), and the second is a fraction specifying how many seconds each unit of the duration represents. For example, a number of minutes stored in a `short` is `std::chrono::duration<short,std::ratio<60,1>>`, because there are 60 seconds in a minute. On the other hand, a count of milliseconds stored in a `double` is `std::chrono::duration<double,std::ratio<1,1000>>`, because each millisecond is 1/1000 of a second.

The Standard Library provides a set of predefined `typedefs` in the `std::chrono` namespace for various durations: `nanoseconds`, `microseconds`, `milliseconds`, `seconds`, `minutes`, and `hours`. They all use a sufficiently large integral type for the representation chosen such that you can represent a duration of over 500 *years* in the appropriate units if you so desire. There are also `typedefs` for all the SI ratios from `std::atto`

(10^{-18}) to `std::exa` (10^{18}) (and beyond, if your platform has 128-bit integer types) for use when specifying custom durations such as `std::duration<double,std::centi>` for a count of 1/100 of a second represented in a `double`.

Conversion between durations is implicit where it does not require truncation of the value (so converting hours to seconds is OK, but converting seconds to hours is not). Explicit conversions can be done with `std::chrono::duration_cast<>`:

```
std::chrono::milliseconds ms(54802);
std::chrono::seconds s=
    std::chrono::duration_cast<std::chrono::seconds>(ms);
```

The result is truncated rather than rounded, so `s` will have a value of 54 in this example.

Durations support arithmetic, so you can add and subtract durations to get new durations or multiply or divide by a constant of the underlying representation type (the first template parameter). Thus `5*seconds(1)` is the same as `seconds(5)` or `minutes(1) - seconds(55)`. The count of the number of units in the duration can be obtained with the `count()` member function. Thus `std::chrono::milliseconds(1234).count()` is 1234.

Duration-based waits are done with instances of `std::chrono::duration<>`. For example, you can wait for up to 35 milliseconds for a future to be ready:

```
std::future<int> f=std::async(some_task);
if(f.wait_for(std::chrono::milliseconds(35))==std::future_status::ready)
    do_something_with(f.get());
```

The wait functions all return a status to indicate whether the wait timed out or the waited-for event occurred. In this case, you're waiting for a future, so the function returns `std::future_status::timeout` if the wait times out, `std::future_status::ready` if the future is ready, or `std::future_status::deferred` if the future's task is deferred. The time for a duration-based wait is measured using a steady clock internal to the library, so 35 milliseconds means 35 milliseconds of elapsed time, even if the system clock was adjusted (forward or back) during the wait. Of course, the vagaries of system scheduling and the varying precisions of OS clocks means that the actual time between the thread issuing the call and returning from it may be much longer than 35 ms.

With durations under our belt, we can now move on to time points.

4.3.3 Time points

The time point for a clock is represented by an instance of the `std::chrono::time_point<>` class template, which specifies which clock it refers to as the first template parameter and the units of measurement (a specialization of `std::chrono::duration<>`) as the second template parameter. The value of a time point is the length of time (in multiples of the specified duration) since a specific point in time called the *epoch* of the clock. The epoch of a clock is a basic property but not something that's directly available to query or specified by the C++ Standard. Typical epochs include 00:00 on January 1, 1970 and the instant when the computer running the application booted up. Clocks may share an epoch or have independent epochs. If two clocks share an epoch, the `time_point` `typedef` in one class may specify the other as the clock type associated with the

`time_point`. Although you can't find out when the epoch is, you *can* get the `time_since_epoch()` for a given `time_point`. This member function returns a duration value specifying the length of time since the clock epoch to that particular time point.

For example, you might specify a time point as `std::chrono::time_point<std::chrono::system_clock, std::chrono::minutes>`. This would hold the time relative to the system clock but measured in minutes as opposed to the native precision of the system clock (which is typically seconds or less).

You can add durations and subtract durations from instances of `std::chrono::time_point<>` to produce new time points, so `std::chrono::high_resolution_clock::now() + std::chrono::nanoseconds(500)` will give you a time 500 nanoseconds in the future. This is good for calculating an absolute timeout when you know the maximum duration of a block of code, but there are multiple calls to waiting functions within it or nonwaiting functions that precede a waiting function but take up some of the time budget.

You can also subtract one time point from another that shares the same clock. The result is a duration specifying the length of time between the two time points. This is useful for timing blocks of code, for example:

```
auto start=std::chrono::high_resolution_clock::now();
do_something();
auto stop=std::chrono::high_resolution_clock::now();
std::cout<<"do_something() took "
  <<std::chrono::duration<double,std::chrono::seconds>(stop-start).count()
  <<" seconds"<<std::endl;
```

The clock parameter of a `std::chrono::time_point<>` instance does more than just specify the epoch, though. When you pass the time point to a wait function that takes an absolute timeout, the clock parameter of the time point is used to measure the time. This has important consequences when the clock is changed, because the wait tracks the clock change and won't return until the clock's `now()` function returns a value later than the specified timeout. If the clock is adjusted forward, this may reduce the total length of the wait (as measured by a steady clock), and if it's adjusted backward, this may increase the total length of the wait.

As you may expect, time points are used with the `_until` variants of the wait functions. The typical use case is as an offset from *some-clock*`::now()` at a fixed point in the program, although time points associated with the system clock can be obtained by converting from a `time_t` using the `std::chrono::system_clock::to_time_point()` static member function for scheduling operations at a user-visible time. For example, if you have a maximum of 500 milliseconds to wait for an event associated with a condition variable, you might do something like in the following listing.

Listing 4.11 Waiting for a condition variable with a timeout

```
#include <condition_variable>
#include <mutex>
#include <chrono>
```

```
std::condition_variable cv;
bool done;
std::mutex m;

bool wait_loop()
{
    auto const timeout= std::chrono::steady_clock::now()+
        std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done)
    {
        if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
            break;
    }
    return done;
}
```

This is the recommended way to wait for condition variables with a time limit, if you're not passing a predicate to the wait. This way, the overall length of the loop is bounded. As you saw in section 4.1.1, you need to loop when using condition variables if you don't pass in the predicate, in order to handle spurious wakeups. If you use `wait_for()` in a loop, you might end up waiting almost the full length of time before a spurious wake, and the next time through the wait time starts again. This may repeat any number of times, making the total wait time unbounded.

With the basics of specifying timeouts under your belt, let's look at the functions that you can use the timeout with.

4.3.4 Functions that accept timeouts

The simplest use for a timeout is to add a delay to the processing of a particular thread, so that it doesn't take processing time away from other threads when it has nothing to do. You saw an example of this in section 4.1, where you polled a "done" flag in a loop. The two functions that handle this are `std::this_thread::sleep_for()` and `std::this_thread::sleep_until()`. They work like a basic alarm clock: the thread goes to sleep either for the specified duration (with `sleep_for()`) or until the specified point in time (with `sleep_until()`). `sleep_for()` makes sense for examples like that from section 4.1, where something must be done periodically, and the elapsed time is what matters. On the other hand, `sleep_until()` allows you to schedule the thread to wake at a particular point in time. This could be used to trigger the backups at midnight, or the payroll print run at 6:00 a.m., or to suspend the thread until the next frame refresh when doing a video playback.

Of course, sleeping isn't the only facility that takes a timeout; you already saw that you can use timeouts with condition variables and futures. You can even use timeouts when trying to acquire a lock on a mutex if the mutex supports it. Plain `std::mutex` and `std::recursive_mutex` don't support timeouts on locking, but `std::timed_mutex` does, as does `std::recursive_timed_mutex`. Both these types support `try_lock_for()` and `try_lock_until()` member functions that try to obtain the lock within a specified time period or before a specified time point. Table 4.1 shows

the functions from the C++ Standard Library that can accept timeouts, their parameters, and their return values. Parameters listed as *duration* must be an instance of `std::duration<>`, and those listed as *time_point* must be an instance of `std::time_point<>`.

Table 4.1 Functions that accept timeouts

Class/Namespace	Functions	Return values
`std::this_thread` namespace	`sleep_for(`*duration*`)` `sleep_until (`*time_point*`)`	N/A
`std::condition_variable` or `std::condition_variable_any`	`wait_for(`*lock*`, `*duration*`)` `wait_until(`*lock*`, `*time_point*`)`	`std::cv_status::timeout` or `std::cv_status::no_timeout`
	`wait_for(`*lock*`, `*duration*`, `*predicate*`)` `wait_until(`*lock*`, `*time_point*`, `*predicate*`)`	`bool`—the return value of the *predicate* when awakened
`std::timed_mutex` or `std::recursive_timed_mutex`	`try_lock_for (`*duration*`)` `try_lock_until (`*time_point*`)`	`bool`—`true` if the lock was acquired, `false` otherwise
`std::unique_lock<`*TimedLockable*`>`	`unique_lock(`*lockable*`, `*duration*`)` `unique_lock(`*lockable*`, `*time_point*`)`	N/A—`owns_lock()` on the newly constructed object; returns `true` if the lock was acquired, `false` otherwise
	`try_lock_for(`*duration*`)` `try_lock_until (time_point)`	`bool`—`true` if the lock was acquired, `false` otherwise
`std::future<`*ValueType*`>` or `std::shared_future<`*ValueType*`>`	`wait_for(`*duration*`)` `wait_until (`*time_point*`)`	`std::future_status::timeout` if the wait timed out, `std::future_status::ready` if the future is ready, or `std::future_status::deferred` if the future holds a deferred function that hasn't yet started

Now that I've covered the mechanics of condition variables, futures, promises, and packaged tasks, it's time to look at the wider picture and how they can be used to simplify the synchronization of operations between threads.

4.4 Using synchronization of operations to simplify code

Using the synchronization facilities described so far in this chapter as building blocks allows you to focus on the operations that need synchronizing rather than the mechanics. One way this can help simplify your code is that it accommodates a much more *functional* (in the sense of *functional programming*) approach to programming concurrency. Rather than sharing data directly between threads, each task can be provided with the data it needs, and the result can be disseminated to any other threads that need it through the use of futures.

4.4.1 Functional programming with futures

The term *functional programming* (FP) refers to a style of programming where the result of a function call depends solely on the parameters to that function and doesn't depend on any external state. This is related to the mathematical concept of a function, and it means that if you invoke a function twice with the same parameters, the result is exactly the same. This is a property of many of the mathematical functions in the C++ Standard Library, such as `sin`, `cos`, and `sqrt`, and simple operations on basic types, such as `3+3`, `6*9`, or `1.3/4.7`. A *pure* function doesn't *modify* any external state either; the effects of the function are entirely limited to the return value.

This makes things easy to think about, especially when concurrency is involved, because many of the problems associated with shared memory discussed in chapter 3 disappear. If there are no modifications to shared data, there can be no race conditions and thus no need to protect shared data with mutexes either. This is such a powerful simplification that programming languages such as Haskell,[2] where all functions are pure *by default*, are becoming increasingly popular for programming concurrent systems. Because most things are pure, the *impure* functions that actually *do* modify the shared state stand out all the more, and it's therefore easier to reason about how they fit into the overall structure of the application.

The benefits of functional programming aren't limited to those languages where it's the default paradigm, however. C++ is a multiparadigm language, and it's entirely possible to write programs in the FP style. This is even easier in C++11 than it was in C++98, with the advent of lambda functions (see appendix A, section A.6), the incorporation of `std::bind` from Boost and TR1, and the introduction of automatic type deduction for variables (see appendix A, section A.7). Futures are the final piece of the puzzle that makes FP-style concurrency viable in C++; a future can be passed around between threads to allow the result of one computation to depend on the result of another, *without any explicit access to shared data.*

FP-style Quicksort

To illustrate the use of futures for FP-style concurrency, let's look at a simple implementation of the Quicksort algorithm. The basic idea of the algorithm is simple: given a list of values, take an element to be the pivot element, and then partition the

[2] See http://www.haskell.org/.

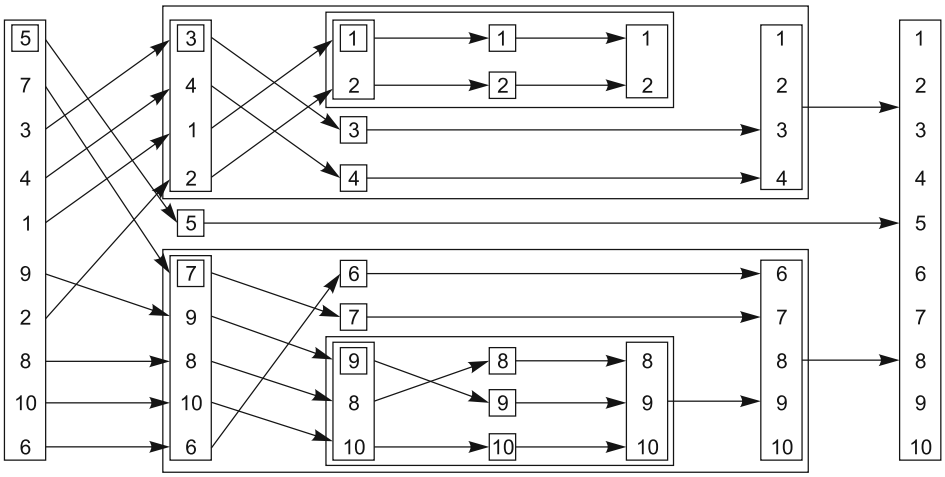

Figure 4.2 FP-style recursive sorting

list into two sets—those less than the pivot and those greater than or equal to the pivot. A sorted copy of the list is obtained by sorting the two sets and returning the sorted list of values less than the pivot, followed by the pivot, followed by the sorted list of values greater than or equal to the pivot. Figure 4.2 shows how a list of 10 integers is sorted under this scheme. An FP-style sequential implementation is shown in the following listing; it takes and returns a list by value rather than sorting in place like `std::sort()` does.

Listing 4.12 A sequential implementation of Quicksort

```
template<typename T>
std::list<T> sequential_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());          ❶
    T const& pivot=*result.begin();                             ❷

    auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});                  ❸

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);                                          ❹

    auto new_lower(
        sequential_quick_sort(std::move(lower_part)));          ❺
    auto new_higher(
        sequential_quick_sort(std::move(input)));               ❻

    result.splice(result.end(),new_higher);                     ❼
```

```
    result.splice(result.begin(),new_lower);          <-- 8
    return result;
}
```

Although the interface is FP-style, if you used FP-style throughout you'd do a lot of copying, so you use "normal" imperative style for the internals. You take the first element as the pivot by slicing it off the front of the list using `splice()` ❶. Although this can potentially result in a suboptimal sort (in terms of numbers of comparisons and exchanges), doing anything else with a `std::list` can add quite a bit of time because of the list traversal. You know you're going to want it in the result, so you can splice it directly into the list you'll be using for that. Now, you're also going to want to use it for comparisons, so let's take a reference to it to avoid copying ❷. You can then use `std::partition` to divide the sequence into those values *less than* the pivot and those *not* less than the pivot ❸. The easiest way to specify the partition criteria is to use a lambda function; you use a reference capture to avoid copying the `pivot` value (see appendix A, section A.5 for more on lambda functions).

`std::partition()` rearranges the list in place and returns an iterator marking the first element that's *not* less than the pivot value. The full type for an iterator can be quite long-winded, so you just use the `auto` type specifier to force the compiler to work it out for you (see appendix A, section A.7).

Now, you've opted for an FP-style interface, so if you're going to use recursion to sort the two "halves," you'll need to create two lists. You can do this by using `splice()` again to move the values from `input` up to the `divide_point` into a new list: `lower_part` ❹. This leaves the remaining values alone in `input`. You can then sort the two lists with recursive calls ❺, ❻. By using `std::move()` to pass the lists in, you can avoid copying here too—the result is implicitly moved out anyway. Finally, you can use `splice()` yet again to piece the `result` together in the right order. The `new_higher` values go on the end ❼, after the pivot, and the `new_lower` values go at the beginning, before the pivot ❽.

FP-STYLE PARALLEL QUICKSORT

Because this uses a functional style already, it's now easy to convert this to a parallel version using futures, as shown in the next listing. The set of operations is the same as before, except that some of them now run in parallel. This version uses an implementation of the Quicksort algorithm using futures and a functional style.

Listing 4.13 Parallel Quicksort using futures

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());
    T const& pivot=*result.begin();
```

```
    auto divide_point=std::partition(input.begin(),input.end(),
            [&](T const& t){return t<pivot;});

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);

    std::future<std::list<T> > new_lower(                    <-- 1
        std::async(&parallel_quick_sort<T>,std::move(lower_part)));

    auto new_higher(
        parallel_quick_sort(std::move(input)));    <-- 2

    result.splice(result.end(),new_higher);        <-- 3
    result.splice(result.begin(),new_lower.get());       <-- 4
    return result;
}
```

The big change here is that rather than sorting the lower portion on the current thread, you sort it on another thread using `std::async()` ❶. The upper portion of the list is sorted with direct recursion as before ❷. By recursively calling `parallel_quick_sort()`, you can take advantage of the available hardware concurrency. If `std::async()` starts a new thread every time, then if you recurse down three times, you'll have eight threads running; if you recurse down 10 times (for ~1000 elements), you'll have 1024 threads running if the hardware can handle it. If the library decides there are too many spawned tasks (perhaps because the number of tasks has exceeded the available hardware concurrency), it may switch to spawning the new tasks synchronously. They will run in the thread that calls `get()` rather than on a new thread, thus avoiding the overhead of passing the task to another thread when this won't help the performance. It's worth noting that it's perfectly conforming for an implementation of `std::async` to start a new thread for each task (even in the face of massive oversubscription) unless `std::launch::deferred` is explicitly specified or to run all tasks synchronously unless `std::launch::async` is explicitly specified. If you're relying on the library for automatic scaling, you're advised to check the documentation for your implementation to see what behavior it exhibits.

Rather than using `std::async()`, you could write your own `spawn_task()` function as a simple wrapper around `std::packaged_task` and `std::thread`, as shown in listing 4.14; you'd create a `std::packaged_task` for the result of the function call, get the future from it, run it on a thread, and return the future. This wouldn't itself offer much advantage (and indeed would likely lead to massive oversubcription), but it would pave the way to migrate to a more sophisticated implementation that adds the task to a queue to be run by a pool of worker threads. We'll look at thread pools in chapter 9. It's probably worth going this way in preference to using `std::async` only if you really know what you're doing and want complete control over the way the thread pool is built and executes tasks.

Anyway, back to `parallel_quick_sort`. Because you just used direct recursion to get `new_higher`, you can just splice it into place as before ❸. But `new_lower` is now a `std::future<std::list<T>>` rather than just a list, so you need to call `get()` to

retrieve the value before you can call `splice()` ❹. This then waits for the background task to complete and *moves* the result into the `splice()` call; `get()` returns an rvalue reference to the contained result, so it can be moved out (see appendix A, section A.1.1 for more on rvalue references and move semantics).

Even assuming that `std::async()` makes optimal use of the available hardware concurrency, this still isn't an ideal parallel implementation of Quicksort. For one thing, `std::partition` does a lot of the work, and that's still a sequential call, but it's good enough for now. If you're interested in the fastest possible parallel implementation, check the academic literature.

Listing 4.14 A sample implementation of `spawn_task`

```
template<typename F,typename A>
std::future<std::result_of<F(A&&)>::type>
    spawn_task(F&& f,A&& a)
{
    typedef std::result_of<F(A&&)>::type result_type;
    std::packaged_task<result_type(A&&)>
        task(std::move(f)));
    std::future<result_type> res(task.get_future());
    std::thread t(std::move(task),std::move(a));
    t.detach();
    return res;
}
```

Functional programming isn't the only concurrent programming paradigm that eschews shared mutable data; another paradigm is CSP (Communicating Sequential Processes),[3] where threads are conceptually entirely separate, with no shared data but with communication channels that allow messages to be passed between them. This is the paradigm adopted by the programming language Erlang (http://www.erlang.org/) and by the MPI (Message Passing Interface) (http://www.mpi-forum.org/) environment commonly used for high-performance computing in C and C++. I'm sure that by now you'll be unsurprised to learn that this can also be supported in C++ with a bit of discipline; the following section discusses one way to achieve this.

4.4.2 Synchronizing operations with message passing

The idea of CSP is simple: if there's no shared data, each thread can be reasoned about entirely independently, purely on the basis of how it behaves in response to the messages that it received. Each thread is therefore effectively a state machine: when it receives a message, it updates its state in some manner and maybe sends one or more messages to other threads, with the processing performed depending on the initial state. One way to write such threads would be to formalize this and implement a Finite State Machine model, but this isn't the only way; the state machine can be implicit in the structure of the application. Which method works better in any given scenario

[3] *Communicating Sequential Processes*, C.A.R. Hoare, Prentice Hall, 1985. Available free online at http://www.usingcsp.com/cspbook.pdf.

depends on the exact behavioral requirements of the situation and the expertise of the programming team. However you choose to implement each thread, the separation into independent processes has the potential to remove much of the complication from shared-data concurrency and therefore make programming easier, lowering the bug rate.

True communicating sequential processes have no shared data, with all communication passed through the message queues, but because C++ threads share an address space, it's not possible to enforce this requirement. This is where the discipline comes in: as application or library authors, it's our responsibility to ensure that we don't share data between the threads. Of course, the message queues must be shared in order for the threads to communicate, but the details can be wrapped in the library.

Imagine for a moment that you're implementing the code for an ATM. This code needs to handle interaction with the person trying to withdraw money and interaction with the relevant bank, as well as control the physical machinery to accept the person's card, display appropriate messages, handle key presses, issue money, and return their card.

One way to handle everything would be to split the code into three independent threads: one to handle the physical machinery, one to handle the ATM logic, and one to communicate with the bank. These threads could communicate purely by passing messages rather than sharing any data. For example, the thread handling the machinery would send a message to the logic thread when the person at the machine entered their card or pressed a button, and the logic thread would send a message to the machinery thread indicating how much money to dispense, and so forth.

One way to model the ATM logic would be as a state machine. In each state the thread waits for an acceptable message, which it then processes. This may result in transitioning to a new state, and the cycle continues. The states involved in a simple implementation are shown in figure 4.3. In this simplified implementation, the system waits for a card to be inserted. Once the card is inserted, it then waits for the user to enter their PIN, one digit at a time. They can delete the last digit entered. Once enough digits have been entered, the PIN is verified. If the PIN is not OK, you're finished, so you return the card to the customer and resume waiting for someone to enter their card. If the PIN is OK, you wait for them to either cancel the transaction or select an amount to withdraw. If they cancel, you're finished, and you return their card. If they select an amount, you wait for confirmation from the bank before issuing the cash and returning the card or displaying an "insufficient funds" message and returning their card. Obviously, a real ATM is considerably more complex, but this is enough to illustrate the idea.

Having designed a state machine for your ATM logic, you can implement it with a class that has a member function to represent each state. Each member function can then wait for specific sets of incoming messages and handle them when they arrive, possibly triggering a switch to another state. Each distinct message type is represented by a separate `struct`. Listing 4.15 shows part of a simple implementation of the ATM logic in such a system, with the main loop and the implementation of the first state, waiting for the card to be inserted.

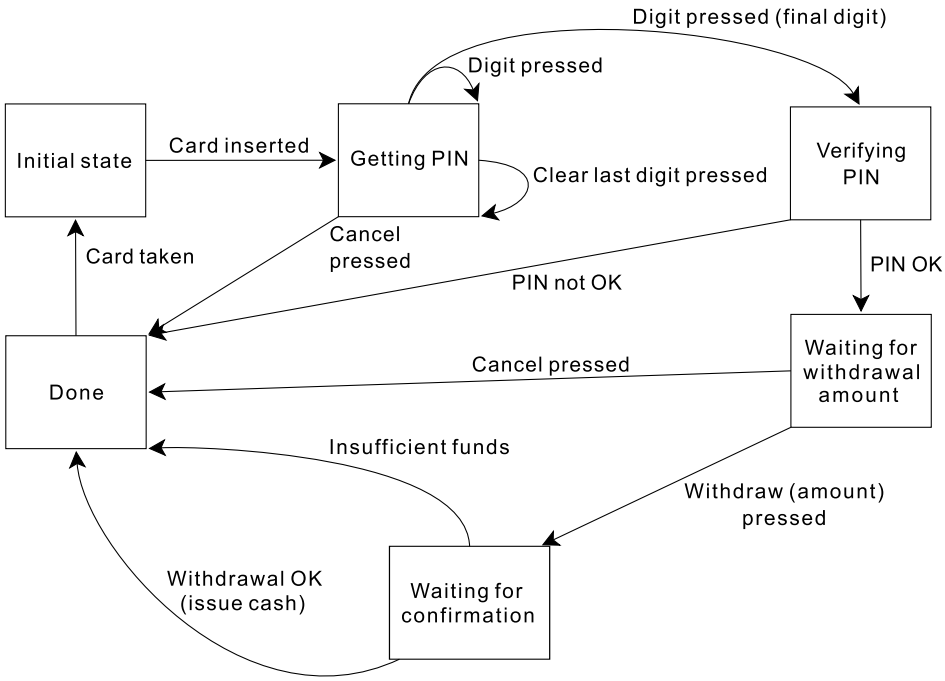

Figure 4.3 A simple state machine model for an ATM

As you can see, all the necessary synchronization for the message passing is entirely hidden inside the message-passing library (a basic implementation of which is given in appendix C, along with the full code for this example).

Listing 4.15 A simple implementation of an ATM logic class

```
struct card_inserted
{
    std::string account;
};
class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface_hardware;
    void (atm::*state)();

    std::string account;
    std::string pin;

    void waiting_for_card()                                    <-(1)
    {
        interface_hardware.send(display_enter_card());         <-(2)
        incoming.wait()                                        <-(3)
            .handle<card_inserted>(
                [&](card_inserted const& msg)                  <-(4)
                {
                    account=msg.account;
                    pin="";
```

```
                    interface_hardware.send(display_enter_pin());
                    state=&atm::getting_pin;
                }
                );
    }
    void getting_pin();
public:
    void run()          <-- 5
    {
        state=&atm::waiting_for_card;    <-- 6
        try
        {
            for(;;)
            {
                (this->*state)();    <-- 7
            }
        }
        catch(messaging::close_queue const&)
        {
        }
    }
};
```

As already mentioned, the implementation described here is grossly simplified from the real logic that would be required in an ATM, but it does give you a feel for the message-passing style of programming. There's no need to think about synchronization and concurrency issues, just which messages may be received at any given point and which messages to send. The state machine for this ATM logic runs on a single thread, with other parts of the system such as the interface to the bank and the terminal interface running on separate threads. This style of program design is called the *Actor model*—there are several discrete *actors* in the system (each running on a separate thread), which send messages to each other to perform the task at hand, and there's no shared state except that directly passed via messages.

Execution starts with the `run()` member function ❺, which sets the initial state to `waiting_for_card` ❻ and then repeatedly executes the member function representing the current state (whatever it is) ❼. The state functions are simple member functions of the `atm` class. The `waiting_for_card` state function ❶ is also simple: it sends a message to the interface to display a "waiting for card" message ❷ and then waits for a message to handle ❸. The only type of message that can be handled here is a `card_inserted` message, which you handle with a lambda function ❹. You could pass any function or function object to the `handle` function, but for a simple case like this, it's easiest to use a lambda. Note that the `handle()` function call is chained onto the `wait()` function; if a message is received that doesn't match the specified type, it's discarded, and the thread continues to wait until a matching message is received.

The lambda function itself just caches the account number from the card in a member variable, clears the current PIN, sends a message to the interface hardware to display something asking the user to enter their PIN, and changes to the "getting PIN"

state. Once the message handler has completed, the state function returns, and the main loop then calls the new state function ❼.

The `getting_pin` state function is a bit more complex in that it can handle three distinct types of message, as in figure 4.3. This is shown in the following listing.

Listing 4.16 The `getting_pin` state function for the simple ATM implementation

```
void atm::getting_pin()
{
    incoming.wait()
        .handle<digit_pressed>(                    ❶
            [&](digit_pressed const& msg)
            {
                unsigned const pin_length=4;
                pin+=msg.digit;
                if(pin.length()==pin_length)
                {
                    bank.send(verify_pin(account,pin,incoming));
                    state=&atm::verifying_pin;
                }
            }
            )
        .handle<clear_last_pressed>(               ❷
            [&](clear_last_pressed const& msg)
            {
                if(!pin.empty())
                {
                    pin.resize(pin.length()-1);
                }
            }
            )
        .handle<cancel_pressed>(                   ❸
            [&](cancel_pressed const& msg)
            {
                state=&atm::done_processing;
            }
            );
}
```

This time, there are three message types you can process, so the `wait()` function has three `handle()` calls chained on the end ❶, ❷, ❸. Each call to `handle()` specifies the message type as the template parameter and then passes in a lambda function that takes that particular message type as a parameter. Because the calls are chained together in this way, the `wait()` implementation knows that it's waiting for a `digit_pressed` message, a `clear_last_pressed` message, or a `cancel_pressed` message. Messages of any other type are again discarded.

This time, you don't necessarily change state when you get a message. For example, if you get a `digit_pressed` message, you just add it to the `pin` unless it's the final digit. The main loop ❼ in listing 4.15) will then call `getting_pin()` again to wait for the next digit (or clear or cancel).

This corresponds to the behavior shown in figure 4.3. Each state box is implemented by a distinct member function, which waits for the relevant messages and updates the state as appropriate.

As you can see, this style of programming can greatly simplify the task of designing a concurrent system, because each thread can be treated entirely independently. It is thus an example of using multiple threads to separate concerns and as such requires you to explicitly decide how to divide the tasks between threads.

4.5 Summary

Synchronizing operations between threads is an important part of writing an application that uses concurrency: if there's no synchronization, the threads are essentially independent and might as well be written as separate applications that are run as a group because of their related activities. In this chapter, I've covered various ways of synchronizing operations from the basic condition variables, through futures, promises, and packaged tasks. I've also discussed ways of approaching the synchronization issues: functional-style programming where each task produces a result entirely dependent on its input rather than on the external environment, and message passing where communication between threads is via asynchronous messages sent through a messaging subsystem that acts as an intermediary.

Having discussed many of the high-level facilities available in C++, it's now time to look at the low-level facilities that make it all work: the C++ memory model and atomic operations.