```
    {}
    ~thread_guard()
    {
        if(t.joinable())
        {
            t.join();
        }
    }
    thread_guard(thread_guard const&)=delete;
    thread_guard& operator=(thread_guard const&)=delete;
};
struct func;
void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);
    do_something_in_current_thread();
}
```

See definition in listing 2.1

When the execution of the current thread reaches the end of `f`, the local objects are destroyed in reverse order of construction. Consequently, the `thread_guard` object, `g`, is destroyed first, and the thread is joined with, in the destructor. This even happens if the function exits because `do_something_in_current_thread` throws an exception.

The destructor of `thread_guard` in listing 2.3 first tests to see if the `std::thread` object is `joinable()` before calling `join()`. This is important, because `join()` can be called only once for a given thread of execution, so it would be a mistake to do so if the thread had already been joined.

The copy constructor and copy-assignment operators are marked `=delete` to ensure that they're not automatically provided by the compiler. Copying or assigning such an object would be dangerous, because it might then outlive the scope of the thread it was joining. By declaring them as deleted, any attempt to copy a `thread_guard` object will generate a compilation error. See appendix A, section A.2, for more about deleted functions.

If you don't need to wait for a thread to finish, you can avoid this exception-safety issue by *detaching* it. This breaks the association of the thread with the `std::thread` object and ensures that `std::terminate()` won't be called when the `std::thread` object is destroyed, even though the thread is still running in the background.

2.1.4 Running threads in the background

Calling `detach()` on a `std::thread` object leaves the thread to run in the background, with no direct means of communicating with it. It's no longer possible to wait for that thread to complete; if a thread becomes detached, it isn't possible to obtain a `std::thread` object that references it, so it can no longer be joined. Detached threads truly run in the background; ownership and control are passed over to the

C++ Runtime Library, which ensures that the resources associated with the thread are correctly reclaimed when the thread exits.

Detached threads are often called *daemon threads* after the UNIX concept of a *daemon process* that runs in the background without any explicit user interface. Such threads are typically long-running; they run for almost the entire lifetime of the application, performing a background task such as monitoring the filesystem, clearing unused entries out of object caches, or optimizing data structures. At the other extreme, it may make sense to use a detached thread where there's another mechanism for identifying when the thread has completed or where the thread is used for a fire-and-forget task.

As you've saw in section 2.1.2, you detach a thread by calling the `detach()` member function of the `std::thread` object. After the call completes, the `std::thread` object is no longer associated with the actual thread of execution and is therefore no longer joinable:

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
```

In order to detach the thread from a `std::thread` object, there must be a thread to detach: you can't call `detach()` on a `std::thread` object with no associated thread of execution. This is exactly the same requirement as for `join()`, and you can check it in exactly the same way—you can only call `t.detach()` for a `std::thread` object `t` when `t.joinable()` returns `true`.

Consider an application such as a word processor that can edit multiple documents at once. There are many ways to handle this, both at the UI level and internally. One way that's increasingly common at the moment is to have multiple, independent, top-level windows, one for each document being edited. Although these windows appear to be completely independent, each with its own menus, they're running within the same instance of the application. One way to handle this internally is to run each document-editing window in its own thread; each thread runs the same code but with different data relating to the document being edited and the corresponding window properties. Opening a new document therefore requires starting a new thread. The thread handling the request isn't going to care about waiting for that other thread to finish, because it's working on an unrelated document, so this makes it a prime candidate for running a detached thread.

The following listing shows a simple code outline for this approach.

Listing 2.4 Detaching a thread to handle other documents

```
void edit_document(std::string const& filename)
{
    open_document_and_display_gui(filename);
    while(!done_editing())
    {
        user_command cmd=get_user_input();
```

```
        if(cmd.type==open_new_document)
        {
            std::string const new_name=get_filename_from_user();
            std::thread t(edit_document,new_name);
            t.detach();
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
```

If the user chooses to open a new document, you prompt them for the document to open, start a new thread to open that document, and then detach it. Because the new thread is doing the same operation as the current thread but on a different file, you can reuse the same function (`edit_document`) with the newly chosen filename as the supplied argument.

This example also shows a case where it's helpful to pass arguments to the function used to start a thread: rather than just passing the name of the function to the `std::thread` constructor, you also pass in the filename parameter. Although other mechanisms could be used to do this, such as using a function object with member data instead of an ordinary function with parameters, the C++ Standard Library provides you with an easy way of doing it.

2.2 Passing arguments to a thread function

As shown in listing 2.4, passing arguments to the callable object or function is fundamentally as simple as passing additional arguments to the `std::thread` constructor. But it's important to bear in mind that by default, the arguments are copied into internal storage, where they can be accessed by the newly created thread of execution, and then passed to the callable object or function as rvalues as if they were temporaries. This is done even if the corresponding parameter in the function is expecting a reference. Here's an example:

```
void f(int i,std::string const& s);
std::thread t(f,3,"hello");
```

This creates a new thread of execution associated with `t`, which calls `f(3,"hello")`. Note that even though `f` takes a `std::string` as the second parameter, the string literal is passed as a `char const*` and converted to a `std::string` only in the context of the new thread. This is particularly important when the argument supplied is a pointer to an automatic variable, as follows:

```
void f(int i,std::string const& s);
void oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer, "%i",some_param);
```

```
    std::thread t(f,3,buffer);
    t.detach();
}
```

In this case, it's the pointer to the local variable `buffer` that's passed through to the new thread and there's a significant chance that the `oops` function will exit before the buffer has been converted to a `std::string` on the new thread, thus leading to undefined behavior. The solution is to cast to `std::string` before passing the buffer to the `std::thread` constructor:

```
void f(int i,std::string const& s);
void not_oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,std::string(buffer));
    t.detach();
}
```

Using std::string avoids dangling pointer

In this case, the problem is that you were relying on the implicit conversion of the pointer to the buffer into the `std::string` object expected as a function parameter, but this conversion happens too late because the `std::thread` constructor copies the supplied values as is, without converting to the expected argument type.

It's not possible to get the reverse scenario: the object is copied, and you wanted a non-`const` reference, because this won't compile. You might try this if the thread is updating a data structure that's passed in by reference; for example:

```
void update_data_for_widget(widget_id w,widget_data& data);
void oops_again(widget_id w)
{
    widget_data data;
    std::thread t(update_data_for_widget,w,data);
    display_status();
    t.join();
    process_widget_data(data);
}
```

Although `update_data_for_widget` expects the second parameter to be passed by reference, the `std::thread` constructor doesn't know that; it's oblivious to the types of the arguments expected by the function and blindly copies the supplied values. But the internal code passes copied arguments as rvalues in order to work with move-only types, and will thus try to call `update_data_for_widget` with an rvalue. This will fail to compile because you can't pass an rvalue to a function that expects a non-`const` reference. For those of you familiar with `std::bind`, the solution will be readily apparent: you need to wrap the arguments that need to be references in `std::ref`. In this case, if you change the thread invocation to

```
std::thread t(update_data_for_widget,w,std::ref(data));
```

then `update_data_for_widget` will be correctly passed a reference to `data` rather than a temporary copy of `data`, and the code will now compile successfully.

If you're familiar with `std::bind`, the parameter-passing semantics will be unsurprising, because both the operation of the `std::thread` constructor and the operation of `std::bind` are defined in terms of the same mechanism. This means that, for example, you can pass a member function pointer as the function, provided you supply a suitable object pointer as the first argument:

```
class X
{
public:
    void do_lengthy_work();
};
X my_x;
std::thread t(&X::do_lengthy_work,&my_x);
```

This code will invoke `my_x.do_lengthy_work()` on the new thread, because the address of `my_x` is supplied as the object pointer. You can also supply arguments to such a member function call: the third argument to the `std::thread` constructor will be the first argument to the member function, and so forth.

Another interesting scenario for supplying arguments is where the arguments can't be copied but can only be *moved*: the data held within one object is transferred over to another, leaving the original object empty. An example of such a type is `std::unique_ptr`, which provides automatic memory management for dynamically allocated objects. Only one `std::unique_ptr` instance can point to a given object at a time, and when that instance is destroyed, the pointed-to object is deleted. The *move constructor* and *move assignment operator* allow the ownership of an object to be transferred around between `std::unique_ptr` instances (see appendix A, section A.1.1, for more on move semantics). Such a transfer leaves the source object with a `NULL` pointer. This moving of values allows objects of this type to be accepted as function parameters or returned from functions. Where the source object is temporary, the move is automatic, but where the source is a named value, the transfer must be requested directly by invoking `std::move()`. The following example shows the use of `std::move` to transfer ownership of a dynamic object into a thread:

```
void process_big_object(std::unique_ptr<big_object>);
std::unique_ptr<big_object> p(new big_object);
p->prepare_data(42);
std::thread t(process_big_object,std::move(p));
```

By specifying `std::move(p)` in the `std::thread` constructor, the ownership of `big_object` is transferred first into internal storage for the newly created thread and then into `process_big_object`.

Several of the classes in the C++ Standard Library exhibit the same ownership semantics as `std::unique_ptr`, and `std::thread` is one of them. Though `std::thread` instances don't own a dynamic object in the same way as `std::unique_ptr` does, they do

own a resource: each instance is responsible for managing a thread of execution. This ownership can be transferred between instances, because instances of `std::thread` are movable, even though they aren't copyable. This ensures that only one object is associated with a particular thread of execution at any one time while allowing programmers the option of transferring that ownership between objects.

2.3 Transferring ownership of a thread

Suppose you want to write a function that creates a thread to run in the background, but passes ownership of the new thread back to the calling function rather than waiting for it to complete; or maybe you want to do the reverse: create a thread and pass ownership in to some function that should wait for it to complete. In either case, you need to transfer ownership from one place to another.

This is where the move support of `std::thread` comes in. As described in the previous section, many resource-owning types in the C++ Standard Library, such as `std::ifstream` and `std::unique_ptr`, are movable but not copyable, and `std::thread` is one of them. This means that the ownership of a particular thread of execution can be moved between `std::thread` instances, as in the following example. The example shows the creation of two threads of execution and the transfer of ownership of those threads among three `std::thread` instances, `t1`, `t2`, and `t3`:

```
void some_function();
void some_other_function();
std::thread t1(some_function);
std::thread t2=std::move(t1);
t1=std::thread(some_other_function);
std::thread t3;
t3=std::move(t2);
t1=std::move(t3);
```

This assignment will terminate the program!

First, a new thread is started and associated with `t1`. Ownership is then transferred over to `t2` when `t2` is constructed, by invoking `std::move()` to explicitly move ownership. At this point, `t1` no longer has an associated thread of execution; the thread running `some_function` is now associated with `t2`.

Then, a new thread is started and associated with a temporary `std::thread` object. The subsequent transfer of ownership into `t1` doesn't require a call to `std::move()` to explicitly move ownership, because the owner is a temporary object—moving from temporaries is automatic and implicit.

`t3` is default-constructed, which means that it's created without any associated thread of execution. Ownership of the thread currently associated with `t2` is transferred into `t3`, again with an explicit call to `std::move()`, because `t2` is a named object. After all these moves, `t1` is associated with the thread running `some_other_function`, `t2` has no associated thread, and `t3` is associated with the thread running `some_function`.

The final move transfers ownership of the thread running `some_function` back to `t1` where it started. But in this case `t1` already had an associated thread (which was running `some_other_function`), so `std::terminate()` is called to terminate the program.

This is done for consistency with the `std::thread` destructor. You saw in section 2.1.1 that you must explicitly wait for a thread to complete or detach it before destruction, and the same applies to assignment: you can't just drop a thread by assigning a new value to the `std::thread` object that manages it.

The move support in `std::thread` means that ownership can readily be transferred out of a function, as shown in the following listing.

Listing 2.5 Returning a `std::thread` from a function

```
std::thread f()
{
    void some_function();
    return std::thread(some_function);
}
std::thread g()
{
    void some_other_function(int);
    std::thread t(some_other_function,42);
    return t;
}
```

Likewise, if ownership should be transferred into a function, it can accept an instance of `std::thread` by value as one of the parameters, as shown here:

```
void f(std::thread t);
void g()
{
    void some_function();
    f(std::thread(some_function));
    std::thread t(some_function);
    f(std::move(t));
}
```

One benefit of the move support of `std::thread` is that you can build on the `thread_guard` class from listing 2.3 and have it take ownership of the thread. This avoids any unpleasant consequences should the `thread_guard` object outlive the thread it was referencing, and it also means that no one else can join or detach the thread once ownership has been transferred into the object. Because this would primarily be aimed at ensuring that threads are completed before a scope is exited, I named this class `scoped_thread`. The implementation is shown in the following listing, along with a simple example.

Listing 2.6 `scoped_thread` and example usage

```
class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_):
        t(std::move(t_))
```

```
    {
        if(!t.joinable())
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};
struct func;      <-------- See listing 2.1
void f()
{
    int some_local_state;
    scoped_thread t{std::thread(func(some_local_state))};
    do_something_in_current_thread();
}
```

The example is similar to listing 2.3, but the new thread is passed in directly to `scoped_thread` rather than having to create a separate named variable for it. When the initial thread reaches the end of `f`, the `scoped_thread` object is destroyed and then joins with the thread supplied to the constructor. Whereas with the `thread_guard` class from listing 2.3 the destructor had to check that the thread was still joinable, you can do that in the constructor and throw an exception if it's not.

One of the proposals for C++17 was for a `joining_thread` class that would be similar to `std::thread`, except that it would automatically join in the destructor much like `scoped_thread` does. This didn't get consensus in the committee, so it wasn't accepted into the standard (though it's still on track for C++20 as `std::jthread`), but it's relatively easy to write. One possible implementation is shown in the next listing.

Listing 2.7 A `joining_thread` class

```
class joining_thread
{
    std::thread t;
public:
    joining_thread() noexcept=default;
    template<typename Callable,typename ... Args>
    explicit joining_thread(Callable&& func,Args&& ... args):
        t(std::forward<Callable>(func),std::forward<Args>(args)...)
    {}
    explicit joining_thread(std::thread t_) noexcept:
        t(std::move(t_))
    {}
    joining_thread(joining_thread&& other) noexcept:
        t(std::move(other.t))
    {}
    joining_thread& operator=(joining_thread&& other) noexcept
    {
        if(joinable())
            join();
```

```
        t=std::move(other.t);
        return *this;
    }
    joining_thread& operator=(std::thread other) noexcept
    {
        if(joinable())
            join();
        t=std::move(other);
        return *this;
    }
    ~joining_thread() noexcept
    {
        if(joinable())
            join();
    }
    void swap(joining_thread& other) noexcept
    {
        t.swap(other.t);
    }
    std::thread::id get_id() const noexcept{
        return t.get_id();
    }
    bool joinable() const noexcept
    {
        return t.joinable();
    }
    void join()
    {
        t.join();
    }
    void detach()
    {
        t.detach();
    }
    std::thread& as_thread() noexcept
    {
        return t;
    }
    const std::thread& as_thread() const noexcept
    {
        return t;
    }
};
```

The move support in `std::thread` also allows for containers of `std::thread` objects, if those containers are move-aware (like the updated `std::vector<>`). This means that you can write code like that in the following listing, which spawns a number of threads and then waits for them to finish.

Listing 2.8 Spawns some threads and waits for them to finish

```
void do_work(unsigned id);
void f()
```

```
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)
    {
        threads.emplace_back(do_work,i);      <-- Spawns threads
    }
    for(auto& entry: threads)                 <-- Calls join() on each
        entry.join();                             thread in turn
}
```

If the threads are being used to subdivide the work of an algorithm, this is often what's required; before returning to the caller, all threads must have finished. The simple structure of listing 2.8 implies that the work done by the threads is self-contained, and the result of their operations is purely the side effects on shared data. If `f()` were to return a value to the caller that depended on the results of the operations performed by these threads, then as written, this return value would have to be determined by examining the shared data after the threads had terminated. Alternative schemes for transferring the results of operations between threads are discussed in chapter 4.

Putting `std::thread` objects in a `std::vector` is a step toward automating the management of those threads: rather than creating separate variables for those threads and joining with them directly, they can be treated as a group. You can take this a step further by creating a dynamic number of threads determined at runtime, rather than creating a fixed number, as in listing 2.8.

2.4 Choosing the number of threads at runtime

One feature of the C++ Standard Library that helps here is `std::thread::hardware_concurrency()`. This function returns an indication of the number of threads that can truly run concurrently for a given execution of a program. On a multicore system it might be the number of CPU cores, for example. This is only a hint, and the function might return `0` if this information isn't available, but it can be a useful guide for splitting a task among threads.

Listing 2.9 shows a simple implementation of a parallel version of `std::accumulate`. In real code you'll probably want to use the parallel version of `std::reduce` described in chapter 10, rather than implementing it yourself, but this illustrates the basic idea. It divides the work among the threads, with a minimum number of elements per thread in order to avoid the overhead of too many threads. Note that this implementation assumes that none of the operations will throw an exception, even though exceptions are possible; the `std::thread` constructor will throw if it can't start a new thread of execution, for example. Handling exceptions in such an algorithm is beyond the scope of this simple example and will be covered in chapter 8.

Listing 2.9 A naïve parallel version of `std::accumulate`

```
template<typename Iterator,typename T>
struct accumulate_block
```

```
{
    void operator()(Iterator first,Iterator last,T& result)
    {
        result=std::accumulate(first,last,result);
    }
};
template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);
    if(!length)
        return init;
    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread;
    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();
    unsigned long const num_threads=
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);
    unsigned long const block_size=length/num_threads;
    std::vector<T> results(num_threads);
    std::vector<std::thread>  threads(num_threads-1);
    Iterator block_start=first;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;
        std::advance(block_end,block_size);
        threads[i]=std::thread(
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;
    }
    accumulate_block<Iterator,T>()(
        block_start,last,results[num_threads-1]);

    for(auto& entry: threads)
           entry.join();
    return std::accumulate(results.begin(),results.end(),init);
}
```

Although this is a long function, it's straightforward. If the input range is empty, you return the initial value supplied as the `init` parameter value. Otherwise, there's at least one element in the range, so you can divide the number of elements to process by the minimum block size in order to give the maximum number of threads . This is to avoid creating 32 threads on a 32-core machine when you have only five values in the range.

The number of threads to run is the minimum of your calculated maximum and the number of hardware threads. You don't want to run more threads than the hardware can support (which is called *oversubscription*), because the context switching will mean that more threads will decrease the performance. If the call to `std::thread::hardware_concurrency()` returned `0`, you'd substitute a number of your choice; in

this case I've chosen `2`. You don't want to run too many threads because that would slow things down on a single-core machine, but likewise you don't want to run too few because you'd be passing up the available concurrency.

The number of entries for each thread to process is the length of the range divided by the number of threads. If you're worrying about cases where the number doesn't divide evenly, don't—you'll handle that later.

Now that you know how many threads you have, you can create a `std::vector<T>` for the intermediate results and a `std::vector<std::thread>` for the threads. Note that you need to launch one fewer thread than `num_threads`, because you already have one.

Launching the threads is a simple loop: advance the `block_end` iterator to the end of the current block and launch a new thread to accumulate the results for this block. The start of the next block is the end of this one.

After you've launched all the threads, this thread can then process the final block. This is where you take account of any uneven division: you know the end of the final block must be `last`, and it doesn't matter how many elements are in that block.

Once you've accumulated the results for the last block, you can wait for all the threads you spawned with `std::for_each`, as in listing 2.8, and then add up the results with a final call to `std::accumulate`.

Before you leave this example, it's worth pointing out that where the addition operator for the type `T` isn't associative (such as for `float` or `double`), the results of this `parallel_accumulate` may vary from those of `std::accumulate` because of the grouping of the range into blocks. Also, the requirements on the iterators are slightly more stringent: they must be at least *forward iterators*, whereas `std::accumulate` can work with single-pass *input iterators*, and `T` must be *default-constructible* so that you can create the `results` vector. These sorts of requirement changes are common with parallel algorithms; by their nature they're different in order to make them parallel, and this has consequences for the results and requirements. Implementing parallel algorithms is covered in more depth in chapter 8, and chapter 10 covers the standard supplied ones from C++17 (the equivalent to the `parallel_accumulate` described here being the parallel form of `std::reduce`). It's also worth noting that because you can't return a value directly from a thread, you must pass in a reference to the relevant entry in the `results` vector. Alternative ways of returning results from threads are addressed through the use of futures in chapter 4.

In this case, all the information required by each thread was passed in when the thread was started, including the location in which to store the result of its calculation. This isn't always the case; sometimes it's necessary to be able to identify the threads in some way for part of the processing. You could pass in an identifying number, such as the value of `i` in listing 2.8, but if the function that needs the identifier is several levels deep in the call stack and could be called from any thread, it's inconvenient to have to do it that way. When we were designing the C++ Standard Library we foresaw this need, so each thread has a unique identifier.

2.5 Identifying threads

Thread identifiers are of type `std::thread::id` and can be retrieved in two ways. First, the identifier for a thread can be obtained from its associated `std::thread` object by calling the `get_id()` member function. If the `std::thread` object doesn't have an associated thread of execution, the call to `get_id()` returns a default-constructed `std::thread::id` object, which indicates "not any thread." Alternatively, the identifier for the current thread can be obtained by calling `std::this_thread:: get_id()`, which is also defined in the `<thread>` header.

Objects of type `std::thread::id` can be freely copied and compared; they wouldn't be of much use as identifiers otherwise. If two objects of type `std::thread::id` are equal, they represent the same thread, or both are holding the "not any thread" value. If two objects aren't equal, they represent different threads, or one represents a thread and the other is holding the "not any thread" value.

The C++ Standard Library doesn't limit you to checking whether thread identifiers are the same or not; objects of type `std::thread::id` offer the complete set of comparison operators, which provide a total ordering for all distinct values. This allows them to be used as keys in associative containers, or sorted, or compared in any other way that you as a programmer may see fit. The comparison operators provide a total order for all non-equal values of `std::thread::id`, so they behave as you'd intuitively expect: if `a<b` and `b<c`, then `a<c`, and so forth. The Standard Library also provides `std::hash<std::thread::id>` so that values of type `std::thread::id` can be used as keys in the new unordered associative containers too.

Instances of `std::thread::id` are often used to check whether a thread needs to perform some operation. For example, if threads are used to divide work, as in listing 2.9, the initial thread that launched the others might need to perform its work slightly differently in the middle of the algorithm. In this case it could store the result of `std::this_thread::get_id()` before launching the other threads, and then the core part of the algorithm (which is common to all threads) could check its own thread ID against the stored value:

```
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
    if(std::this_thread::get_id()==master_thread)
    {
        do_master_thread_work();
    }
    do_common_work();
}
```

Alternatively, the `std::thread::id` of the current thread could be stored in a data structure as part of an operation. Later operations on that same data structure could then check the stored ID against the ID of the thread performing the operation to determine what operations are permitted/required.

Similarly, thread IDs could be used as keys into associative containers where specific data needs to be associated with a thread and alternative mechanisms such as thread-local storage aren't appropriate. Such a container could, for example, be used by a controlling thread to store information about each of the threads under its control or for passing information between threads.

The idea is that `std::thread::id` will suffice as a generic identifier for a thread in most circumstances; it's only if the identifier has semantic meaning associated with it (such as being an index into an array) that alternatives should be necessary. You can even write out an instance of `std::thread::id` to an output stream such as `std::cout`:

```
std::cout<<std::this_thread::get_id();
```

The exact output you get is strictly implementation-dependent; the only guarantee given by the standard is that thread IDs that compare as equal should produce the same output, and those that aren't equal should give different output. This is therefore primarily useful for debugging and logging, but the values have no semantic meaning, so there's not much more that could be said anyway.

Summary

In this chapter, I covered the basics of thread management with the C++ Standard Library: starting threads, waiting for them to finish, and *not* waiting for them to finish because you want them to run in the background. We also saw how to pass arguments into the thread function when a thread is started, how to transfer the responsibility for managing a thread from one part of the code to another, and how groups of threads can be used to divide work. Finally, we discussed identifying threads in order to associate data or behavior with specific threads that's inconvenient to associate through alternative means. Although you can do quite a lot with purely independent threads that each operate on separate data, sometimes it's desirable to share data among threads while they're running. Chapter 3 discusses the issues surrounding sharing data directly among threads, and chapter 4 covers more general issues surrounding synchronizing operations with and without shared data.