

# **C++ is fun – Part 15**

## at Turbine/Warner Bros.!

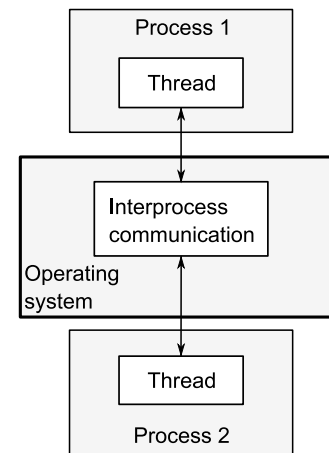
Russell Hanson

# Syllabus

- 1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment Mar 25-27
- 2) Objects, encapsulation, abstract data types, data protection and scope April 1-3
- 3) Basic data structures and how to use them, opening files and performing operations on files – April 8-10
- 4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms April 15-17
- Project 1 Due – April 17
- 5) More AI: search, heuristics, optimization, decision trees, supervised/unsupervised learning – April 22-24
- 6) Game API and/or event-oriented programming, model view controller, map reduce filter – April 29, May 1
- 7) Basic threads models and some simple databases SQLite May 6-8
- 8) Graphics programming, shaders, textures, 3D models and rotations May 13-15
- Project 2 Due May 15
- 9) Threads, Atomic, and Exceptions, more May 20
- 10) Gesture recognition & depth controllers like the Microsoft Kinect, Network Programming & TCP/IP, OSC May 27
- 11) Selected Topics June 3
- 12) Working on student projects - June 10
- Final project presentations Project 3/Final Project Due June 10

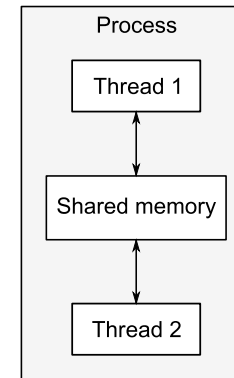
### CONCURRENCY WITH MULTIPLE PROCESSES

The first way to make use of concurrency within an application is to divide the application into multiple, separate, single-threaded processes that are run at the same time, much as you can run your web browser and word processor at the same time. These separate processes can then pass messages to each other through all the normal inter-process communication channels (signals, sockets, files, pipes, and so on), as shown in figure 1.3. One downside is that such communication between processes is often either complicated to set up or slow or both, because operating systems typically provide a lot of protection between processes to avoid one process accidentally modifying data belonging to another process. Another downside is that there's an inherent overhead in running multiple processes: it takes time to start a process, the operating system must devote internal resources to managing the process, and so forth.



**Figure 1.3 Communication between a pair of processes running concurrently**

The shared address space and lack of protection of data between threads makes the overhead associated with using multiple threads much smaller than that from using multiple processes, because the operating system has less bookkeeping to do. But the flexibility of shared memory also comes with a price: if data is accessed by multiple threads, the application programmer must ensure that the view of data seen by each thread is consistent whenever it is accessed. The issues surrounding sharing data between threads and the tools to use and guidelines to follow to avoid problems are covered throughout this book, notably in chapters 3, 4, 5, and 8. The problems are not insurmountable, provided suitable care is taken when writing the code, but they do mean that a great deal of thought must go into the communication between threads.



**Figure 1.4** Communication between a pair of threads running concurrently in a single process

## Hello, Concurrent World

Let's start with a classic example: a program to print "Hello World." A really simple Hello, World program that runs in a single thread is shown here, to serve as a baseline when we move to multiple threads:

```
#include <iostream>

int main()
{
    std::cout<<"Hello World\n";
}
```

All this program does is write "Hello World" to the standard output stream. Let's compare it to the simple Hello, Concurrent World program shown in the following listing, which starts a separate thread to display the message.

### Listing 1.1 A simple Hello, Concurrent World program

```
#include <iostream>
#include <thread>           ← 1

void hello()               ← 2
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);  ← 3
    t.join();             ← 4
}
```

The first difference is the extra `#include <thread>` 1. The declarations for the multi-threading support in the Standard C++ Library are in new headers: the functions and classes for managing threads are declared in `<thread>`, whereas those for protecting shared data are declared in other headers.

this case, the `std::thread` object named `t` ❸ has the new function `hello()` as its initial function.

This is the next difference: rather than just writing directly to standard output or calling `hello()` from `main()`, this program launches a whole new thread to do it, bringing the thread count to two—the initial thread that starts at `main()` and the new thread that starts at `hello()`.

After the new thread has been launched ❸, the initial thread continues execution. If it didn't wait for the new thread to finish, it would merrily continue to the end of `main()` and thus end the program—possibly before the new thread had had a chance to run. This is why the call to `join()` is there ❹—as described in chapter 2, this causes the calling thread (in `main()`) to wait for the thread associated with the `std::thread` object, in this case, `t`.

## Class Exercise, join a thread

```
#include <iostream>
#include <thread>

// int main() {
//   std::cout<<"Hello World\n";
// }

void hello(){
  std::cout<<"Hello Concurrent World\n";
}

int main() {
  std::thread t(hello);
  t.join();
}
```

**Hello Concurrent World**

## Launching a thread

As you saw in chapter 1, threads are started by constructing a `std::thread` object that specifies the task to run on that thread. In the simplest case, that task is just a plain, ordinary `void`-returning function that takes no parameters. This function runs on its own thread until it returns, and then the thread stops. At the other extreme, the task could be a function object that takes additional parameters and performs a series of independent operations that are specified through some kind of messaging system while it's running, and the thread stops only when it's signaled to do so, again via some kind of messaging system. It doesn't matter what the thread is going to do or where it's launched from, but starting a thread using the C++ Thread Library always boils down to constructing a `std::thread` object:

```
void do_some_work();
std::thread my_thread(do_some_work);
```

This is just about as simple as it gets. Of course, you have to make sure that the `<thread>` header is included so the compiler can see the definition of the `std::thread` class. As with much of the C++ Standard Library, `std::thread` works with any *callable* type, so you can pass an instance of a class with a function call operator to the `std::thread` constructor instead:

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};
background_task f;
std::thread my_thread(f);
```

In this case, the supplied function object is *copied* into the storage belonging to the newly created thread of execution and invoked from there. It's therefore essential that the copy behave equivalently to the original, or the result may not be what's expected.



For example,

```
std::thread my_thread(background_task());
```

declares a function `my_thread` that takes a single parameter (of type pointer to a function taking no parameters and returning a `background_task` object) and returns a `std::thread` object, rather than launching a new thread. You can avoid this by naming your function object as shown previously, by using an extra set of parentheses, or by using the new uniform initialization syntax, for example:

```
std::thread my_thread((background_task()));           ← ❶  
std::thread my_thread{background_task()};           ← ❷
```

In the first example ❶, the extra parentheses prevent interpretation as a function declaration, thus allowing `my_thread` to be declared as a variable of type `std::thread`. The second example ❷ uses the new uniform initialization syntax with braces rather than parentheses, and thus would also declare a variable.

One type of callable object that avoids this problem is a *lambda expression*. This is a new feature from C++11 which essentially allows you to write a local function, possibly capturing some local variables and avoiding the need of passing additional arguments (see section 2.2). For full details on lambda expressions, see appendix A, section A.5. The previous example can be written using a lambda expression as follows:

```
std::thread my_thread([](  
    do_something();  
    do_something_else();  
});
```

## Listing 2.1 A function that returns while a thread still has access to local variables

```
struct func
{
    int& i;
    func(int& i_):i(i_){}
    void operator() ()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};

void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

① Potential access to dangling reference

② Don't wait for thread to finish

③ New thread might still be running

In this case, the new thread associated with `my_thread` will probably still be running when `oops` exits ③, because you've explicitly decided not to wait for it by calling `detach()` ②. If the thread *is* still running, then the next call to `do_something(i)` ① will access an already destroyed variable. This is just like normal single-threaded code—allowing a pointer or reference to a local variable to persist beyond the function exit is never a good idea—but it's easier to make the mistake with multithreaded code, because it isn't necessarily immediately apparent that this has happened.

One common way to handle this scenario is to make the thread function self-contained and *copy* the data into the thread rather than sharing the data. If you use a callable object for your thread function, that object is itself copied into the thread, so the original object can be destroyed immediately. But you still need to be wary of objects containing pointers or references, such as that from listing 2.1. In particular, it's a bad idea to create a thread within a function that has access to the local variables in that function, unless the thread is guaranteed to finish before the function exits.

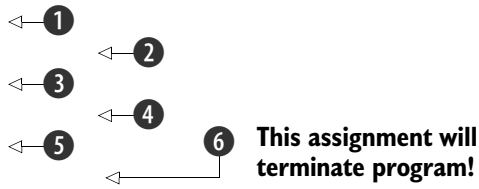
Alternatively, you can ensure that the thread has completed execution before the function exits by *joining* with the thread.

# Transferring ownership of a thread

threads among three `std::thread` instances, `t1`, `t2`, and `t3`:

```
void some_function();
void some_other_function();
std::thread t1(some_function);           ← ❶
std::thread t2=std::move(t1);           ← ❷
t1=std::thread(some_other_function);    ← ❸
std::thread t3;                          ← ❹
t3=std::move(t2);                         ← ❺
t1=std::move(t3);                         ← ❻
```

**This assignment will terminate program!**



First, a new thread is started ❶ and associated with `t1`. Ownership is then transferred over to `t2` when `t2` is constructed, by invoking `std::move()` to explicitly move ownership ❷. 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 ❸. 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 ❹, which means that it's created without any associated thread of execution. Ownership of the thread currently associated with `t2` is transferred into `t3` ❺, 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 ❻ 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.

## Pthreads is another alternative

### Creating Threads:

There is following routine which we use to create a POSIX thread:

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

Here **pthread\_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters:

Parameter	Description
thread	An opaque, unique identifier for the new thread returned by the subroutine.
attr	An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
start_routine	The C++ routine that the thread will execute once it is created.
arg	A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

### Terminating Threads:

There is following routine which we use to terminate a POSIX thread:

```
#include <pthread.h>
pthread_exit (status)
```

Here **pthread\_exit** is used to explicitly exit a thread. Typically, the pthread\_exit() routine is called after a thread has completed its work and is no longer required to exist.

If main() finishes before the threads it has created, and exits with pthread\_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes.

```

#include <iostream>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    cout << "Hello World! Thread ID, " << tid << endl;
    pthread_exit(NULL);
}

int main ()
{
    pthread_t threads[NUM_THREADS];
    int rc;
    int i;
    for( i=0; i < NUM_THREADS; i++ ){
        cout << "main() : creating thread, " << i << endl;
        rc = pthread_create(&threads[i], NULL,
            PrintHello, (void *)i);
        if (rc){
            cout << "Error:unable to create thread," << rc << endl;
            exit(-1);
        }
    }
    pthread_exit(NULL);
}

```

## Class Exercise, using pthreads

```

main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4

```

## Passing Arguments to Threads

```
#include <iostream>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
struct thread_data{
    int thread_id;
    char *message;
};
void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    my_data = (struct thread_data *) threadarg;
    cout << "Thread ID : " << my_data->thread_id ;
    cout << " Message : " << my_data->message << endl;
    pthread_exit(NULL);
}

int main ()
{
    pthread_t threads[NUM_THREADS];
    struct thread_data td[NUM_THREADS];
    int rc;
    int l;
    for( i=0; i < NUM_THREADS; i++){
        cout <<"main() : creating thread, " << i << endl;
        td[i].thread_id = i;
        td[i].message = "This is message";
        rc = pthread_create(&threads[i], NULL,
            PrintHello, (void *)&td[i]);
        if (rc){
            cout << "Error:unable to create thread," << rc << endl;
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

**main() : creating thread, 0**

**main() : creating thread, 1**

**main() : creating thread, 2**

**main() : creating thread, 3**

**main() : creating thread, 4**

**Thread ID : 1 Message : This is message**

**Thread ID : 0 Message : This is message**

**Thread ID : 2 Message : This is message**

**Thread ID : 3 Message : This is message**

**Thread ID : 4 Message : This is message**

## Joining and Detaching Threads:

There are following two routines which we can use to join or detach threads:

```
pthread_join (threadid, status)  
pthread_detach (threadid)
```

The `pthread_join()` subroutine blocks the calling thread until the specified `threadid` thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

```

#include <iostream>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

void *wait(void *t)
{
    int i;
    long tid;

    tid = (long)t;

    sleep(1); //
    cout << "Sleeping in thread " << endl;
    cout << "Thread with id : " << tid << " ...exiting " << endl;
    pthread_exit(NULL);
}

int main ()
{
    int rc;
    int i;
    pthread_t threads[NUM_THREADS];
    pthread_attr_t attr;
    void *status;

    // Initialize and set thread joinable
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for( i=0; i < NUM_THREADS; i++){
        cout << "main() : creating thread, " << i << endl;
        rc = pthread_create(&threads[i], NULL, wait, (void *)i );
        if (rc){
            cout << "Error:unable to create thread," << rc << endl;
            exit(-1);
        }
    }
}

```

```

// free attribute and wait for the other threads
pthread_attr_destroy(&attr);
for( i=0; i < NUM_THREADS; i++){
    rc = pthread_join(threads[i], &status);
    if (rc){
        cout << "Error:unable to join," << rc << endl;
        exit(-1);
    }
    cout << "Main: completed thread id : " << i ;
    cout << " exiting with status : " << (int)status << endl;
}

cout << "Main: program exiting." << endl;
pthread_exit(NULL);
}

```

```

main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 ...exiting
Sleeping in thread
Thread with id : 1 ...exiting
Sleeping in thread
Thread with id : 2 ...exiting
Sleeping in thread
Thread with id : 3 ...exiting
Sleeping in thread
Thread with id : 4 ...exiting

```



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 2.7 Spawn some threads and wait for them to finish

```
void do_work(unsigned id);

void f()
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)
    {
        threads.push_back(std::thread(do_work,i));
    }
    std::for_each(threads.begin(),threads.end(),
                  std::mem_fn(&std::thread::join));
}
```

← **Spawn threads**

← **Call join() on each thread in turn**

If the threads are being used to subdivide the work of an algorithm, this is often just what's required; before returning to the caller, all threads must have finished. Of course, the simple structure of listing 2.7 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

## Listing 2.8 A naive 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) ← 1
        return init;

    unsigned long const min_per_thread=25;
    unsigned long const max_threads=
        (length+min_per_thread-1)/min_per_thread; ← 2

    unsigned long const hardware_threads=
        std::thread::hardware_concurrency();
    unsigned long const num_threads= ← 3
        std::min(hardware_threads!=0?hardware_threads:2,max_threads);

    unsigned long const block_size=length/num_threads; ← 4

    std::vector<T> results(num_threads);
    std::vector<std::thread> threads(num_threads-1); ← 5

    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); ← 6
        threads[i]=std::thread( ← 7
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i])); ← 8
        block_start=block_end;
    }
    accumulate_block<Iterator,T>()( ← 9
        block_start,last,results[num_threads-1]);

    std::for_each(threads.begin(),threads.end(), ← 10
        std::mem_fn(&std::thread::join));

    return std::accumulate(results.begin(),results.end(),init); ← 11
}

```

## And the bullet-point by bullet-point explanation/elaboration

Although this is quite a long function, it's actually straightforward. If the input range is empty ❶, you just return the initial value `init`. 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 ❷. 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 ❸. 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 simply 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 then 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 ❹. If you're worrying about the case 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 ❺. Note that you need to launch one fewer thread than `num_threads`, because you already have one.

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

After you've launched all the threads, this thread can then process the final block ❾. 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` ❿, as in listing 2.7, and then add up the results with a final call to `std::accumulate` ⓫.

# Thread Identifiers

## *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 Thread 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 2.8, 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:

# Thread Identifiers

```
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();
```

## **Problems with sharing data between threads**

When it comes down to it, the problems with sharing data between threads are all due to the consequences of modifying data. *If all shared data is read-only, there's no problem, because the data read by one thread is unaffected by whether or not another thread is reading the same data.* However, if data is shared between threads, and one or more threads start modifying the data, there's a lot of potential for trouble. In this case, you must take care to ensure that everything works out OK.

One concept that's widely used to help programmers reason about their code is that of *invariants*—statements that are always true about a particular data structure, such as “this variable contains the number of items in the list.” These invariants are often broken during an update, especially if the data structure is of any complexity or the update requires modification of more than one value.

Consider a doubly linked list, where each node holds a pointer to both the next node in the list and the previous one. One of the invariants is that if you follow a “next” pointer from one node (A) to another (B), the “previous” pointer from that node (B) points back to the first node (A). In order to remove a node from the list, the nodes on either side have to be updated to point to each other. Once one has been updated, the invariant is broken until the node on the other side has been updated too; after the update has completed, the invariant holds again.

The steps in deleting an entry from such a list are shown in figure 3.1:

- 1 Identify the node to delete (N).
- 2 Update the link from the node prior to N to point to the node after N.
- 3 Update the link from the node after N to point to the node prior to N.
- 4 Delete node N.

As you can see, between steps b and c, the links going in one direction are inconsistent with the links going in the opposite direction, and the invariant is broken.

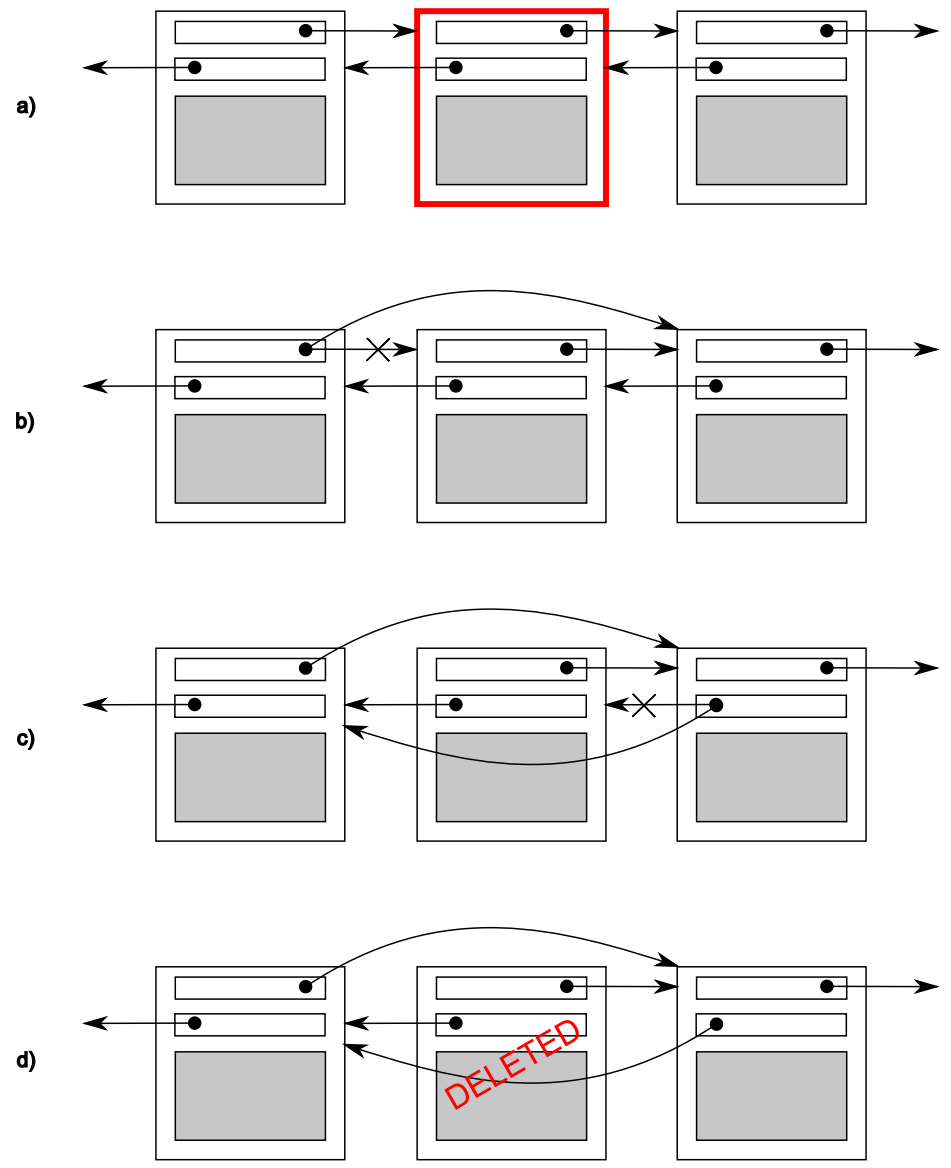


Figure 3.1 Deleting a node from a doubly linked list

### **Avoiding problematic race conditions**

There are several ways to deal with problematic race conditions. The simplest option is to wrap your data structure with a protection mechanism, to ensure that only the thread actually performing a modification can see the intermediate states where the invariants are broken. From the point of view of other threads accessing that data structure,

such modifications either haven't started or have completed. The C++ Standard Library provides several such mechanisms, which are described in this chapter.

Another option is to modify the design of your data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants. This is generally referred to as *lock-free programming* and is difficult to get right. If you're working at this level, the nuances of the memory model and identifying which threads can potentially see which set of values can get complicated. The memory model is covered in chapter 5, and lock-free programming is discussed in chapter 7.

Another way of dealing with race conditions is to handle the updates to the data structure as a *transaction*, just as updates to a database are done within a transaction. The required series of data modifications and reads is stored in a transaction log and then committed in a single step. If the commit can't proceed because the data structure has been modified by another thread, the transaction is restarted. This is termed *software transactional memory (STM)*, and it's an active research area at the time of writing. This won't be covered in this book, because there's no direct support for STM in C++. However, the basic idea of doing something privately and then committing in a single step is something that I'll come back to later.

The most basic mechanism for protecting shared data provided by the C++ Standard is the *mutex*, so we'll look at that first.



# mutex – *mutual exclusion*

## ***Protecting shared data with mutexes***

So, you have a shared data structure such as the linked list from the previous section, and you want to protect it from race conditions and the potential broken invariants that can ensue. Wouldn't it be nice if you could mark all the pieces of code that access the data structure as *mutually exclusive*, so that if any thread was running one of them, any other thread that tried to access that data structure had to wait until the first thread was finished? That would make it impossible for a thread to see a broken invariant except when it was the thread doing the modification.

Well, this isn't a fairy tale wish—it's precisely what you get if you use a synchronization primitive called a *mutex* (*mutual exclusion*). Before accessing a shared data structure, you *lock* the mutex associated with that data, and when you've finished accessing the data structure, you *unlock* the mutex. The Thread Library then ensures that once one thread has locked a specific mutex, all other threads that try to lock the same mutex have to wait until the thread that successfully locked the mutex unlocks it. This ensures that all threads see a self-consistent view of the shared data, without any broken invariants.

Mutexes are the most general of the data-protection mechanisms available in C++, but they're not a silver bullet; it's important to structure your code to protect the right data (see section 3.2.2) and avoid race conditions inherent in your interfaces (see section 3.2.3). Mutexes also come with their own problems, in the form of a *deadlock* (see section 3.2.4) and protecting either too much or too little data (see section 3.2.8). Let's start with the basics.

### Using mutexes in C++

In C++, you create a mutex by constructing an instance of `std::mutex`, lock it with a call to the member function `lock()`, and unlock it with a call to the member function `unlock()`. However, it isn't recommended practice to call the member functions directly, because this means that you have to remember to call `unlock()` on every code path out of a function, including those due to exceptions. Instead, the Standard C++ Library provides the `std::lock_guard` class template, which implements that RAII idiom for a mutex; it locks the supplied mutex on construction and unlocks it on destruction, thus ensuring a locked mutex is always correctly unlocked. The following listing shows how to protect a list that can be accessed by multiple threads using a `std::mutex`, along with `std::lock_guard`. Both of these are declared in the `<mutex>` header.

**Listing 3.1** Protecting a list with a mutex

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;           ← 1
std::mutex some_mutex;            ← 2

void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex); ← 3
    some_list.push_back(new_value);
}

bool list_contains(int value_to_find)
{
    std::lock_guard<std::mutex> guard(some_mutex); ← 4
    return std::find(some_list.begin(), some_list.end(), value_to_find)
        != some_list.end();
}
```

In listing 3.1, there's a single global variable ①, and it's protected with a corresponding global instance of `std::mutex` ②. The use of `std::lock_guard<std::mutex>` in `add_to_list()` ③ and again in `list_contains()` ④ means that the accesses in these functions are mutually exclusive: `list_contains()` will never see the list partway through a modification by `add_to_list()`.

### Listing 3.4 An outline class definition for a thread-safe stack

```
#include <exception>
#include <memory>
struct empty_stack: std::exception
{
    const char* what() const throw();
};
template<typename T>
class threadsafe_stack
{
public:
    threadsafe_stack();
    threadsafe_stack(const threadsafe_stack&);
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value);
    std::shared_ptr<T> pop();
    void pop(T& value);
    bool empty() const;
};
```

← For `std::shared_ptr<>`

Assignment operator is deleted ① ←

By paring down the interface you allow for maximum safety; even operations on the whole stack are restricted. The stack itself can't be assigned, because the assignment operator is deleted ① (see appendix A, section A.2), and there's no `swap()` function. It can, however, be copied, assuming the stack elements can be copied. The `pop()` functions throw an `empty_stack` exception if the stack is empty, so everything still works even if the stack is modified after a call to `empty()`. As mentioned in the description of option 3, the use of `std::shared_ptr` allows the stack to take care of the memory-allocation issues and avoid excessive calls to `new` and `delete` if desired. Your five stack operations have now become three: `push()`, `pop()`, and `empty()`. Even `empty()` is superfluous. This simplification of the interface allows for better control over the data; you can ensure that the mutex is locked for the entirety of an operation. The following listing shows a simple implementation that's a wrapper around `std::stack<>`.

exchanges data between two instances of the same class; in order to ensure that the data is exchanged correctly, without being affected by concurrent modifications, the mutexes on both instances must be locked. However, if a fixed order is chosen (for example, the mutex for the instance supplied as the first parameter, then the mutex for the instance supplied as the second parameter), this can backfire: all it takes is for two threads to try to exchange data between the same two instances with the parameters swapped, and you have deadlock!

Thankfully, the C++ Standard Library has a cure for this in the form of `std::lock`—a function that can lock two or more mutexes at once without risk of deadlock. The example in the next listing shows how to use this for a simple swap operation.

### Listing 3.6 Using `std::lock()` and `std::lock_guard` in a swap operation

```
class some_big_object;
void swap(some_big_object& lhs, some_big_object& rhs);

class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd) {}

    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::lock(lhs.m, rhs.m);
        std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
        std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
        swap(lhs.some_detail, rhs.some_detail);
    }
};
```

First, the arguments are checked to ensure they are different instances, because attempting to acquire a lock on a `std::mutex` when you already hold it is undefined behavior. (A mutex that does permit multiple locks by the same thread is provided in the form of `std::recursive_mutex`. See section 3.3.3 for details.) Then, the call to `std::lock()` **1** locks the two mutexes, and two `std::lock_guard` instances are constructed **2**, **3**, one for each mutex. The `std::adopt_lock` parameter is supplied in addition to the mutex to indicate to the `std::lock_guard` objects that the mutexes are already locked, and they should just adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex in the constructor.

### **Further guidelines for avoiding deadlock**

Deadlock doesn't just occur with locks, although that's the most frequent cause; you can create deadlock with two threads and no locks just by having each thread call `join()` on the `std::thread` object for the other. In this case, neither thread can make progress because it's waiting for the other to finish, just like the children fighting over their toys. This simple cycle can occur anywhere that a thread can wait for another thread to perform some action if the other thread can simultaneously be waiting for the first thread, and it isn't limited to two threads: a cycle of three or more threads will still cause deadlock. The guidelines for avoiding deadlock all boil down to one idea: don't wait for another thread if there's a chance it's waiting for you. The individual guidelines provide ways of identifying and eliminating the possibility that the other thread is waiting for you.

#### **AVOID NESTED LOCKS**

The first idea is the simplest: don't acquire a lock if you already hold one. If you stick to this guideline, it's impossible to get a deadlock from the lock usage alone because each thread only ever holds a single lock. You could still get deadlock from other things (like the threads waiting for each other), but mutex locks are probably the most common cause of deadlock. If you need to acquire multiple locks, do it as a single action with `std::lock` in order to acquire them without deadlock.

#### **AVOID CALLING USER-SUPPLIED CODE WHILE HOLDING A LOCK**

This is a simple follow-on from the previous guideline. Because the code is user supplied, you have no idea what it could do; it could do anything, including acquiring a lock. If you call user-supplied code while holding a lock, and that code acquires a lock, you've violated the guideline on avoiding nested locks and could get deadlock. Sometimes this is unavoidable; if you're writing generic code such as the stack in section 3.2.3, every operation on the parameter type or types is user-supplied code. In this case, you need a new guideline.

#### **ACQUIRE LOCKS IN A FIXED ORDER**

If you absolutely must acquire two or more locks, and you can't acquire them as a single operation with `std::lock`, the next-best thing is to acquire them in the same

### Listing 3.10 Locking one mutex at a time in a comparison operator

```
class Y
{
private:
    int some_detail;
    mutable std::mutex m;

    int get_detail() const
    {
        std::lock_guard<std::mutex> lock_a(m);    ← ❶
        return some_detail;
    }
public:
    Y(int sd):some_detail(sd){}

    friend bool operator==(Y const& lhs, Y const& rhs)
    {
        if(&lhs==&rhs)
            return true;
        int const lhs_value=lhs.get_detail();
        int const rhs_value=rhs.get_detail();    ← ❷
        return lhs_value==rhs_value;           ← ❸
    }
};
```

In this case, the comparison operator first retrieves the values to be compared by calling the `get_detail()` member function ❷, ❸. This function retrieves the value while

protecting it with a lock ❶. The comparison operator then compares the retrieved values ❹. Note, however, that as well as reducing the locking periods so that only one lock is held at a time (and thus eliminating the possibility of deadlock), *this has subtly changed the semantics of the operation* compared to holding both locks together. In listing 3.10, if the operator returns `true`, it means that the value of `lhs.some_detail` at one point in time is equal to the value of `rhs.some_detail` at another point in time. The two values could have been changed in any way in between the two reads; the values could have been swapped in between ❷ and ❸, for example, thus rendering the comparison meaningless. The equality comparison might thus return `true` to indicate that the values were equal, even though there was never an instant in time when the values were actually equal. It's therefore important to be careful when making such changes that the semantics of the operation are not changed in a problematic fashion: *if you don't hold the required locks for the entire duration of an operation, you're exposing yourself to race conditions.*

The following listing shows a simple DNS cache like the one just described, using a `std::map` to hold the cached data, protected using a `boost::shared_mutex`.

### Listing 3.13 Protecting a data structure with a `boost::shared_mutex`

```
#include <map>
#include <string>
#include <mutex>
#include <boost/thread/shared_mutex.hpp>

class dns_entry;

class dns_cache
{
    std::map<std::string,dns_entry> entries;
    mutable boost::shared_mutex entry_mutex;
public:
    dns_entry find_entry(std::string const& domain) const
    {
        boost::shared_lock<boost::shared_mutex> lk(entry_mutex); ← ❶
        std::map<std::string,dns_entry>::const_iterator const it=
            entries.find(domain);
        return (it==entries.end())?dns_entry():it->second;
    }
    void update_or_add_entry(std::string const& domain,
                            dns_entry const& dns_details)
    {
        std::lock_guard<boost::shared_mutex> lk(entry_mutex); ← ❷
        entries[domain]=dns_details;
    }
};
```

In listing 3.13, `find_entry()` uses an instance of `boost::shared_lock<>` to protect it for shared, read-only access ❶; multiple threads can therefore call `find_entry()` simultaneously without problems. On the other hand, `update_or_add_entry()` uses an instance of `std::lock_guard<>` to provide exclusive access while the table is updated ❷; not only are other threads prevented from doing updates in a call `update_or_add_entry()`, but threads that call `find_entry()` are blocked too.

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();
    }
}
```

The diagram consists of three numbered callouts with arrows pointing to specific lines in the code. Callout 1, labeled 'Unlock the mutex', has an arrow pointing to the `lk.unlock();` line. Callout 2, labeled 'Sleep for 100 ms', has an arrow pointing to the `std::this_thread::sleep_for(std::chrono::milliseconds(100));` line. Callout 3, labeled 'Relock the mutex', has an arrow pointing to the `lk.lock();` line.

In the loop, the function unlocks the mutex ① before the sleep ② and locks it again afterward ③, so another thread gets a chance to acquire it and set the flag.



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

```
std::mutex mut;
std::queue<data_chunk> data_queue;   ← ❶
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);       ← ❷
        data_cond.notify_one();     ← ❸
    }
}

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut); ← ❹

        data_cond.wait(
            lk, []{return !data_queue.empty();}); ← ❺
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();                ← ❻
        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()`.

# On Windows, setting threads for each processor

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

HANDLE *m_threads = NULL;
DWORD_PTR WINAPI threadMain(void* p);

DWORD_PTR GetNumCPUs() {
    SYSTEM_INFO m_si = {0, };
    GetSystemInfo(&m_si);
    return (DWORD_PTR)m_si.dwNumberOfProcessors;
}

int wmain(int argc, wchar_t **args) {
    DWORD_PTR c = GetNumCPUs();
    m_threads = new HANDLE[c];
    for(DWORD_PTR i = 0; i < c; i++) {
        DWORD_PTR m_id = 0;
        DWORD_PTR m_mask = 1 << i;
        m_threads[i] = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)threadMain, (LPVOID)i, NULL, &m_id);
        SetThreadAffinityMask(m_threads[i], m_mask);

        wprintf(L"Creating Thread %d (0x%08x) Assigning to
CPU 0x%08x\r\n", i, (LONG_PTR)m_threads[i], m_mask);
    }
    return 0;
}

DWORD_PTR WINAPI threadMain(void* p) {
    return 0;
}
```

# std::thread::hardware\_concurrency

---

`static unsigned hardware_concurrency();` (since C++11)

Returns number of concurrent threads supported by the implementation. The value should be considered only a hint.

## Parameters

(none)

## Return value

number of concurrent threads supported. If the value is not well defined or not computable, returns 0.

## Exceptions

`noexcept` specification: `noexcept` (since C++11)

## Example

[run this code](#) 

```
#include <iostream>
#include <thread>

int main() {
    unsigned int n = std::thread::hardware_concurrency();
    std::cout << n << " concurrent threads are supported.\n";
}
```

```
#include <iostream>
```

```
#include <thread>
```

```
int main() {
```

```
    unsigned int n = std::thread::hardware_concurrency();
```

```
    std::cout << n << " concurrent threads are supported.\n";
```

```
}
```

**2 concurrent threads are supported.**

# Using atomic types – indivisible operations can't be half-done

## **Atomic operations and types in C++**

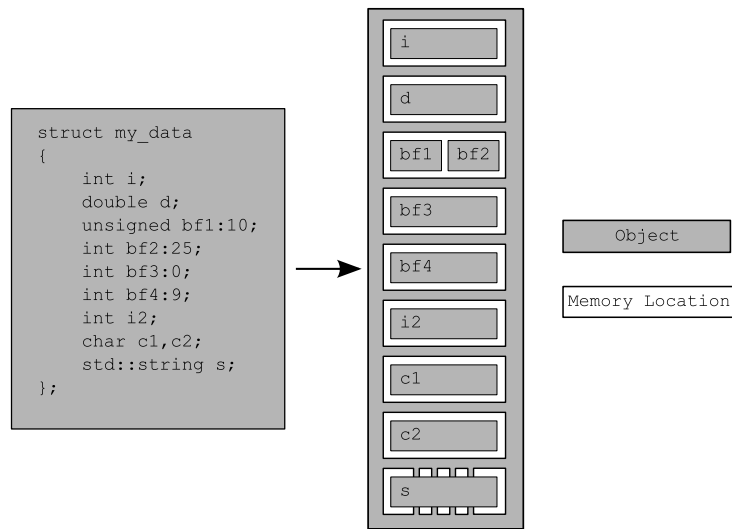
An *atomic operation* is an indivisible operation. You can't observe such an operation half-done from any thread in the system; it's either done or not done. If the load operation that reads the value of an object is *atomic*, and all modifications to that object are also *atomic*, that load will retrieve either the initial value of the object or the value stored by one of the modifications.

The flip side of this is that a nonatomic operation might be seen as half-done by another thread. If that operation is a store, the value observed by another thread might be neither the value before the store nor the value stored but something else. If the nonatomic operation is a load, it might retrieve part of the object, have another thread modify the value, and then retrieve the remainder of the object, thus retrieving neither the first value nor the second but some combination of the two. This is a simple problematic race condition, as described in chapter 3, but at this level it may constitute a *data race* (see section 5.1) and thus cause undefined behavior.

In C++, you need to use an atomic type to get an atomic operation in most cases, so let's look at those.

## **The standard atomic types**

The standard *atomic types* can be found in the `<atomic>` header. All operations on such types are atomic, and only operations on these types are atomic in the sense of the language definition, although you can use mutexes to make other operations *appear* atomic. In actual fact, the standard atomic types themselves might use such emulation: they (almost) all have an `is_lock_free()` member function, which allows the user to determine whether operations on a given type are done directly with atomic instructions (`x.is_lock_free()` returns `true`) or done by using a lock internal to the compiler and library (`x.is_lock_free()` returns `false`).



**Figure 5.1** The division of a struct into objects and memory locations

There are four important things to take away from this:

- Every variable is an object, including those that are members of other objects.
- Every object occupies *at least one* memory location.
- Variables of fundamental type such as int or char are *exactly one* memory location, whatever their size, even if they're adjacent or part of an array.
- Adjacent bit fields are part of the same memory location.

I'm sure you're wondering what this has to do with concurrency, so let's take a look.

### **Objects, memory locations, and concurrency**

Now, here's the part that's crucial for multithreaded applications in C++: everything hinges on those memory locations. If two threads access *separate* memory locations, there's no problem: everything works fine. On the other hand, if two threads access the *same* memory location, then you have to be careful. If neither thread is updating the memory location, you're fine; read-only data doesn't need protection or synchronization. If either thread is modifying the data, there's a potential for a race condition, as described in chapter 3.

In order to avoid the race condition, there has to be an enforced ordering between the accesses in the two threads. One way to ensure there's a defined ordering is to use mutexes as described in chapter 3; if the same mutex is locked prior to both accesses, only one thread can access the memory location at a time, so one must happen before the other. The other way is to use the synchronization properties of *atomic* operations (see section 5.2 for the definition of atomic operations) either on the same or other memory locations to enforce an ordering between the accesses in the two

**Table 5.1** The alternative names for the standard atomic types and their corresponding `std::atomic<>` specializations

Atomic type	Corresponding specialization
<code>atomic_bool</code>	<code>std::atomic&lt;bool&gt;</code>
<code>atomic_char</code>	<code>std::atomic&lt;char&gt;</code>
<code>atomic_schar</code>	<code>std::atomic&lt;signed char&gt;</code>
<code>atomic_uchar</code>	<code>std::atomic&lt;unsigned char&gt;</code>
<code>atomic_int</code>	<code>std::atomic&lt;int&gt;</code>
<code>atomic_uint</code>	<code>std::atomic&lt;unsigned&gt;</code>
<code>atomic_short</code>	<code>std::atomic&lt;short&gt;</code>
<code>atomic_ushort</code>	<code>std::atomic&lt;unsigned short&gt;</code>
<code>atomic_long</code>	<code>std::atomic&lt;long&gt;</code>
<code>atomic_ulong</code>	<code>std::atomic&lt;unsigned long&gt;</code>
<code>atomic_llong</code>	<code>std::atomic&lt;long long&gt;</code>
<code>atomic_ullong</code>	<code>std::atomic&lt;unsigned long long&gt;</code>
<code>atomic_char16_t</code>	<code>std::atomic&lt;char16_t&gt;</code>
<code>atomic_char32_t</code>	<code>std::atomic&lt;char32_t&gt;</code>
<code>atomic_wchar_t</code>	<code>std::atomic&lt;wchar_t&gt;</code>

**Table 5.3 The operations available on atomic types**

Operation	atomic_flag	atomic<bool>	atomic<T*>	atomic<integral-type>	atomic<other-type>
test_and_set	✓				
clear	✓				
is_lock_free		✓	✓	✓	✓
load		✓	✓	✓	✓
store		✓	✓	✓	✓
exchange		✓	✓	✓	✓
compare_exchange_weak, compare_exchange_strong		✓	✓	✓	✓
fetch_add, +=			✓	✓	
fetch_sub, -=			✓	✓	
fetch_or,  =				✓	
fetch_and, &=				✓	
fetch_xor, ^=				✓	
++, --			✓	✓	

The C++ Standard Library also provides free functions for accessing instances of `std::shared_ptr<>` in an atomic fashion. This is a break from the principle that only the atomic types support atomic operations, because `std::shared_ptr<>` is quite definitely *not* an atomic type. However, the C++ Standards Committee felt it was sufficiently important to provide these extra functions. The atomic operations available are *load*, *store*, *exchange*, and *compare/exchange*, which are provided as overloads of the same operations on the standard atomic types, taking a `std::shared_ptr<>*` as the first argument:

```
std::shared_ptr<my_data> p;
void process_global_data()
{
    std::shared_ptr<my_data> local=std::atomic_load(&p);
    process_data(local);
}
void update_global_data()
{
    std::shared_ptr<my_data> local(new my_data);
    std::atomic_store(&p,local);
}
```



### Listing 5.2 Reading and writing variables from different threads

```
#include <vector>
#include <atomic>
#include <iostream>

std::vector<int> data;
std::atomic<bool> data_ready(false);

void reader_thread()
{
    while(!data_ready.load()) ← ❶
    {
        std::this_thread::sleep(std::milliseconds(1));
    }
    std::cout<<"The answer="<<data[0]<<"\n"; ← ❷
}

void writer_thread()
{
    data.push_back(42); ← ❸
    data_ready=true; ← ❹
}
```

Leaving aside the inefficiency of the loop waiting for the data to be ready ❶, you really need this to work, because otherwise sharing data between threads becomes impractical: every item of data is forced to be atomic. You've already learned that it's undefined behavior to have nonatomic reads ❷ and writes ❸ accessing the same data without an enforced ordering, so for this to work there must be an enforced ordering somewhere.

The required enforced ordering comes from the operations on the `std::atomic<bool>` variable `data_ready`; they provide the necessary ordering by virtue of the memory model relations *happens-before* and *synchronizes-with*. The write of the data ❸ happens-before the write to the `data_ready` flag ❹, and the read of the flag ❶ happens-before the read of the data ❷. When the value read from `data_ready` ❶ is true, the write synchronizes-with that read, creating a happens-before relationship. Because happens-before is transitive, the write to the data ❸ happens-before the write to the flag ❹, which happens-before the read of the true value from the flag ❶, which happens-before the read of the data ❷, and you have an enforced ordering: the write of the data happens-before the read of the data and everything is OK. Figure 5.2 shows the important happens-before relationships in the two threads. I've added a couple of iterations of the while loop from the reader thread.

The simplest possible thread pool

### Listing 9.1 Simple thread pool

```
class thread_pool
{
    std::atomic_bool done;
    thread_safe_queue<std::function<void()> > work_queue; ← 1
    std::vector<std::thread> threads; ← 2
    join_threads joiner; ← 3

    void worker_thread()
    {
        while(!done) ← 4
        {
            std::function<void()> task;
            if(work_queue.try_pop(task)) ← 5
            {
                task(); ← 6
            }
            else
            {
                std::this_thread::yield(); ← 7
            }
        }
    }

public:
    thread_pool():
        done(false), joiner(threads)
    {
        unsigned const thread_count=std::thread::hardware_concurrency(); ← 8

        try
        {
            for(unsigned i=0;i<thread_count;++i)
            {
                threads.push_back(
                    std::thread(&thread_pool::worker_thread,this)); ← 9
            }
        }
        catch(...)
        {
            done=true; ← 10
            throw;
        }
    }

    ~thread_pool()
    {
        done=true; ← 11
    }

    template<typename FunctionType>
    void submit(FunctionType f)
    {
        work_queue.push(std::function<void()>(f)); ← 12
    }
};
```

This implementation has a vector of worker threads ❷ and uses one of the thread-safe queues from chapter 6 ❶ to manage the queue of work. In this case, users can't wait for the tasks, and they can't return any values, so you can use `std::function<void()>` to encapsulate your tasks. The `submit()` function then wraps whatever function or callable object is supplied inside a `std::function<void()>` instance and pushes it on the queue ❷.

The threads are started in the constructor: you use `std::thread::hardware_concurrency()` to tell you how many concurrent threads the hardware can support ❸, and you create that many threads running your `worker_thread()` member function ❹.

Starting a thread can fail by throwing an exception, so you need to ensure that any threads you've already started are stopped and cleaned up nicely in this case. This is achieved with a `try-catch` block that sets the done flag when an exception is thrown ❿, alongside an instance of the `join_threads` class from chapter 8 ❸ to join all the threads. This also works with the destructor: you can just set the done flag ⓫, and the `join_threads` instance will ensure that all the threads have completed before the pool is destroyed. Note that the order of declaration of the members is important: both the done flag and the `worker_queue` must be declared before the `threads` vector, which must in turn be declared before the `joiner`. This ensures that the members are destroyed in the right order; you can't destroy the queue safely until all the threads have stopped, for example.

The `worker_thread` function itself is quite simple: it sits in a loop waiting until the done flag is set ❹, pulling tasks off the queue ❺ and executing them ❻ in the meantime. If there are no tasks on the queue, the function calls `std::this_thread::yield()` to take a small break ❼ and give another thread a chance to put some work on the queue before it tries to take some off again the next time around.

For many purposes such a simple thread pool will suffice, especially if the tasks are entirely independent and don't return any values or perform any blocking operations. But there are also many circumstances where such a simple thread pool may not adequately address your needs and yet others where it can cause problems such as deadlock. Also, in the simple cases you may well be better served using `std::async` as in many of the examples in chapter 8. Throughout this chapter, we'll look at more complex thread pool implementations that have additional features either to address user needs or reduce the potential for problems. First up: waiting for the tasks we've submitted.

Next time... threadsafe Cinder Kinect!



Since you have remapped the normals back onto a webcam image, it is a trivial matter to create a lightsource and dynamically change the lighting of your realtime webcam input. For the following video, I have created a virtual swinging light source above my head.

