C++ Concurrency in Action

Practical Multithreading

Anthony Williams

MEAP Unedited Draft

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1

Introduction

These are exciting times for C++ users. Eleven years after the original C++ Standard was published in 1998, the C++ Standards committee is giving the language and its supporting library a major overhaul. The new C++ Standard (referred to as C++0x) is due to be published in 2010 and will bring with it a whole swathe of changes that will make working with C++ easier and more productive.

One of the most significant new features in the C++0x Standard is the support of multi-threaded programs. For the first time, the C++ Standard will acknowledge the existence of multi-threaded applications in the language, and provide components in the library for writing multi-threaded applications. This will make it possible to write multi-threaded C++ programs without relying on platform-specific extensions, and thus allow us to write portable multi-threaded code with guaranteed behaviour. It also comes at a time when programmers are increasingly looking to concurrency in general, and multi-threaded programming in particular in order to improve application performance.

This book is about writing programs in C++ using multiple threads for concurrency, and the C++ language features and library facilities that make that possible. I'll start by explaining what I mean by concurrency and multi-threading, and why you would want to use it in your applications. After a quick detour into why you might *not* want to use it in your application, I'll give an overview of the concurrency support in C++, and round off this chapter with a simple example of C++ concurrency in action. Readers experienced with developing multi-threaded applications may wish to skip the early sections. In subsequent chapters we'll cover more extensive examples, and look at the library facilities in more depth. The book will finish with an in-depth reference to all the Standard C++ Library facilities for multi-threading and concurrency.

So, what do I mean by *concurrency* and *multi-threading*?

1.1 What is Concurrency?

At the simplest and most basic level, concurrency is about two or more separate activities happening at the same time. We encounter concurrency as a natural part of life: we can walk and talk at the same time or perform different actions with each hand, and of course we each go about our lives independently of each other — you can watch football whilst I go swimming, and so on.

1.1.1 Concurrency in Computer Systems

When we talk about concurrency in terms of computers, we mean a single system performing multiple independent activities in parallel, rather than sequentially one after the other. It is not a new phenomenon: multi-tasking operating systems that allow a single computer to run multiple applications at the same time through task switching have been common place for many years, and high-end server machines with multiple processors that enable genuine concurrency have been available for even longer. What *is* new is the increased prevalence of computers that can genuinely run multiple tasks in parallel rather than just giving the illusion of doing so.

Historically, most computers have had one processor, with a single processing unit or core, and this remains true for many desktop machines today. Such a machine can really only perform one task at a time, but they can switch between tasks many times per second. By doing a bit of one task and then a bit of another and so on, it appears that they are happening concurrently. This is called *task switching*. We still talk about *concurrency* with such systems: since the task switches are so fast, you can't tell at which point a task may be suspended as the processor switches to another one. The task switching provides an illusion of concurrency both to the user and the applications themselves. Since there is only an *illusion* of concurrency, the behaviour of applications may be subtly different when executing in a single-processor task-switching environment compared to when executing in an environment with true concurrency. In particular, incorrect assumptions about the memory model (covered in chapter 5) may not show up in such an environment. This is discussed in more depth in chapter 10.

Computers containing multiple processors have been used for servers and high-performance computing tasks for a number of years, and now computers based around processors with more than one core on a single chip (multi-core processors) are becoming increasingly common as desktop machines too. Whether they have multiple processors or multiple cores within a processor (or both), these computers are capable of genuinely running more than one task in parallel. We call this *hardware concurrency*.

Figure 1.1 shows an idealized scenario of a computer with precisely two task to do, each divided into ten equally-sized chunks. On a dual-core machine (which thus has two processing cores), each task can execute on its own core. On a single-core machine doing task-switching, the chunks from each task are interleaved. However, they are also spaced out a bit (in the diagram this is shown by the grey bars separating the chunks being thicker): in order to do the interleaving, the system has to perform a *context switch* every time it changes from one task to another, and this takes time. In order to perform a context switch the OS has to save the CPU state and instruction pointer for the currently running task, work out which task to switch to, and reload the CPU state for the task being switched to. The CPU will then potentially have to load the memory for the instructions and data for the new task into cache, which can prevent the CPU executing any instructions, thus causing further delay.

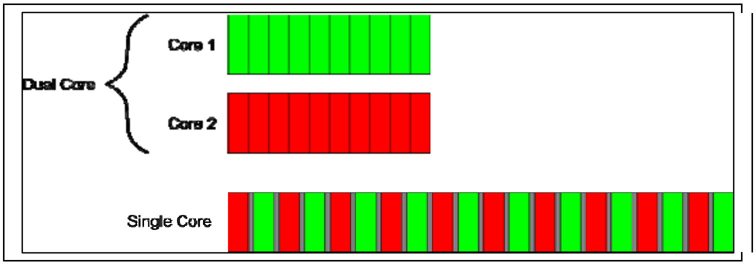


Figure 1.1 Two approaches to concurrency: parallel execution on a dual-core machine vs task-switching on a single core machine.

Though the availability of concurrency in the hardware is most obvious with multi-processor or multi-core systems, some processors can execute multiple threads on a single core. The important factor to consider is really the number of *hardware threads*: the measure of how many independent tasks the hardware can genuinely run concurrently. Even with a system that has genuine hardware concurrency, it is easy to have more tasks than the hardware can run in parallel, so task switching is still used in these cases. For example, on a typical desktop computer there may be hundreds of tasks running, performing background operations, even when the computer is nominally idle. It is the task-switching that allows these background tasks to run, and allows you to run your word processor, compiler, editor and web browser (or any combination of applications) all at once. Figure 1.2 shows task switching between four tasks on a dual-core machine, again for an idealized scenario with the tasks divided neatly into equal-sized chunks. In practice there are many issues which will make the divisions uneven and the scheduling irregular. Some of these are covered in chapter 8 when we look at factors affecting the performance of concurrent code.

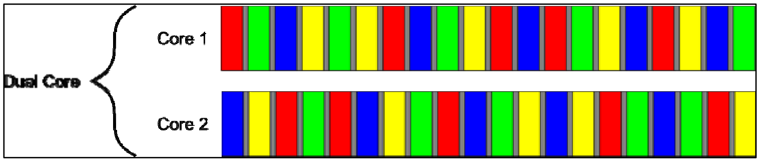


Figure 1.2 Task switching with two cores

All the techniques, functions and classes covered in this book can be used whether you're application is running on a machine with one single-core processor, or a machine with many multi-core processors, and are not affected by whether the concurrency is achieved through task switching or by genuine hardware concurrency. However, as you may imagine, how you make use of concurrency in your application may well depend on the amount of hardware concurrency available. This is covered in chapter 8, where I cover the issues involved with designing concurrent code in C++.

1.1.2 Approaches to Concurrency

Imagine for a moment a pair of programmers working together on a software project. If your developers are in separate offices, they can go about their work peacefully, without being disturbed by each other, and they each have their own set of reference manuals. However, communication is not straightforward: rather than just turning round and talking, they have to use the phone or email or get up and walk. Also, you've got the overhead of two offices to manage, and multiple copies of reference manuals to purchase.

Now imagine that you move your developers in to the same office. They can now talk to each other freely to discuss the design of the application, and can easily draw diagrams on paper or on a whiteboard to help with design ideas or explanations. You've now only got one office to manage, and one set of resources will often suffice. On the negative side, they might find it harder to concentrate, and there may be issues with sharing resources ("Where's the reference manual gone now?").

These two ways of organising your developers illustrate the two basic approaches to concurrency. Each developer represents a thread, and each office represents a process. The first approach is to have multiple single-threaded processes, which is similar to having each developer in his own office, and the second approach is to have multiple threads in a single process, which is like having two developers in the same room. You can of course combine these in an arbitrary fashion and have multiple processes, some of which are multi-threaded, and some of which are single-threaded, but the principles are the same. Let's now have a brief look at these two approaches to concurrency in an application.

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 which 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 interprocess communication channels (signals, sockets, files, pipes, etc.), as shown in figure 1.3. One downside is that such communication between processes is often either complicated to set up, slow, or both, since 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 is 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.

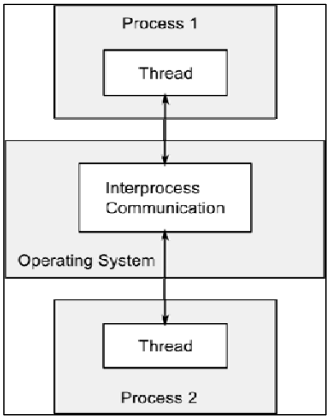


Figure 1.3 Communication between a pair of processes running concurrently

Of course, it's not all downside: the added protection operating systems typically provide between processes and the higher-level communication mechanisms mean that it can be easier to write *safe* concurrent code with processes rather than threads. Indeed, environments such as that provided for the Erlang programming language use processes as the fundamental building block of concurrency to great effect.

Using separate processes for concurrency also has an additional advantage — you can run the separate processes on distinct machines connected over a network. Though this increases the communication cost, on a carefully designed system it can be a very cost effective way of increasing the available parallelism, and improving performance.

Concurrency with Multiple Threads

The alternative approach to concurrency is to run multiple threads in a single process. Threads are very much like lightweight processes — each thread runs independently of the others, and each thread may run a different sequence of instructions. However, all threads in a process share the same address space, and the majority of data can be accessed directly from all threads — global variables remain global, and pointers or references to objects or data can be passed around between threads. Though it is often possible to share memory between processes, this is more complicated to set up, and often harder to manage, as memory addresses of the same data are not necessarily the same in different processes. Figure 1.4 shows two threads within a process communicating through shared memory.

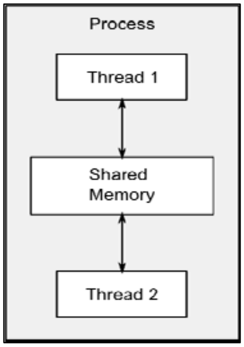


Figure 1.4 Communication between a pair of threads running concurrently in a single process

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, as the operating system has less book-keeping to do. However, the flexibility of shared memory also comes with a price — if data is accessed by multiple threads, then 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 the 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 in to the communication between threads.

The low overhead associated with launching and communicating between multiple threads within a process compared to launching and communicating between multiple single-threaded processes means that this is the favoured approach to concurrency in mainstream languages including C++, despite the potential problems arising from the shared memory. In addition, the C++ standard does not provide any intrinsic support for communication between processes, so applications that use multiple processes will have to rely on platform-specific APIs to do so. This book therefore focuses exclusively on using multi-threading for concurrency, and future references to concurrency are under the assumption that this is achieved by using multiple threads.

Having clarified what we mean by concurrency, let's now look at why we would use concurrency in our applications.

1.2 Why Use Concurrency?

There are two main reasons to use concurrency in an application: separation of concerns and performance. In fact, I'd go so far as to say they are the pretty much the *only* reasons to use concurrency: anything else boils down to one or the other (or maybe even both) when you look hard enough (well, except for reasons like “because I want to”).

1.2.1 Using Concurrency for Separation of Concerns

Separation of concerns is almost always a good idea when writing software: by grouping related bits of code together, and keeping unrelated bits of code apart we can make our programs easier to understand and test, and thus less likely to contain bugs. We can use concurrency to separate distinct areas of functionality even when the operations in these distinct areas need to happen at the same time: without the explicit use of concurrency we either have to write a task-switching framework, or actively make calls to unrelated areas of code during an operation.

Consider a processing-intensive application with a user-interface, such as a DVD player application for a desktop computer. Such an application fundamentally has two sets of responsibilities: not only does it have to read the data from the disk, decode the images and sound and send them to the graphics and sound hardware in a timely fashion so the DVD plays without glitches, but it must also take input from the user, such as when the user clicks “pause” or “return to menu”, or even “quit”. In a single thread, the application has to check for user input at regular intervals during the playback, thus conflating the DVD playback code with the user interface code. By using multi-threading to separate these concerns, the user interface code and DVD playback code no longer have to be so closely intertwined: one thread can handle the user interface, and another the DVD playback. Of course there will have to be interaction between them, such as when the user clicks “pause”, but now these interactions are directly related to the task at hand.

This gives the illusion of responsiveness, as the user-interface thread can typically respond immediately to a user request, even if the response is simply to display a “busy” cursor or “please wait” message whilst the request is conveyed to the thread doing the work.

Similarly, separate threads are often used to run tasks which must run continuously in the background, such as monitoring the filesystem for changes in a desktop search application. Using threads in this way generally makes the logic in each thread much simpler, as the interactions between them can be limited to clearly identifiable points, rather than having to intersperse the logic of the different tasks.

In this case, the number of threads is independent of the number of CPU cores available, since the division into threads is based on the conceptual design, rather than an attempt to increase throughput.

1.2.2 Using Concurrency for Performance

Multi-processor systems have existed for decades, but until recently they were mostly only found in supercomputers, mainframes and large server systems. However, chip manufacturers have increasingly been favouring multi-core designs with 2, 4, 16 or more processors on a single chip over better performance with a single core. Consequently, multi-core desktop computers, and even multi-core embedded devices, are now increasingly prevalent. The increased computing power of these machines comes not from running a single task faster, but from running multiple tasks in parallel. In the past, programmers have been able to sit back and watch their programs get faster with each new generation of processors, without any effort on their part, but now, as Herb Sutter put it: “The free lunch is over.” [Sutter2005] **If software is to take advantage of this increased computing power, it must be designed to run multiple tasks concurrently.** Programmers must therefore take heed, and those who have hitherto ignored concurrency must now look to add it to their toolbox.

There are two ways to use concurrency for performance. The first, and most obvious, is to divide a single task into parts, and run each in parallel, thus reducing the total runtime. This is *task parallelism*. Though this sounds straight-forward, it can be quite a complex process, as there may be many dependencies between the various parts. The divisions may be either in terms of processing — one thread performs one part of the algorithm, whilst another thread performs a different part — or in terms of data: each thread performs the same operation on different parts of the data. This latter is called *data parallelism*.

Algorithms which are readily susceptible to such parallelism are frequently called *Embarrassingly Parallel*. Despite the implications that you might be embarrassed to have code so easy to parallelize, this is a good thing: other terms I've encountered for such algorithms are *naturally parallel* and *conveniently concurrent.* Embarrassingly parallel algorithms have very good scalability properties — as the number of available hardware threads goes up, the parallelism in the algorithm can be increased to match. Such an algorithm is the perfect embodiment of “Many hands make light work”. For those parts of the algorithm that aren't embarrassingly parallel, you might be able to divide the algorithm into a fixed (and therefore not scalable) number of parallel tasks. Techniques for dividing tasks between threads are covered in chapter 8.

The second way to use concurrency for performance is to use the available parallelism to solve bigger problems — rather than processing one file at a time, process two or ten or twenty, as appropriate. Though this is really just an application of *data parallelism*, by performing the same operation on multiple sets of data concurrently, there's a different focus. It still takes the same amount of time to process one chunk of data, but now more data can be processed in the same amount of time. Obviously, there are limits to this approach too, and this will not be beneficial in all cases, but the increase in throughput that comes from such an approach can actually make new things possible — increased resolution in video processing, for example, if different areas of the picture can be processed in parallel.

1.2.3 When Not to use Concurrency

It is just as important to know when *not* to use concurrency as it is to know when *to* use it. Fundamentally, the one and only reason not to use concurrency is when the benefit is not worth the cost. Code using concurrency is harder to understand in many cases, so there is a direct intellectual cost to writing and maintaining multi-threaded code, and the additional complexity can also lead to more bugs. Unless the potential performance gain is large enough or separation of concerns clear enough to justify the additional development time required to get it right, and the additional costs associated with maintaining multi-threaded code, don't use concurrency.

Also, the performance gain might not be as large as expected: there is an inherent overhead associated with launching a thread, as the OS has to allocate the associated kernel resources and stack space, and then add the new thread to the scheduler, all of which takes time. If the task being run on the thread is completed quickly, then the actual time taken by the task may be dwarfed by the overhead of launching the thread, possibly making the overall performance of the application worse than if the task had been executed directly by the spawning thread.

Furthermore, threads are a limited resource. If you have too many threads running at once, this consumes OS resources, and may make the system as a whole run slower. Not only that, but using too many threads can exhaust the available memory or address space for a process, since each thread requires a separate stack space. This is particularly a problem for 32-bit processes with a “flat” architecture where there is a 4Gb limit in the available address space: if each thread has a 1Mb stack (as is typical on many systems), then the address space would be all used up with 4096 threads, without allowing for any space for code or static data or heap data. Though 64-bit (or larger) systems don't have this direct address-space limit, they still have finite resources: if you run too many threads this will eventually cause problems. Though thread pools (see chapter 9) can be used to limit the number of threads, these are not a silver bullet, and they do have their own issues.

If the server side of a client-server application launched a separate thread for each connection, this works fine for a small number of connections, but can quickly exhaust system resources by launching too many threads if the same technique is used for a high-demand server which has to handle many connections. In this scenario, careful use of thread pools can provide optimal performance (see chapter 9).

Finally, the more threads you have running, the more context switching the operating system has to do. Each context switch takes time that could be spent doing useful work, so at some point adding an extra thread will actually *reduce* the overall application performance rather than increase it. For this reason, if you are trying to achieve the best possible performance of the system, it is necessary to adjust the number of threads running to take account of the available hardware concurrency (or lack of it).

Use of concurrency for performance is just like any other optimization strategy — it has potential to greatly improve the performance of your application, but it can also complicate the code, making it harder to understand, and more prone to bugs. Therefore it is only worth doing for those performance-critical parts of the application where there is the potential for measurable gain. Of course, if the potential for performance gains is only secondary to clarity of design or separation of concerns then it may still be worth using a multi-threaded design.

Assuming that you've decided you *do* want to use concurrency in your application, whether for performance, separation of concerns or because it's “multi-threading Monday”, what does that mean for us C++ programmers?

1.3 Concurrency and Multi-threading in C++

Standardized support for concurrency through multi-threading is a new thing for C++. It is only with the upcoming C++0x standard that you will be able to write multi-threaded code without resorting to platform-specific extensions. In order to understand the rationale behind lots of the decisions in the new Standard C++ thread library, it's important to understand the history.

1.3.1 History of multi-threading in C++

The 1998 C++ Standard does not acknowledge the existence of threads, and the operational effects of the various language elements are written in terms of a sequential abstract machine. Not only that, but the memory model is not formally defined, so you can't write multi-threaded applications without compiler-specific extensions to the 1998 C++ Standard.

Of course, compiler vendors are free to add extensions to the language, and the prevalence of C APIs for multi-threading — such as those in the POSIX C Standard and the

Microsoft Windows API — has led many C++ compiler vendors to support multi-threading with various platform specific extensions. This compiler support is generally limited to allowing the use of the corresponding C API for the platform, and ensuring that the C++ runtime library (such as the code for the exception handling mechanism) works in the presence of multiple threads. Though very few compiler vendors have provided a formal multi-threading-aware memory model, the actual behaviour of the compilers and processors has been sufficiently good that a large number of multi-threaded C++ programs have been written.

Not content with using the platform-specific C APIs for handling multi-threading, C++ programmers have looked to their class libraries to provide object-oriented multi-threading facilities. Application frameworks such as MFC, and general-purpose C++ libraries such as Boost and ACE have accumulated sets of C++ classes that wrap the underlying platform-specific APIs and provide higher-level facilities for multi-threading that simplify the tasks. Though the precise details of the class libraries have varied considerably, particularly in the area of launching new threads, the overall shape of the classes has had a lot in common. One particularly important design that is common to many C++ class libraries, and which provides considerable benefit to the programmer, has been the use of the *Resource Acquisition Is Initialization* (RAII) idiom with locks to ensure that mutexes are unlocked when the relevant scope is exited.

For many cases, the multi-threading support of existing C++ compilers, combined with the availability of platform-specific APIs and platform-independent class libraries such as Boost and ACE provides a good solid foundation on which to write multi-threaded C++ code, and as a result there are probably millions of lines of C++ code written as part of multi-threaded applications. However, the lack of Standard support means that there are occasions where the lack of a thread-aware memory model causes problems, particularly for those who try to gain higher performance by using knowledge of the processor hardware, or for those writing cross-platform code where the actual behaviour of the compilers varies between platforms.

1.3.2 Concurrency Support in the New Standard

All this changes with the release of the new C++0x Standard. Not only is there a brand new thread-aware memory model, but the C++ Standard library has been extended to include classes for managing threads (see chapter 2), protecting shared data (see chapter 3), synchronizing operations between threads (see chapter 4) and low-level atomic operations (see chapter 5).

The new C++ thread library is heavily based on the prior experience accumulated through the use of the C++ class libraries mentioned above. In particular, the Boost thread library has been used as the primary model on which the new library is based, with many of the classes sharing their names and structure with the corresponding ones from Boost. As the new Standard has evolved, this has been a two-way flow, and the Boost thread library has itself changed to match the C++ Standard in many respects, so users transitioning from Boost should find themselves very much at home.

Concurrency support is just one of the changes with the new C++ Standard — as mentioned at the beginning of this chapter, there are many enhancements to the language itself to make programmers' lives easier. Though these are generally outside the scope of this book, some of those changes have actually had a direct impact on the thread library itself, and the ways in which it can be used. Appendix A provides a brief introduction to these language features.

The support for atomic operations directly in C++ enables programmers to write efficient code with defined semantics without the need for platform-specific assembly language. This is a real boon for those of us trying to write efficient, portable code: not only does the compiler take care of the platform specifics, but the optimizer can be written to take into account the semantics of the operations, thus enabling better optimization of the program as a whole.

1.3.3 Efficiency in the C++ Thread Library

One of the concerns that developers involved in high-performance computing often raise regarding C++ in general, and C++ classes that wrap low-level facilities, such as those in the new Standard C++ Thread Library specifically, is that of efficiency. If you're after the utmost in performance, then it is important to understand the implementation costs associated with using any high-level facilities, compared to using the underlying low-level facilities directly. This cost is the *Abstraction Penalty*.

The C++ Standards committee has been very aware of this when designing the Standard C++ Library in general, and the Standard C++ Thread Library in particular — one of the design goals has been that there should be little or no benefit to be gained from using the lower-level APIs directly, where the same facility is to be provided. The library has therefore been designed to allow for efficient implementation (with a very low abstraction penalty) on most major platforms.

Another goal of the C++ Standards committee has been to ensure that C++ provides sufficient low-level facilities for those wishing to work close to the metal for the ultimate performance. To this end, along with the new memory model comes a comprehensive atomic operations library for direct control over individual bits and bytes, and the inter-thread synchronization and visibility of any changes. These atomic types, and the corresponding operations can now be used in many places where developers would previously have chosen to drop down to platform-specific assembly language. Code using the new standard types and operations is thus more portable and easier to maintain.

The Standard C++ Library also provides higher level abstractions and facilities that make writing multi-threaded code easier and less error-prone. Sometimes the use of these facilities does comes with a performance cost due to the additional code that must be executed. However, this performance cost does not necessarily imply a higher abstraction penalty though: in general the cost is no higher than would be incurred by writing equivalent functionality by hand, and the compiler may well inline much of the additional code anyway.

In some cases, the high level facilities provide additional functionality beyond what may be required for a specific use. Most of the time this is not an issue: you don't pay for what you don't use. On rare occasions this unused functionality will impact the performance of other code. If you are aiming for performance, and the cost is too high, you may be better off hand-crafting the desired functionality from lower-level facilities. In the vast majority of cases, the additional complexity and chance of errors far outweighs the potential benefits from a small performance gain. Even if profiling *does* demonstrate that the bottleneck is in the C++ Standard Library facilities, it may be due to poor application design rather than a poor library implementation. For example, if too many threads are competing for a mutex it *will* impact the performance significantly. Rather than trying to shave a small fraction of time off the mutex operations, it would probably be more beneficial to restructure the application so that there was less contention on the mutex. This sort of issue is covered in chapter 8.

In those very rare cases where the C++ Standard Library does not provide the performance or behaviour required, it might be necessary to use platform specific facilities.

1.3.4 Platform-Specific Facilities

Whilst the C++ Thread Library provides reasonably comprehensive facilities for multi-threading and concurrency, on any given platform there will be platform-specific facilities that go beyond what is offered. In order to gain easy access to those facilities without giving up the benefits of using the Standard C++ thread library, the types in the C++ Thread Library may offer a **native_handle()** member function which allows the underlying implementation to be directly manipulated using a platform-specific API. By its very nature, any operations performed using the **native_handle()** are entirely platform dependent, and out of the scope of this book (and the C++ Standard Library itself).

Of course, before even considering using platform-specific facilities, it's important to understand what the Standard library provides, so let's get started with an example.

1.4 Getting Started

OK, so you've got a nice shiny C++09-compatible compiler. What next? What does a multi-threaded C++ program look like? It looks pretty much like any other C++ program, with the usual mix of variables, classes and functions. The only real distinction is that some functions might be running concurrently, so care needs to be taken to ensure that shared data is safe for concurrent access, as described in chapter 3. Of course, in order to run functions concurrently, specific functions and objects must be used in order to manage the different threads.

1.4.1 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 below, to serve as our 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” shown in listing 1.1, 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
}
```

Cueballs in Code and Text

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>`**, whilst those for protecting shared data are declared in other headers.

Secondly, the code for writing the message has been moved to a separate function (#2). This is because every thread has to have an *initial function*, which is where the new thread of execution begins. For the initial thread in an application, this is **main()**, but for every other thread it is specified in the constructor of a **std::thread** object — in this case, the **std::thread** object named **t** (#3) 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 (#3), 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 (#4) — 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**.

If this seems like a lot of work to go to just to write a message to standard output, it is — as described in section 1.2.3 above, it is generally not worth the effort to use multiple threads for such a simple task, especially if the initial thread has nothing to do in the mean time. Later in the book, we will work through examples which show scenarios where there is a clear gain to using multiple threads.

1.5 Summary

In this chapter, we've covered what is meant by concurrency and multi-threading, and why we would choose to use it (or not) in our applications. We've also covered the history of multi-threading in C++ from the complete lack of support in the 1998 Standard, through various platform-specific extensions to proper multi-threading support in the new C++ Standard, C++0x. This support is coming just in time to allow programmers to take advantage of the greater hardware concurrency becoming available with newer CPUs, as chip manufacturers choose add more processing power in the form of multiple cores which allow more tasks to be executed concurrently, rather than increasing the execution speed of a single core.

We've also seen quite how simple to use the classes and functions from the C++ Standard Library can be, in the examples from section 1.4. In C++, using multiple threads is not complicated in and of itself — the complexity lies in designing the code so that it behaves as intended.

After the taster examples of section 1.4, it's time for something with a bit more substance. In chapter 2 we'll look at the classes and functions available for managing threads.

2

Managing Threads

OK, so you've decided to use concurrency for your application. In particular, you've decided to use multiple threads. What now? How do you launch these threads, how do you check that they've finished, and how do you keep tabs on them? The C++ Standard Library makes most thread management tasks relatively easy, with just about everything managed through the **`std::thread`** object associated with a given thread, as you'll see. For those tasks that aren't so straightforward, the library provides the flexibility to build what you need from the basic building blocks.

In this chapter, we'll start by covering the basics: launching a thread, waiting for it to finish, or running it in the background. We'll then proceed to look at passing additional parameters to the thread function when it is launched, and how to transfer ownership of a thread from one **`std::thread`** object to another. Finally, we'll look at choosing the number of threads to use, and identifying particular threads.

2.1 Basic Thread Management

Every C++ program has at least one thread, which is started by the C++ runtime: the thread running **`main().`** **Y**our program can then launch additional threads which have another function as the entry point. These threads then run concurrently with each other and with the initial thread. Just as the program exits when the program returns from **`main()`**, when the specified entry point function returns, the thread is finished. As we'll see, if you have a **`std::thread`** object for a thread, you can wait for it to finish, but first we have to start it so let's look at launching threads.

2.1.1 Launching a Thread

As we 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 whilst it is running, and the thread only stops when it is signalled 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, as with much of the Standard C++ library, 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 is therefore essential that the copy behaves equivalently to the original, or the result may not be what is expected.

Since the callable object supplied to the constructor is copied into the thread, the original object can be destroyed immediately. However, if the object contains any pointers or references, it is important to ensure that those pointers and references remain valid as long as they may be accessed from the new thread, otherwise undefined behaviour will result. In particular, it is 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. Listing 2.1 shows an example of a just such a problematic function.

Listing 2.1: A function that returns whilst a thread still has access to local variables

```
    struct func
    {
        int& i;

        func(int& i_):i(i_){}

        void operator()()
        {
            for(unsigned j=0;j<1000;++j)
            {
                do_something(i);           #1
            }
        }
    };


    void oops()
    {
        int some_local_state=0;
        std::thread my_thread(func(some_local_state));
    }                                          #2
```

Cueballs in code and text

#1 Potential access to dangling reference
#2 The new thread might still be running

In this case, the new thread associated with **my_thread** will probably still be running when **oops** exits (#2), in which case 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 is easier to make the mistake with multi-threaded code, as it is not necessarily immediately apparent that this has happened. In cases like this, it is desirable to ensure that the thread has completed execution before the function exits.

2.1.2 Waiting for a Thread to Complete

If you need to wait for a thread to complete, this can be done by calling **join()** on the associated **std::thread** instance. In the case of listing 2.1, inserting a call to **my_thread.join()** before the closing brace of the function body would therefore be sufficient to ensure that the thread was finished before the function was exited, and thus before the local variables were destroyed. In this case, it would mean there was little point running the function on a separate thread, as the first thread would not be doing anything useful in the mean time, but in real code then the original thread would either have work to do itself, or it would have launched several threads to do useful work before waiting for all of them to complete.

join() is very simple, and brute-force — either you wait for a thread to finish, or you don't. If you need more fine-grained control over waiting for a thread, such as just to check whether a thread is finished, or wait only a certain period of time, then you have to use alternative mechanisms. The act of calling **join()** also cleans up any storage associated with the thread, so the **std::thread** object is no longer associated with the now-finished thread — it is not associated with any thread.

Listing 2.2: Waiting for a thread to finish

```
struct func;                          #A

void f()
{
    int some_local_state=0;
    std::thread t(func(some_local_state));
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();                      #2
        throw;
    }
    t.join();                          #1
}
```

Cueballs in code and text

#A See definition in listing 2.1

Listing 2.2 shows code to ensure that a thread with access to local state is finished before the function exits, whether the function exits normally (#1) or by an exception (#2). Just as it is important to ensure that any other locally allocated resources are properly cleaned up on function exit, local threads are no exception — if the thread must complete before the function exits, whether because it has a reference to other local variables, or for any other reason, then it is important to ensure this is the case for all possible exit paths, whether normal or exceptional. One way of doing this is to use the standard *Resource Acquisition Is Initialization* idiom (RAII), and provide a class that does the **join()** in its destructor, as in listing 2.3. See how it simplifies the function **f()**.

Listing 2.3: Using RAII to wait for a thread to complete

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):
        t(t_)
    {}
```

```
        ~thread_guard()
        {
            if(t.joinable())                    #2
            {
                t.join();                       #3
            }
        }
        thread_guard(thread_guard const&)=delete;#4
        thread_guard& operator=(
             thread_guard const&)=delete;
    };

    void func(int&);

    void f()
    {
        int some_local_state;
        std::thread t(func(some_local_state));
        thread_guard g(t);

        do_something_in_current_thread();
    }                                           #1
```

Cueballs in code and text

When the execution of the current thread reaches the end of **f** (#1), the local objects are destroyed in reverse order of construction. Consequently, the **thread_guard** object **g** is destroyed first, and the thread joined with in the destructor (#3). 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()** (#2) before calling **join()** (#3). This is important, because **join()** can only be called once for a given thread of execution, so it would therefore be a mistake to do so if the thread had already been joined with.

The copy constructor and copy-assignment operator are marked **=delete** (#4) to ensure that they are not automatically provided by the compiler: copying or assigning such an object would be dangerous, as it might then outlive the scope of the thread it was joining.

The reason we have to take such precautions to ensure that our threads are joined when they reference local variables is because a thread can continue running even when the **std::thread** object that was managing it has been destroyed. Such a thread is said to be *detached* — it is no longer attached to a **std::thread** object. This means that the C++ runtime library is now responsible for cleaning up the resources associated with the thread when it exits, rather than that being the responsibility of the **std::thread** object. It is also no longer possible to wait for that thread to complete — once a thread becomes detached it is not possible to obtain a **std::thread** object that references it, so it can no longer be joined with. Detached threads truly run in the background: ownership and control is passed over to the C++ runtime library.

2.1.3 Running Threads in the Background

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 may well run for almost the entire lifetime of the application, performing a background task such as monitoring the file system, 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 is another mechanism for identifying when the thread has completed, or where the thread is used for a “fire and forget” task.

As we've already seen in section 2.1.2, one way to detach a thread is just to destroy the associated **std::thread** object. This is fine for those circumstances where you *can* destroy the **std::thread** object, either because it is a local object and is destroyed when the containing scope is exited, or because it was allocated dynamically either directly with **new**, or as part of a container. If the **std::thread** object cannot be destroyed at the point in code where you wish to detach the thread, you can do so 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());
```

The thread of execution no longer has an associated management object, just as if the **std::thread** object had been destroyed. The C++ runtime library is therefore responsible for cleaning up the resources associated with running the thread when it completes.

Even if the **std::thread** object for a thread is to be destroyed at this point in the code, sometimes it is worth calling **detach()** to be explicit in your intent: it makes it clear to whoever maintains the code that this thread was intended to be detached. Given that multi-threaded code can be quite complex, anything that makes it easier to understand should be considered.

Of course, in order to detach the thread from a **std::thread** object, there must be a thread to detach: you cannot call **detach()** on a **std::thread** object with no associated thread of execution. This is exactly the same requirement as **join()**, and you can therefore 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 does seem to be increasingly common at the moment is to have multiple independent top-level windows, one for each document being edited. Though these windows appear to be completely independent, each with their own menus and so forth, they are 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 is not going to care about waiting for that other thread to finish, as it is working on an unrelated document, so this makes it a prime case for running a detached thread.

Listing 2.4 shows a simple code outline for this approach: if the user chooses to open a new document, we prompt them for the document to open, then start a new thread to open that document (#1), and detach it (#2). Since the new thread is doing the same operation as the current thread, but on a different file, we can reuse the same function (**edit_document**), but with the newly-chosen filename as the supplied argument.

Listing 2.4: Detaching 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,  #1
                          new_name);
            t.detach();                   #2
        }
        else
        {
            process_user_input(cmd);
        }
    }
}
```

Cueballs in code and text

This example also shows a case where it is 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 (#1), we also pass in the filename parameter. Though 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 thread library provides us with an easy way of doing it.

2.2 Passing Arguments to a Thread Function

As seen 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. However, it is 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, even if the corresponding parameter in the function is expecting a reference. Here's a simple 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 only converted to a **std::string** in the context of the new thread. This is particularly important when the argument supplied is a pointer to an automatic variable, as below:

```
    void f(int i,std::string const& s);

    void oops(int some_param)
    {
        char buffer[1024];                  #1
        sprintf(buffer,"%i",some_param);
        std::thread(f,3,buffer);            #2
    }
```

Cueballs in code and text

In this case, it is the pointer to the local variable **buffer** (#1) that is passed through to the new thread (#2), and there's a significant chance that the function **oops** will exit before the buffer has been converted to a **std::string** on the new thread, thus leading to undefined behaviour. 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(f,3,std::string(buffer));  #1
    }
```

#1 Convert to std::string explicitly to avoid dangling pointer

In this case, the problem was that we were relying on the implicit conversion of the pointer to the buffer into the **std::string** object expected as a function parameter, as the **std::thread** constructor copies the supplied values as-is, without converting to the expected argument type.

It's also possible to get the reverse scenario: the object is copied, and what you wanted was a reference. This might happen if the thread is updating a data structure which is passed in by reference, for example:

```
void update_data_for_widget(widget_id w,
    widget_data& data);                   #1

void oops_again(widget_id w)
{
    widget_data data;
    std::thread t(update_data_for_widget,
        w,data);                          #2
    display_status();
    t.join();
    process_widget_data(data);            #3
}
```

Cueballs in code and text

Though **update_data_for_widget** (#1) expects the second parameter to be passed by reference, the **std::thread** constructor (#2) doesn't know that: it is oblivious to the types of the arguments expected by the function and just blindly copies the supplied values. When it calls **update_data_for_widget**, it will end up passing a reference to the internal copy of **data**, and not a reference to **data** itself. Consequently, when the thread finishes these updates will be discarded as the internal copies of the supplied arguments are destroyed, and **process_widget_data** will be passed an unchanged **data** (#3) rather than a correctly-updated version. For those of you familiar with **std::bind**, the solution will be readily apparent: you need to wrap the arguments that really need to be references in **std::ref**. In this case, if we 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 reference to a *copy* of **data**.

If you're familiar with **std::bind**, the whole parameter-passing semantics will be unsurprising, since 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);   #1
```

Cueballs in code and text

This code will invoke **my_x.do_lengthy_work()** on the new thread, since the address of **my_x** is supplied as the object pointer (#1). 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 cannot 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. 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 a temporary, the move is automatic, but where the source is a named value the transfer must be requested directly by invoking **std::move()**. The example below 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 the **big_object** is transferred first into internal storage for the newly-created thread, and then into **process_big_object** in turn.

Several of the classes in the Standard thread 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, since 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, whilst 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 back ownership of the new thread to the calling function rather than wait 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 example below. The example shows the creation of two threads of execution, and the transfer of ownership of those threads between three **std::thread** instances, **t1**, **t2**, and **t3**.

```
void some_function();
void some_other_function();
std::thread t1(some_function);              #1
std::thread t2=std::move(t1);               #2
t1=std::thread(some_other_function);        #3
std::thread t3;                             #4
t3=std::move(t2);                           #5
t1=std::move(t3);                           #6
```

Cueballs in code and text

Firstly, a new thread is started (#1), and associated with **t1**. Ownership is then transferred over to **t2** when **t2** is constructed, by invoking **std::move()** to explicitly move ownership (#2). 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 (#3). The subsequent transfer of ownership into **t1** doesn't require a call to **std::move()** to explicitly move ownership, since the owner is a temporary object — moving from temporaries is automatic and implicit.

t3 is default-constructed (#4), which means that it is created without any associated thread of execution. Ownership of the thread currently associated with **t2** is transferred into **t3** (#5), again with an explicit call to **std::move()**, since **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 (#6) transfers ownership of the thread running **some_function** back to **t1** where it started. However, in this case **t1** already had an associated thread (which was running **some_other_function**), so that thread is therefore detached as it no longer has an associated **std::thread** object.

The move support **in std::thread** means that ownership can readily be transferred out of a function, as shown in listing 2.5.

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 just 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 we can build on the **thread_guard** class from listing 2.3 and have it actually 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. Since this would primarily be aimed at ensuring threads are completed before a scope is exited, I named this class **scoped_thread**. The implementation is shown in listing 2.6, 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_):    #4
        t(std::move(t_))
    {
        if(!t.joinable())                      #5
            throw std::logic_error(“No thread”);
    }
    ~scoped_thread()
    {
        t.join();                              #3
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(
         scoped_thread const&)=delete;
};

void func(int&);

void f()
{
    int some_local_state;
    scoped_thread t(                           #1
        std::thread(func(some_local_state)));

    do_something_in_current_thread();
}                                              #2
```

The example is very similar to that from listing 2.3, but the new thread is passed in directly to the **scoped_thread** (#1), rather than having to create a separate named variable for it. When the initial thread reaches the end of **f** (#2), the **scoped_thread** object is destroyed and then joins with (#3) the thread supplied to the constructor (#4). Whereas with the **thread_guard** class from listing 2.3, the destructor had to check the thread was still joinable, we can do that in the constructor (#5), and throw an exception if not.

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 in listing 2.7, 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));        #1
        }
        std::for_each(threads.begin(),threads.end(),
            std::mem_fn(&std::thread::join));   #2
    }
```

#1 Spawn threads
#2 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 is 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 towards 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. We 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 2.7.

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 multi-core 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 is not available, but it can be a useful guide for splitting a task between threads.

Listing 2.8 shows a simple implementation of parallel version of **`std::accumulate`**. It divides the work between 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 cannot 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 2.8: A 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]));
        block_start=block_end;                                        #8
    }
    accumulate_block()(block_start,last,results[num_threads-1]);      #9

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

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

Cueballs in code and text

Though this is quite a long function, it's actually really straightforward. If the input range is empty (#1), we just return the initial value **init**. Otherwise, there's at least one element in the range, so we can divide the number of elements to process by the minimum block size, in order to give the maximum number of threads (#2). This is to avoid us creating 32 threads on a 32-core machine when we've only got 5 values in the range.

The number of threads to run is the minimum of our calculated maximum, and the number of hardware threads (#3): we don't want to run more threads than the hardware can support, as the context switching will mean that more threads will decrease the performance. If the call to **std::hardware_concurrency()** returned 0, we simply substitute a number of our choice: in this case I've chosen 2. We don't want to run too many threads, as that would slow things down on a single-core machine, but likewise we don't want to run too few as then we're passing up the available concurrency.

The number of entries for each thread to process is simply the length of the range divided by the number of threads (#4). If you're worrying about the case where the number doesn't divide evenly, don't — we'll handle that later.

Now we know how many threads we've got we can create a **std::vector<T>** for the intermediate results, and a **std::vector<std::thread>** for the threads (#5). Note that we need to launch one fewer thread than **num_threads**, since we've already got one.

Launching the threads is just a simple loop: advance the **block_end** iterator to the end of the current block (#6), and launch a new thread to accumulate the results for this block (#7). The start of the next block is just the end of this one (#8).

After we've launched all the threads, this thread can then process the final block (#9). This is where we take account of any uneven division: we know the end of the final block must be **last**, and it doesn't matter how many elements are in that block.

Once we've accumulated the results for the last block, we can wait for all the threads we spawned with **std::for_each** (#10), like in listing 2.8, and then add up the results with a final call to **std::accumulate** (#11).

Before we leave this example, it's worth pointing out that where the addition operator for the type **T** is not associative (such as for **float** or **double**), the results of this **parallel_accumulate** may vary from those of **std::accumulate**, due to 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 we can create the **results** vector. These sorts of requirement changes are common with parallel algorithms: by their very nature they are different in some manner in order to make them parallel, and this has consequences on the results and requirements. Parallel algorithms are covered in more depth in chapter 8.

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 is not always the case: sometimes it is necessary to be able to identify the threads in some way for part of the processing. You could of course 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 is inconvenient to have to do it that way. When we were designing the C++ thread library we foresaw this need, and 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. Firstly, 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()**.

Objects of type **std::thread::id** can be freely copied around and compared: they wouldn't be much use as identifiers otherwise. If two objects of type **std::thread::id** are equal then they represent the same thread, or both are holding the “not any thread” value. If two objects are not equal then 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 or not 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 would 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 or not a thread needs to perform some operation. For example, if threads are used to divide work as in listing 2.8 then 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 there is specific data that needs associating with a thread, and alternative mechanisms such as thread-local storage are not 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 is 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 equal should produce the same output, and those that are not equal should give different output. This is therefore primarily useful for debugging and logging, but the values have no semantic meaning, so there is not much more that could be said anyway.

2.6 Summary

In this chapter we've 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 we want them to run in the background. We've also seen how to pass arguments into the thread function when a thread is started, and 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've discussed identifying threads in order to associate data or behaviour with specific threads that is inconvenient to associate through alternative means. Though you can do quite a lot with purely independent threads that each operate on separate data, as in listing 2.8 for example, sometimes it is desirable to share data between threads whilst they are running. Chapter 3 discusses the issues surrounding sharing data directly between threads, whilst chapter 4 covers more general issues surrounding synchronizing operations with and without shared data.

3

Sharing Data between Threads

One of the key benefits of using threads for concurrency is the possibility of easily and directly sharing data between them, so now we've covered starting and managing threads, let's look at the issues surrounding shared data.

Imagine for a moment that you're sharing a flat with a friend. There's only one kitchen and only one bathroom. Unless you're particularly friendly, you can't both use the bathroom at the same time, and if your flatmate occupies the bathroom for a long time, it can be frustrating if you need to use it. Likewise, though it might be possible to both cook meals at the same time, if you've got a combined oven and grill, it's just not going to end well if one of you tries to grill some sausages at the same time as the other is baking cakes. Furthermore, we all know the frustration of sharing a space and getting half-way through a task only to find that someone has borrowed something you need, or changed something from how you had it.

It's the same with threads. If you're sharing data between threads, you need to have rules for which thread can access which bit of data when, and how any updates are communicated to the other threads that care about that data. The ease with which data can be shared between multiple threads in a single process is not just a benefit — it can also be a big drawback too. Incorrect use of shared data is one of the biggest causes of concurrency-related bugs, and the consequences can be far worse than sausage-flavoured cakes.

This chapter is about sharing data safely between threads in C++, avoiding the potential problems that can arise, and maximizing the benefits.

3.1 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 is no problem, since 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 is a lot of potential for trouble. In this case, you must take care to ensure that everything works out OK.

One concept that is 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), and 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 either side have to be updated to point to each other. Once one has been updated, the invariant is broken until the node 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 back from the node after N to point to the node prior to N.
4. Delete the node N.

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

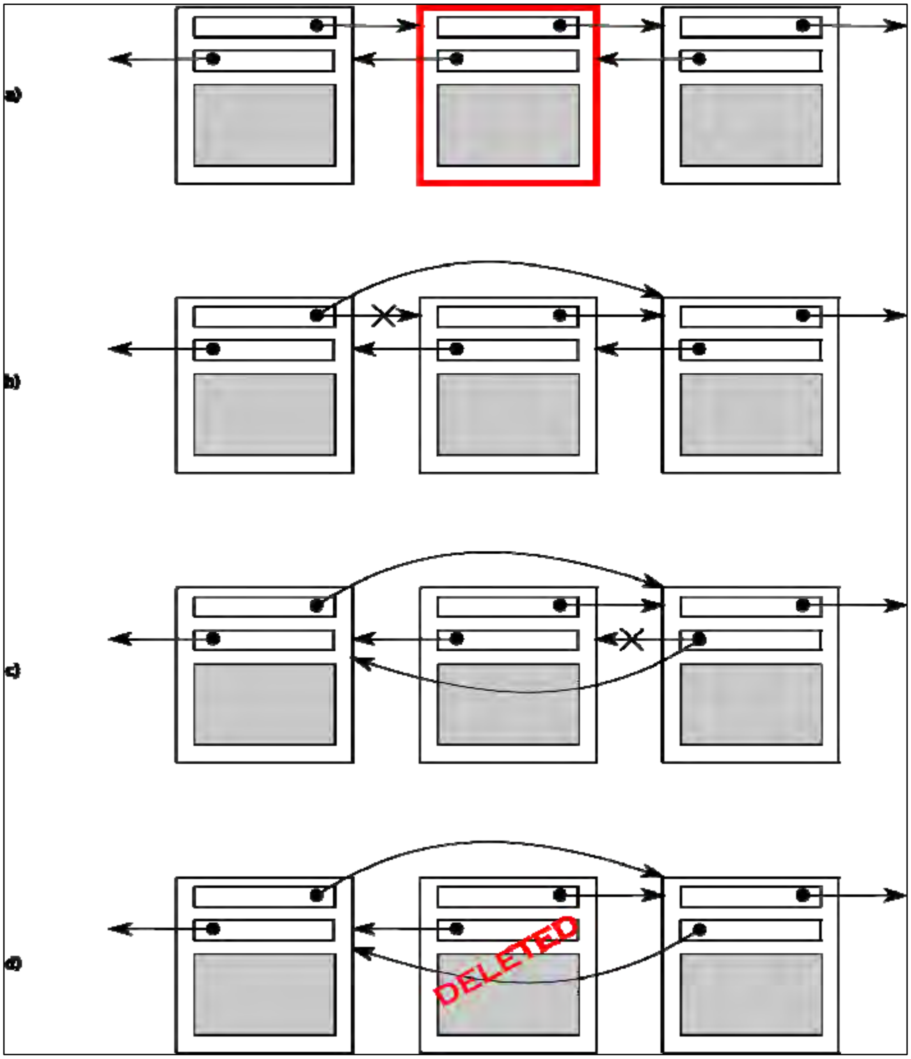


Figure 3.5 Deleting a node from a doubly-linked list

The simplest potential problem with modifying data that is shared between threads is that of broken invariants. If you don't do anything special to ensure otherwise, if one thread is reading the doubly-linked list whilst another is removing a node, it is quite possible for the reading thread to see the list with a node only partially removed (because only one of the links has been changed, as in step b) of figure 3.1), so the invariant is broken. The consequences of this broken invariant can vary — if the other thread is just reading the list items from left to right in the diagram, it will just skip the node being deleted. On the other hand, if the second thread is trying to delete the right-most node in the diagram, it might end up permanently corrupting the data structure, and eventually crashing the program. Whatever the outcome, this is an example of one of the most common causes of bugs in concurrent code: a *race condition*.

3.1.1 Race Conditions

Suppose you're buying tickets to see a film at the cinema. If it's a big cinema, there will be multiple cashiers taking money, so more than one person can buy tickets at the same time. If someone at another cashier's desk is also buying tickets for the same film as you, which seats are available for you to choose from depends on whether the other person actually books first, or you do. If there's only a few seats left, this difference can be quite crucial: it might literally be a race to see who gets the last tickets. This is an example of a *race condition*: which seats you get (or even whether or not you get tickets) depends on the relative ordering of the two purchases.

In concurrency, a race condition is anything where the outcome depends on the relative ordering of execution of operations on two or more threads — the threads race to perform their respective operations. Most of the time, this is quite benign because all possible outcomes are acceptable, even though they may change with different relative orderings. For example, if two threads are adding items to a queue for processing, it generally doesn't matter which item gets added first, provided that the invariants of the system are maintained. It is when the race condition leads to broken invariants that there is a problem, such as with the doubly-linked list example above. When talking about concurrency, the term *race condition* is usually used to mean a *problematic* race condition — benign race conditions are not so interesting, and aren't a cause of bugs. The C++ Standard also defines the term *data race* to mean the specific type of race condition that arises due to concurrent modification to a single object (see section 5.1.2 for details); data races cause the dreaded *undefined behaviour*.

Problematic race conditions typically occur where completing an operation requires modification of two or more distinct pieces of data, such as the two link pointers in the example above. Because the operation must access two separate pieces of data, these must be modified in separate instructions, and another thread could potentially access the data structure when only one of them has been completed. Race conditions can often be hard to find and hard to duplicate because the window of opportunity is small — if the modifications are done as consecutive CPU instructions, then the chance of the problem exhibiting on any one run through is very small, even if the data structure is being accessed by another thread concurrently. As the loading on the system increases, and the number of times the operation is performed increases, the chance of the problematic execution sequence occurring also increases. It is almost inevitable that such problems will show up at the most inconvenient time. Since race conditions are generally timing sensitive, they can often disappear entirely when the application is run under the debugger, since the debugger affects the timing of the program, even if only slightly.

If you're writing multi-threaded programs, race conditions can easily be the bane of your life: a large amount of the complexity in writing software that uses concurrency comes from avoiding problematic race conditions.

3.1.2 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 have either not 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 very hard 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. Lock-free programming is discussed in chapter 8.

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 cannot proceed because the data structure has been modified by another thread, then the transaction is restarted. This is termed *Software Transactional Memory*, and is an active research area at the time of writing. This will not be covered in this book, as there is no direct support for STM in C++.

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

3.2 Protecting Shared Data with Mutexes

So, you've got 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 is precisely what you get if you use a synchronization primitive called a *mutex* (named after **mut**ual **ex**clusion). Before accessing a shared data structure you *lock* the mutex associated with that data, and when you're done 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 *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.

3.2.1 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 is not recommended practice to call the member functions directly, as 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. Listing 3.1 shows how to protect a list which can be accessed by multiple threads using a **std::mutex**, along with **std::lock_guard**.

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();
    }
```

Cueballs in code and text

In listing 3.1, there is a single global variable (#1), and it is 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 part-way through a modification by **add_to_list()**.

Though there are occasions where this use of global variables is appropriate, in the majority of cases it is common to group the mutex and the protected data together in a class rather than use global variables. This is just a standard application of object-oriented design rules: by putting them in a class you're clearly marking them out as related, and you can encapsulate the functionality and enforce the protection. In this case, the functions **add_to_list** and **list_contains** would become member functions of the class, and the mutex and protected data would both become **private** members of the class, making it much easier to identify which code has access to the data, and thus which code needs to lock the mutex. If all the member functions of the class lock the mutex before accessing any other data members, and unlock it when done, then the data is nicely protected from all-comers.

Well, that's not *quite* true, as the astute among you will have noticed: if one of the member functions returns a pointer or reference to the protected data, then it doesn't matter that the member functions all lock the mutex in a nice orderly fashion, as you've just blown a big hole in the protection. **Any code that has access to that pointer or reference can now access (and potentially modify) the protected data without locking the mutex**. Protecting data with a mutex therefore requires careful interface design, to ensure that the mutex is locked before any access to the protected data, and that there are no back doors.

3.2.2 Structuring Code For Protecting Shared Data

As we've just seen, protecting data with a mutex is not quite as easy as just slapping a **std::lock_guard** object in every member function: one stray pointer or reference, and all that protection is for nothing. At one level, checking for stray pointers or references is easy — as long as none of the member functions return a pointer or reference to the protected data to their caller either via their return value or via an out parameter, the data is safe. If you dig a little deeper, it's not that straightforward — nothing ever is. As well as checking that the member functions don't pass out pointers or references to their callers, it is also important to check that they don't pass such pointers or references *in* to functions they call which are not under your control. This is just as dangerous: those functions might store the pointer or reference in a place where it can later be used without the protection of the mutex. Particularly dangerous in this regard are functions which are supplied at runtime via a function argument or other means, as in listing 3.2.

Listing 3.2: Accidentally Passing Out a Reference to Protected Data

```
class some_data
{
public:
    void do_something();
};

class data_wrapper
{
private:
    some_data data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> l(m);
        func(data);                              #1
    }
};

some_data* unprotected;

void malicious_function(some_data& protected_data)
{
    unprotected=&protected_data;
}

data_wrapper x;

void foo()
{
    x.process_data(malicious_function);        #2
    unprotected->do_something();               #3
}
```

Cueballs in code and text

#1 Call to user-defined function passing protected data
#2 Pass our malicious function in to bypass the protection
#3 Unprotected access to the "protected" data

In this example, the code in **process_data** looks harmless enough, nicely protected with **std::lock_guard**, but the call to the user-supplied function **func** (#1) means that **foo** can pass in **malicious_function** to bypass the protection (#2), and then call **do_something()** without the mutex being locked (#3).

Fundamentally, the problem with this code is that we haven't done what we set out to do: mark all the pieces of code that access the data structure as *mutually exclusive*. In this case, we missed the code in **foo()** that calls **unprotected->do_something()**. Unfortunately, this part of the problem is not something the C++ Thread Library can help us with: it is up to us as programmers to lock the right mutex to protect our data. On the upside, we have a guideline to follow, which will help us in these cases: **Don't pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions.**

Though this is a common mistake when trying to use mutexes to protect shared data, it's far from the only potential pitfall. As we'll see in the next section, it's still possible to have race conditions, even when data is protected with a mutex.

3.2.3 Spotting Race Conditions Inherent in Interfaces

Just because you're using a mutex or other mechanism to protect shared data, you're not necessarily protected from race conditions — you've still got to ensure that the appropriate data is protected. Consider our doubly-linked list example again. In order for a thread to safely delete a node, we need to ensure that we are preventing concurrent accesses to three nodes: the node being deleted, and the nodes either side. If we protected accesses to the pointers of each node individually, we would be no better off than code that used no mutexes, as the race condition could still happen — it's not the individual nodes that need protecting for the individual steps, but the whole data structure, for the whole delete operation. The easiest solution in this case is to have a single mutex which protects the entire list, as in listing 3.1.

However, just because individual operations on the list are safe, we're not out of the woods yet: you can still get race conditions, even with a really simple interface. Consider a stack data structure like the **std::stack** container adapter shown in listing 3.3. Aside from the constructors and **swap()**, there are only 5 things you can do to a **std::stack**: **push()** a new element on to the stack, **pop()** an element off the stack, read the **top()** element, check whether it's **empty()**, and read the number of elements: the **size()** of the stack. If we change **top()** so that it returns a copy rather than a reference (so we're following the guideline from 3.2.2), and protect the internal data with a mutex, this interface is still inherently subject to race conditions. This problem is not unique to a mutex-based implementation: it is an interface problem, so the race conditions would still occur with a lock-free implementation.

Listing 3.3: The interface to the std::stack container adapter.

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

    bool empty() const;
    size_t size() const;
    T& top();
    T const& top() const;
    void push(T const&);
    void push(T&&);
    void pop();
    void swap(stack&&);
};
```

The problem here is that the results of **empty()** and **size()** can't be relied upon: though they might be correct at the time of the call, once they have returned, other threads are free to access the stack, and might **push()** new elements, or **pop()** the existing ones off the stack before the thread that called **empty()** or **size()** could use that information.

In particular, if the **stack** instance is **not shared**, it is safe to check for **empty()**, and then call **top()** to access the top element if the stack is not empty, as shown below:

```
stack<int> s;
if(!s.empty())                                           #1
{
    int const value=s.top();                             #2
    s.pop();                                             #3
    do_something(value);
}
```

Cueballs in code and text

Not only is it safe in single-threaded code, it is expected: calling **top()** on an empty stack is undefined behaviour. With a shared **stack** object, **this call sequence is no longer safe**, as there might be a call to **pop()** from another thread which removes the last element in between the call to **empty()** (#1) and the call to **top()** (#2). This is therefore a classic race condition, and the use of a mutex internally to protect the stack contents doesn't prevent it: it's a consequence of the interface.

What's the solution? Well, this problem happens as a consequence of the design of the interface, so the solution is to change the interface. However, that still begs the question — what changes need to be made? In the simplest case, we could just declare that **top()** will throw an exception if there's not actually any elements in the stack when it's called. Though this directly addresses this issue, it makes for more cumbersome programming, as now we need to be able to catch an exception, even if the call to **empty()** returned **false**. This essentially makes the call to **empty()** completely redundant.

If you look closely at the snippet above, there's also potential for another race condition, but this time between the call to **top()** (#2) and the call to **pop()** (#3). Consider two threads running the above snippet of code, and both referencing the same **stack** object**, s**. This is not an unusual situation: when using threads for performance it is quite common to have several threads running the same code on different data, and a shared **stack** object is ideal for dividing work between them. Suppose that initially the stack has two elements, so we don't have to worry about the race between **empty()** and **top()** on either thread, and consider the potential execution patterns.

If the stack is protected by a mutex internally, then only one thread can be running a stack member function at any one time, so the calls get nicely interleaved, whilst the calls to **do_something()** can run concurrently. One possible execution is then as follows:

Thread A	Thread B
if(!s.empty())	
	if(!s.empty())
int const value=s.top();	
	int const value=s.top();
s.pop();	
do_something(value);	**s.pop();**
	do_something(value);

As you can see, if these are the only threads running, then there is nothing in between the two calls to **top()** to modify the stack, so both threads will see the same value. Not only that, but **there are no calls to top() between the calls to pop()**. Consequently, one of the two values on the stack is discarded without ever having been read, whilst the other is processed twice. This is yet another race condition, and far more insidious than the undefined behaviour of the **empty()**/**top()** race — there is never anything obviously wrong going on, and the consequences of the bug are likely far removed from the cause, though they obviously depend on exactly what **do_something()** really does.

This calls for a more radical change to the interface: one that combines the calls to **top()** and **pop()** under protection of the mutex. Those of you who know the history behind this part of the Standard C++ Library will be aware of the issues associated with such a call: the **std::stack** stack interface is designed this way for exception safety, after Tom Cargill pointed out [Cargill] that a combined call can have exactly the same problem, if the copy constructor for the objects on the stack can throw an exception. This problem was dealt with fairly comprehensively from an exception-safety point of view by Herb Sutter [Sutter1999], but the potential for race conditions brings something new to the mix.

For those of you who aren't aware of the issue, consider a **stack<vector<int>>**. Now, a **vector** is a dynamically-sized container, so when you copy a **vector** the library has to allocate some more memory from the heap in order to copy the contents. If the system is heavily loaded, or there are significant resource constraints, this memory allocation can fail, so the copy constructor for **vector** might throw a **std::bad_alloc** exception. This is especially likely if the **vector** contains a lot of elements. If the **pop()** function was defined to return the value popped, as well as remove it from the stack, then we have a potential problem: the value being popped is only returned to the caller *after* the stack has been modified, but the process of copying the data to return to the caller might throw an exception. If this happens, then the data just popped is lost: it has been removed from the stack, but the copy was unsuccessful! The designers of the **std::stack** interface therefore split the operation in two: get the top element (**top()**), and then remove it from the stack (**pop()**), so that if you can't safely copy the data it stays on the stack — if the problem was lack of heap memory, then maybe the application can free some memory and try again.

Unfortunately, it is precisely this split that we are trying to avoid in eliminating our race condition! Thankfully, there are alternatives, but they are not without cost.

Option 1: Pass in a reference

The first option is to pass a reference to a variable in which you wish to receive the popped value as an argument in the call to **pop()**:

```
    std::vector<int> result;
    some_stack.pop(result);
```

This works well for many cases, but has the distinct disadvantage that it requires the calling code construct an instance of the stack's value type prior to the call, in order to pass this in as the target. For some types this is just impractical, as constructing an instance is expensive in time or resources. For other types, this is not always possible, as the constructors require parameters that aren't necessarily available at this point in the code. Finally, it requires that the stored type is assignable. This is an important restriction: many user-defined types do not support assignment, though they may support move-construction or even copy-construction (and thus allow return-by-value).

Option 2: Require a no-throw copy constructor or move constructor

There is only an exception safety problem with a value-returning **`pop()`** if the return by value can throw an exception. Many types have copy constructors that don't throw exceptions, and with the new rvalue-reference support in the C++ Standard (see appendix A, section A.1), many more types will have a move constructor that doesn't throw exceptions, even if their copy-constructor does. One valid option is to restrict the use of our thread-safe stack to those types that can safely be returned by value without throwing an exception.

Though this is safe, it's not ideal: it is not possible to detect at compile time the existence of a copy or move constructor that doesn't throw an exception, and so as a library author we cannot prevent users violating this requirement, and storing values with throwing copy constructors. Also, it is quite limiting: there are many more user-defined types with copy constructors that can throw and no move constructor than there are with copy and/or move constructors that can't throw. It would be unfortunate if such types could not be stored in our thread-safe stack.

Option 3: Return a pointer to the popped item

The third option is to return a pointer to the popped item rather than return the item by value. The advantage here is that pointers can be freely copied without throwing an exception, so we've avoided Cargill's exception problem. The disadvantage is that returning a pointer requires a means of managing the memory allocated to the object, and for simple types such as integers, the overhead of such memory management can exceed the cost of just returning the type by value. For any interface that uses this option, **`std::shared_ptr`** would be a good choice of pointer type: not only does it avoid memory leaks, since the object is destroyed once the last pointer is destroyed, but the library is in full control of the memory allocation scheme, and doesn't have to use **`new`** and **`delete`**. This can be important for optimization purposes: requiring that each object in the stack be allocated separately with **`new`** would impose quite an overhead compared to the original non-thread-safe version.

Option 4: Provide both options 1 and either option 2 or 3.

Flexibility should never be ruled out, especially in generic code. If you've opted for option 2 or 3, it is relatively easy to provide option 1 as well, and this provides users of your code the ability to choose whichever option is most appropriate to them for very little additional cost.

Example Definition of a Thread-safe Stack

Listing 3.4 shows the class definition for a stack with no race conditions in the interface, that implements options 1 and 3 above: there are two overloads of **pop()**, one which takes a reference to a location in which to store the value, and one which returns a **std::shared_ptr<>**. It has a really simple interface, with only two functions: **push()** and **pop()**.

Listing 3.4: A class definition for a thread-safe stack

```
#include <exception>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class thread_safe_stack
{
public:
    stack();
    stack(const stack&);
    stack& operator=(const stack&) = delete;       #1

    void push(T new_value);
    std::shared_ptr<T> pop();
    void pop(T& value);
    bool empty() const;
};
```

Cueballs in code and text

By paring down the interface we allow for maximum safety: even operations on the whole stack are restricted: the stack itself cannot be assigned, as the assignment operator is deleted (#1) (see appendix A, section A.2), and there is 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. Our 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: we can ensure that the mutex is locked for the entirety of an operation. listing 3.5 shows a simple implementation that is a wrapper around **std::stack<>**.

Listing 3.5: A class definition for a thread-safe stack

```
#include <exception>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class thread_safe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    stack(){}
    stack(const stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;                                  #1
    }
    stack& operator=(const stack&) = delete;

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(new_value);
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();             #2
        std::shared_ptr<T> const res(new T(data.top()));   #3
        data.pop();
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
```

```
            value=data.top();
            data.pop();
        }
        bool empty() const
        {
            std::lock_guard<std::mutex> lock(m);
            return data.empty();
        }
    };
```

#1 We can't use the member initializer list to copy the stack as we must lock the source object first
#2 Check for empty before trying to pop a value
#3 Allocate the return value before modifying the stack with `pop()`

As the discussion of **`top()`** and **`pop()`** shows, problematic race conditions in interfaces essentially arise due to locking at too small a granularity: the protection doesn't cover the entirety of the desired operation. Problems with mutexes can also arise due to locking at too large a granularity: the extreme situation is a single global mutex which protects all shared data. In a system where there is a significant amount of shared data this can eliminate any performance benefits of concurrency, as the threads are forced to run one at a time, even when they are accessing different bits of data. The first versions of the Linux kernel that were designed to handle multi-processor systems used a single global kernel lock. Though this worked, it meant that a two-processor system typically had much worse performance than two single-processor systems, and performance on a four-processor system was nowhere near that of four single-processor systems — there was too much contention for the kernel, so the threads running on the additional processors were unable to perform useful work. Later revisions of the Linux kernel have moved to a more fine-grained locking scheme, so the performance of a four-processor system is much nearer the ideal of four times that of a single processor system, as there is much less contention.

One issue with fine-grained locking schemes is that sometimes you need more than one mutex locked in order to protect all the data in an operation. As described above, sometimes the right thing to do is just increase the granularity of the data covered by the mutexes, so that only one mutex needs to be locked. However, sometimes that is undesirable, such as when the mutexes are protecting separate instances of a class. In this case, locking at “the next level up” would mean either leaving the locking to the user, or having a single mutex which protected all instances of that class, neither of which is particularly desirable.

If you end up having to lock two or more mutexes for a given operation, there's another potential problem lurking in the wings: *deadlock*. This is almost the opposite of a race condition: rather than two threads “racing” to be first, each one is waiting for the other, so neither makes any progress.

3.2.4 Deadlock: the Problem and a Solution

Imagine you've got a toy that comes in two parts, and you need both parts to play with it — a toy drum and drumstick, for example. Now imagine that you've got two small children, both of whom like playing with it. If one of them gets both the drum and the drumstick, they can merrily play the drum until they get fed up. If the other child wants to play, they have to wait, however sad that makes them. Now imagine that the drum and the drumstick are buried (separately) in the toy box, and your children both decide to play with it at the same time, so go rummaging in the toy box. One finds the drum and the other finds the drumstick. Now they're stuck — unless one or the other decides to be nice and let the other child play, each will hold onto whatever they've got and demand the other gives them the other bit, and neither gets to play.

Now imagine that you've not got children arguing over toys, but threads arguing over locks on mutexes: each of a pair of threads needs to lock both of a pair of mutexes to perform some operation, and each thread has one mutex and is waiting for the other. Neither thread can proceed, as each is waiting for the other to release its mutex. This scenario is called *deadlock*, and is the biggest problem with having to lock two or more mutexes in order to perform an operation.

The common advice for avoiding deadlock is to always lock the two mutexes in the same order: if you always lock mutex A before mutex B, then you'll never deadlock. Sometimes this is straightforward, as the mutexes are serving different purposes, but other times it is not so simple, such as when the mutexes are each protecting a separate instance of the same class. Consider for example a comparison operation on two instances of the same class — in order to avoid the comparison being affected by concurrent modifications, the mutexes on both instances must be locked. However, if a fixed order is chosen (e.g. the mutex for the instance supplied as the first parameter, then the mutex for the instance supplied as the second parameter), then this can backfire: all it takes is for two threads to try and compare the same instances with the parameters swapped, and you've got 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 listing 3.6 shows how to use this for a comparison operator. First, the arguments are checked to ensure they are different instances, as attempting to acquire a lock on a **std::mutex** when you already hold it is undefined behaviour (a mutex that does permit multiple locks by the same thread is provided in the form of **std::recursive_mutex**. See section xxx 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 attempting to lock the mutex in the constructor.

This ensures that the mutexes are correctly unlocked on function exit in the general case where the protected operation might throw an exception, as well as allowing for a simple return with the comparison result in this case. Also, it's worth noting that locking either of **lhs.m** or **rhs.m** inside the call to **std::lock** can throw an exception: in this case the exception is propagated out of **std::lock**. If **std::lock** has successfully acquired a lock on one mutex and an exception is thrown when it tries to acquire a lock on the other mutex, then this first lock is released automatically: **std::lock** provides all-or-nothing semantics with regard to locking the supplied mutexes.

Listing 3.6: Locking two mutexes with std::lock() in a comparison operator

```
class some_big_object;
bool operator<(some_big_object& lhs,some_big_object& rhs);

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

    friend bool operator<(X const& lhs, X const& rhs)
    {
        if(&lhs==&rhs)
            return false;
        std::lock(lhs.m,rhs.m);                                        #1
        std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);    #2
        std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);    #3
        return lhs.some_detail<rhs.some_detail;
    }
};
```

Cueballs in code and preceding text

Though **std::lock** can help us avoid deadlock in those cases where we need to acquire two or more locks together, it doesn't help if they are acquired separately — in that case we have to rely on our discipline as developers to ensure we don't get deadlock. This isn't easy: deadlocks are one of the nastiest problems to encounter in multi-threaded code, and are often unpredictable, with everything working fine the majority of the time. There are, however, some relatively simple rules which can help us to write deadlock-free code.

3.2.5 Further Guidelines for Avoiding Deadlock

Deadlock doesn't just occur with locks, though that is 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 is 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 isn't limited to two threads: a cycle of three or more threads will still cause deadlock. The guidelines for avoiding deadlock essentially 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 completely, it's impossible to get a deadlock from the lock usage alone as 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 whilst holding a lock

This is actually a very simple follow on from the previous guideline. Since it is user-supplied, you have no idea what that code could do: it could do anything, including acquiring a lock. If you call user-supplied code whilst holding a lock, and that code acquires a lock, then 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, then 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 cannot acquire them as a single operation with **`std::lock`**, then the next best thing is to acquire them in the same order in every thread. We touched on this in section 3.2.4 as one way of avoiding deadlock when acquiring two mutexes: the key is to define the order in a way that is consistent between threads. In some cases this is relatively easy. For example, if we look at the stack from section 3.2.3, the mutex is internal to each stack instance, but the operations on the data items stored in stack requires calling user-supplied code. However, we can just add the constraint that none of the operations on the data items stored in the stack should perform any operation on the stack itself. This puts the burden on the user of the stack, but its rather uncommon for the data stored in a container to access that container, and its quite apparent when this is happening, so it's not a particularly difficult burden to carry.

In other cases, this is not so straightforward, as we discovered with the comparison operator in section 3.2.4. At least in that case we could lock the mutexes simultaneously, but that's not always possible. If we look back at the linked list example from section 3.1, then one possibility for protecting the list is to have a mutex per node. Then, in order to access the list, threads must acquire a lock on every node they are interested in. For a thread to delete an item it must then acquire the lock on three nodes: the node being deleted, and the nodes either side, since they are all being modified in some way. Likewise, to traverse the list a thread must keep hold of the lock on the current node whilst it acquires the lock on the next one in the sequence, in order to ensure that the next pointer is not modified in the mean time. Once the lock on the next node has been acquired, the lock on the first can be released as it is no longer necessary.

This “hand over hand” locking style allows multiple threads to access the list, provided each is accessing a different node. However, in order to prevent deadlock the nodes must always be locked in the same order: if two threads tried to traverse the list in reverse order using hand-over-hand locking, then they could deadlock with each other in the middle of the list. If nodes A and B are adjacent in the list, then the thread going one way will try be holding the lock on node A and try and acquire the lock on node B. A thread going the other way would be holding the lock on node B, and trying to acquire the lock on node A: a classic scenario for deadlock.

Likewise, when deleting a node B that lies between nodes A and C, if that thread acquires the lock on B before the locks on A and C then it has the potential to deadlock with a thread traversing the list. Such a thread would either try and lock A or C first (depending on the direction of traversal), but would then find that it couldn't obtain a lock on B because the thread doing the deleting was holding the lock on B and trying to acquire the locks on A and C.

One way to prevent deadlock here is to define an order of traversal, so a thread must always lock A before B, and B before C. This would eliminate the possibility of deadlock, at the expense of disallowing reverse traversal. Similar conventions can often be established for other data structures.

Use a lock hierarchy

Though this is really a particular case of defining a lock ordering, a lock hierarchy can provide a means of checking that the convention is adhered to at run time. The idea is basically that you divide your application into layers, and identify all the mutexes that may be locked in any given layer. When code tries to lock a mutex then it is not permitted to lock that mutex if it already holds a lock from a lower layer. This can be checked at runtime by assigning layer numbers to each mutex and keeping a record of which mutexes are locked by each thread. Listing 3.7 shows an example of two threads using a hierarchical mutex: **thread_a()** (#1) abides by the rules, so runs fine. On the other hand, **thread_b()** (#2) disregards the rules, and therefore will fail at runtime.

Listing 3.7: Using a lock hierarchy to prevent deadlock

```
hierarchical_mutex high_level_mutex(10000);                             #4
hierarchical_mutex low_level_mutex(5000);                               #7

int do_low_level_stuff();

int low_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(low_level_mutex);            #6
    return do_low_level_stuff();
}

void high_level_stuff(int some_param);

void high_level_func()
{
    std::lock_guard<hierarchical_mutex> lk(high_level_mutex);           #3
    high_level_stuff(low_level_func());                                 #5
}

void thread_a()                                                         #1
{
    high_level_func();
}

hierarchical_mutex other_mutex(100);                                    #9
void do_other_stuff();

void other_stuff()
{
    high_level_func();                                                  #10
    do_other_stuff();
}

void thread_b()                                                         #2
{
    std::lock_guard<hierarchical_mutex> lk(other_mutex);                #8
    other_stuff();
}
```

Cueballs in code and preceding and following text

thread_a() calls **high_level_func()**, which locks the **high_level_mutex** (#3) (with a hierarchy value of 10000 (#4)) and then calls **low_level_func()** (#5) with this mutex locked in order to get the parameter for **high_level_stuff()**. **low_level_func()** then locks the **low_level_mutex** (#6), but that's fine since this mutex has a lower hierarchy value of 5000 (#7).

thread_b() on the other hand is **not** fine. First off, it locks the **other_mutex** (#8), which has a hierarchy value of only 100 (#9). This means it should really be protecting ultra-low-level data. When **other_stuff()** calls **high_level_func()** (#10) it is thus violating the hierarchy — **high_level_func()** tries to acquire the **high_level_mutex** which has a value of 10000, considerably more than the current hierarchy value of 100. The **hierarchical_mutex** will therefore report an error, possibly by throwing an exception or aborting the program. Deadlocks between hierarchical mutexes are thus impossible, since the mutexes themselves enforce the lock ordering.

This example also demonstrates another point — the use of the **std::lock_guard<>** template with a user-defined mutex type. **hierarchical_mutex** is not part of the standard, but is easy to write — a simple implementation is shown in listing 3.8. Even though it is a user-defined type, it can be used with **std::lock_guard<>** because it implements the three member functions required to satisfy the mutex concept: **lock()**, **unlock()** and **try_lock()**. We haven't yet seen **try_lock()** used directly, but it is fairly simple — if the lock on the mutex is held by another thread then it just returns **false** rather than waiting until the calling thread can acquire the lock on the mutex. It may also be used by **std::lock()** internally, as part of the deadlock-avoidance algorithm.

Listing 3.8: A simple hierarchical mutex

```
class hierarchical_mutex
{
    std::mutex internal_mutex;
    unsigned long const hierarchy_value;
    unsigned previous_hierarchy_value;
    static thread_local unsigned long this_thread_hierarchy_value;     #1

    void check_for_hierarchy_violation()
    {
        if(this_thread_hierarchy_value <= hierarchy_value)             #3
        {
            throw std::logic_error(“mutex hierarchy violated”);
        }
    }
    void update_hierarchy_value()
```

```
        {
            previous_hierarchy_value=this_thread_hierarchy_value;          #7
            this_thread_hierarchy_value=hierarchy_value;
        }
    public:
        explicit hierarchical_mutex(unsigned long value):
            hierarchy_value(value),
            previous_hierarchy_value(0)
        {}
        void lock()
        {
            check_for_hierarchy_violation();
            internal_mutex.lock();                                         #4
            update_hierarchy_value();                                      #5
        }
        void unlock()
        {
            this_thread_hierarchy_value=previous_hierarchy_value;          #6
            internal_mutex.unlock();
        }
        bool try_lock()
        {
            check_for_hierarchy_violation();
            if(!internal_mutex.try_lock())                                 #8
                return false;
            update_hierarchy_value();
            return true;
        }
    };
    thread_local unsigned long
        hierarchical_mutex::this_thread_hierarchy_value(ULONG_MAX);        #2
```

Cueballs in code and text

The key here is the use of the **thread_local** value representing the hierarchy value for the current thread: **this_thread_hierarchy_value** (#1). It is initialized to the maximum value (#2), so initially any mutex can be locked. Because it is declared **thread_local**, every thread has its own copy, so the state of the variable in one thread is entirely independent of the state of the variable when read from another thread.

So, the first time a thread locks an instance of **hierarchical_mutex** the value of **this_thread_hierarchy_value** is **ULONG_MAX**. By its very nature, this is greater than any other value so the check in **check_for_hierarchy_violation()** (#3) passes. With that check out of the way, **lock()** just delegates to the internal mutex for the actual locking (#4). Once this lock has succeeded, we can update the hierarchy value (#5).

If we now lock *another* **hierarchical_mutex** whilst holding the lock on this first one, then the value of **this_thread_hierarchy_value** now reflects the hierarchy value of the first mutex. The hierarchy value of this second mutex must now be less than that of the mutex already held in order for the check (#3) to pass.

Now, it's important to save the previous value of the hierarchy value for the current thread so we can restore it in **unlock()** (#6), otherwise you'd never be able to lock a mutex with a higher hierarchy value again, even if the thread didn't hold any locks. Since we only store this previous hierarchy value when we hold the **internal_mutex** (#7), and we restore it *before* we unlock the internal mutex (#6), we can safely store it in the **hierarchical_mutex** itself, as it is safely protected by the lock on the internal mutex.

try_lock() works the same as **lock()** except that if the call to **try_lock()** on the **internal_mutex** fails (#8) then we don't own the lock, so we don't update the hierarchy value, and return **false** rather than **true**.

Though detection is a run-time check, it is at least not timing dependent — you don't have to wait around for the rare conditions that cause deadlock to show up. Also, the design process required to divide up the application and mutexes in this way can help eliminate many possible causes of deadlock before they even get written. It might be worth performing the design exercise even if you then don't go as far as actually writing the run-time checks.

Extending these guidelines beyond locks

As I mentioned back at the beginning of this section, deadlock doesn't just occur with locks: it can occur with any synchronization construct that can lead to a wait cycle. It is therefore worth extending these guidelines to cover those cases too. For example, just like acquiring nested locks should be avoided if possible, it is a bad idea to wait for a thread whilst holding a lock, since that thread might need to acquire the lock in order to proceed. Similarly, if you're going to wait for a thread to finish it might be worth identifying a thread hierarchy, such that a thread only waits for threads lower down the hierarchy. One simple way to do this is to ensure that your threads are joined in the same function that started them, as described in sections 3.1.2 and 3.3.

Once you've designed your code to avoid deadlock, **std::lock()** and **std::lock_guard** cover most of the cases of simple locking, but sometimes more flexibility is required. For those cases, the Standard Library provides the **std::unique_lock** template. Like **std::lock_guard**, this is a class template parameterized on the mutex type, and it also provides the same RAII-style lock management as **std::lock_guard**, but with a bit more flexibility.

3.2.6 Flexible Locking with std::unique_lock

std::unique_lock provides a bit more flexibility than **std::lock_guard**, by relaxing the invariants: a **std::unique_lock** instance does not always own the mutex that it is associated with. First off, just as you can pass **std::adopt_lock** as a second argument to the constructor to have the lock object manage the lock on a mutex, you can also pass **std::defer_lock** as the second argument to indicate that the mutex should remain unlocked on construction. The lock can then be acquired later by calling **lock()** on the **std::unique_lock** object (**not** the mutex), or by passing the **std::unique_lock** object itself to **std::lock()**. Listing 3.6 could just as easily have been written as shown in listing 3.9, using **std::unique_lock** and **std::defer_lock** (#1) rather than **std::lock_guard** and **std::adopt_lock**. The code has the same line count, and is essentially equivalent, apart from one small thing: **std::unique_lock** takes more space and is a fraction slower to use than **std::lock_guard**. The flexibility of allowing a **std::unique_lock** instance **not** to own the mutex comes at a price: this information has to be stored, and it has to be updated.

Listing 3.9: Locking two mutexes with lock() in a comparison operator

```
class some_big_object;
bool operator<(some_big_object& lhs,some_big_object& rhs);

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

    friend bool operator<(X const& lhs, X const& rhs)
    {
        if(&lhs==&rhs)
            return false;
        std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock);  #1
        std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock);  #1
        std::lock(lock_a,lock_b);                                    #2
        return lhs.some_detail<rhs.some_detail;
    }
};
```

Cueballs in code and preceding and following text

In listing 3.9, the **std::unique_lock** objects could be passed to **std::lock()** (#2) because **std::unique_lock** provides **lock()**, **try_lock()** and **unlock()** member functions. These forward to the member functions of the same name on the underlying mutex to do the actual work, and just update a flag inside the **std::unique_lock** instance to indicate whether or not the mutex is currently owned by that instance. This flag is necessary in order to ensure that **unlock()** is called correctly in the destructor — if the instance **does** own the mutex, the destructor **must** call **unlock()**, and if the instance **does**

not own the mutex, it **must not** call **unlock()**. This flag can be queried by calling the **owns_lock()** member function.

As you might expect, this flag has to be stored somewhere. Therefore, the size of a **std::unique_lock** object is typically larger than that of a **std::lock_guard** object, and there is also a slight performance penalty when using **std::unique_lock** over **std::lock_guard** as the flag has to be updated or checked, as appropriate. If **std::lock_guard** is sufficient for your needs, I would therefore recommend using it in preference. However, there are cases where **std::unique_lock** is a better fit for the task at hand, as you need to make use of the additional flexibility. One example is deferred locking, as we've already seen; another case is where the ownership of the lock needs to be transferred from one scope to another.

3.2.7 Transferring Mutex Ownership Between Scopes

Since **std::unique_lock** instances do not have to own their associated mutexes, the ownership of a mutex can be transferred between instances by *moving* the instances around. In some cases such transfer is automatic, such as when returning an instance from a function, and in other cases you have to do it explicitly by calling **std::move()**. Fundamentally this depends on whether or not the source is an *lvalue* — a real variable or reference to one — or an *rvalue* — a temporary of some kind. Ownership transfer is automatic if the source is an *rvalue*, and must be done explicitly for an *lvalue* in order to avoid accidentally transferring ownership away from a variable. **std::unique_lock** is an example of a type that is *movable* but not *copyable*. See appendix A, section A.1.1 for more about move semantics.

One possible use is to allow a function to lock a mutex and transfer ownership of that lock to the caller, so the caller can then perform additional actions under the protection of the same lock. The code snippet below shows an example of this: the function **get_lock()** locks the mutex, and then prepares the data before returning the lock to the caller. Since **lk** is an automatic variable declared within the function, it can be returned directly (#1) without a call to **std:move()**: the compiler takes care of calling the move constructor. The **process_data()** function can then transfer ownership directly into its own **std::unique_lock** instance (#2), and the call to **do_something()** can rely on the data being correctly prepared without another thread altering the data in the mean time.

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk;                                      #1
}
void process_data()
```

```
    {
        std::unique_lock<std::mutex> lk(get_lock());    #2
        do_something();
    }
```

Cueballs in code and preceding text

Typically this sort of pattern would be used where the mutex to be locked was dependent on the current state of the program, or on an argument passed in to the function that returned the **std::unique_lock** object. One such usage is where the lock is not returned directly, but is a data member of a "gateway" class used to ensure correctly-locked access to some protected data. In this case, all access to the data is through this gateway class: when you wish to access the data you obtain an instance of the gateway class (by calling a function such as **get_lock()** in the preceding example), which acquires the lock. You can then access the data through member functions of the gateway object. When you're done, you destroy the gateway object, which releases the lock and allows other threads to access the protected data. Such a gateway object may well be movable (so that it can be returned from a function), in which case the lock object data member also needs to be movable.

The flexibility of **std::unique_lock** also allows instances to relinquish their locks before they are destroyed. This can be done with the **unlock()** member function, just like for a mutex: **std::unique_lock** supports the same basic set of member functions for locking and unlocking as a mutex does, in order that it can be used with generic functions such as **std::lock**. The ability to release a lock before the **std::unique_lock** instance is destroyed means that you can optionally release it in a specific code branch if it is apparent that the lock is no longer required. This can be important for the performance of the application: holding a lock for longer than it is required can cause a drop in performance, since other threads waiting for the lock are prevented from proceeding for longer than necessary.

3.2.8 Locking at an Appropriate Granularity

The "granularity" of a lock is something we touched on earlier, in section 3.2.3: the lock granularity is a hand-waving term to describe the amount of data protected by a single lock. A fine-grained lock protects a small amount of data, and a coarse-grained lock protects a large amount of data. Not only is it important to choose a sufficiently coarse lock granularity to ensure the required data is protected, it is also important to ensure that a lock is only held for the operations that actually require it. We all know the frustration of waiting in the queue for a checkout in a supermarket with a trolley full of shopping only for the person currently being served to suddenly realise that they forgot some cranberry sauce, and then leave everybody waiting whilst they go and find some, or for the cashier to be ready for payment and the customer to only then start rummaging in their bag for their wallet. Everything proceeds much more easily if everybody gets to the checkout with everything they want, and with an appropriate means of payment ready.

The same applies to threads: if multiple threads are waiting for the same resource (the cashier at the checkout), then it will increase the total time spent waiting if any thread holds the lock for longer than necessary (don't wait until you've got to the checkout to start looking for the cranberry sauce). Where possible, only lock a mutex whilst actually accessing the shared data: try and do any processing of the data outside the lock. In particular, don't do any really time consuming activities like file I/O whilst holding a lock. File I/O is typically hundreds (if not thousands) of times slower than reading or writing the same volume of data from memory, so unless the lock is really intended to protect access to the file, performing I/O whilst holding the lock will delay *other* threads unnecessarily (as they will block waiting to acquire the lock), potentially eliminating any performance gain from the use of multiple threads.

std::unique_lock works very well in this situation, as you can call **unlock()** when the code no longer needs access to the shared data, and then call **lock()** again if access is required later in the code:

```
void get_and_process_data()
{
    std::unique_lock<std::mutex> my_lock(the_mutex);
    some_class data_to_process=get_next_data_chunk();
    my_lock.unlock();                              #1
    result_type result=process(data_to_process);
    my_lock.lock();                                #2
    write_result(data_to_process,result);
}
```

#1 We don't need the mutex locked across the call to process()
#2 Lock the mutex again to write the result

Hopefully it is obvious that if you have one mutex protecting an entire data structure, then not only is their likely to be more contention for the lock, but also the potential for reducing the time that the lock is held is less: more of the operation steps will require a lock on the same mutex, so the lock must be held for longer. This double-whammy of a cost is thus also a double incentive to move towards finer-grained locking wherever possible.

As the example above shows, locking at an appropriate granularity isn't only about the amount of data locked, it's also about how long the lock is held for, and what operations are performed whilst the lock is held. **In general, a lock should only be held for the minimum possible length of time in order to perform the required operations**. This also means that time consuming operations such as acquiring another lock (even if you know it won't deadlock) or waiting for I/O to complete should not be done whilst holding a lock unless absolutely necessary.

Let's look again at the comparison operator of listings 3.6 and 3.9. In this case, the member being compared is of type **some_big_object** — a name chosen to indicate that it's going to be more expensive to copy the objects than to compare them. However, suppose the data member was just a plain **int**. Would this make a difference? **int**s are cheap to copy, so we could easily copy the data for each object being compared whilst only holding the lock for that object, and then compare the copied values. This would mean that we were holding the lock on each mutex for the minimum amount of time, and also that we were not holding one lock whilst locking another. Listing 3.10 shows a class **Y** for which this is the case, and a sample implementation of the equality comparison operator.

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);           #3
        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();             #1
        int const rhs_value=rhs.get_detail();             #2
        return lhs_value==rhs_value;                      #4
    }
};
```

Cueballs in code and text

In this case, the comparison operator first retrieves the values to be compared by calling the **get_detail()** member function (#1,#2). This function retrieves the value whilst protecting it with a lock (#3). The comparison operator then compares the retrieved values (#4). 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**. If the operator in listing 3.9 returns **true**, it means that at some point in time there were values stored in the two instances of **X** such that **lhs.some_detail<rhs.some_detail**. 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 #1 and #2, 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 is 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 then you are exposing yourself to race conditions**.

Sometimes, there just isn't an appropriate level of granularity because not all accesses to the data structure require the same level of protection. In this case, it might be appropriate to use an alternative mechanism, instead of a plain **std::mutex**.

3.3 Alternative Facilities for Protecting Shared Data

Though they're the most general mechanism, mutexes aren't the only game in town when it comes to protecting shared data: there are alternatives which provide more appropriate protection in specific scenarios.

One particularly extreme (but remarkably common) case is where the shared data only needs protecting from concurrent access whilst it is being initialized, but after that no explicit synchronization is required. This might be because the data is read-only once created, and so there are no possible synchronization issues, or it might be because the necessary protection is performed implicitly as part of the operations on the data. In either case, locking a mutex after the data has been initialized, purely in order to protect the initialization, is unnecessary, and a needless hit to performance. It is for this reason that the C++ Standard provides a mechanism purely for protecting shared data during initialization.

3.3.1 Protecting Shared Data During Initialization

Suppose you have a shared resource that is sufficiently expensive to construct that you only want to do so if it is actually required: maybe it opens a database connection, or allocates a lot of memory. *Lazy-initialization* such as this is common in single-threaded code: each operation that requires the resource first checks to see if it has been initialized, and initializes it before use if not:

```
std::shared_ptr<some_resource> resource_ptr;
void foo()
```

```
    {
        if(!resource_ptr)
        {
            resource_ptr.reset(new some_resource);        #1
        }
        resource_ptr->do_something();
    }
```

Cueballs in code and text

If the shared resource itself is safe for concurrent access, then the only part that needs protecting when converting this to multi-threaded code is the initialization (#1), but a naïve translation such as that in listing 3.11 can cause unnecessary *serialization* of threads using the resource. This is because each thread must wait on the mutex in order to check whether or not the resource has already been initialized.

Listing 3.11: Thread-safe Lazy Initialization Using a Mutex

```
    std::shared_ptr<some_resource> resource_ptr;
    std::mutex resource_mutex;
    void foo()
    {
        std::unique_lock<std::mutex> lk(resource_mutex);    #1
        if(!resource_ptr)
        {
            resource_ptr.reset(new some_resource);          #2
        }
        lk.unlock();
        resource_ptr->do_something();
    }
```

#1 All threads are serialized here
#2 Only the initialization needs protection

This code is common enough, and the unnecessary serialization problematic enough, that many people have tried to come up with a better way of doing this, including the infamous *Double-Checked Locking* pattern: the pointer is first read without acquiring the lock (#1), and the lock is only acquired if the pointer is **NULL**. The pointer is then checked *again* once the lock has been acquired (#2) (hence the *double-checked* part) in case another thread has done the initialization between the first check and this thread acquiring the lock.

```
    void undefined_behaviour_with_double_checked_locking()
    {
        if(!resource_ptr)                                     #1
        {
            std::lock_guard<std::mutex> lk(resource_mutex);
            if(!resource_ptr)                                 #2
            {
                resource_ptr.reset(new some_resource);        #3
            }
        }
        resource_ptr->do_something();                          #4
    }
```

Cueballs in code and preceding and following text

Unfortunately this pattern is infamous for a reason: it has the potential for nasty race conditions, since the read outside the lock (#1) is not synchronized with the write done by another thread inside the lock (#3). This therefore creates a race condition that covers not just the pointer itself, but the object pointed to: even if a thread sees the pointer written by another thread, it might not see the newly-created instance of **some_resource**, resulting in the call to **do_something()** operating on incorrect values. This is an example of the type of race condition defined as a *data race* by the C++ Standard, and thus specified as *undefined behaviour*. It is therefore quite definitely something to be avoided. See chapter 5 for a detailed discussion of the memory model including what constitutes a *data race*.

The C++ Standards Committee also saw that this was an important scenario, and so the C++ Standard Library provides **std::once_flag** and **std::call_once** to handle this situation. Rather than locking a mutex and explicitly checking the pointer, every thread can just use **std::call_once**, safe in the knowledge that the pointer will have been initialized by some thread (in a properly synchronized fashion) by the time **std::call_once** returns. Use of **std::call_once** will typically have a lower overhead than using a mutex explicitly, especially when the initialization has already been done, so should be used in preference where it matches the required functionality. The example below shows the same operation as listing 3.11, rewritten to use **std::call_once**. In this case, the initialization is done by calling a function, but it could just as easily have been done with an instance of a class with a function call operator. Like most of the functions in the standard library that take functions or predicates as arguments, **std::call_once** works with any function or callable object.

```
std::shared_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;

void init_resource()
{
    resource_ptr.reset(new some_resource);
}

void foo()
{
    std::call_once(resource_flag,init_resource); #1
    resource_ptr->do_something();
}
```

#1 This initialization is called exactly once

In this example, both the **std::once_flag** and data being initialized are namespace-scope objects, but **std::call_once()** can just as easily be used for lazy initialization of class members, as in listing 3.12. In that example, the initialization is done either by the first call to **send_data**() (#1), or to the first call to **receive_data()** (#2). The use of the member function **open_connection()** to initialize the data also requires that the **this** pointer is passed in: just as for other functions in the Standard Library that accept callable objects, such as the constructor for **std::thread**, and **std::bind()**, this is done by passing an additional argument to **std::call_once()** (#3).

Listing 3.12: Thread-safe Lazy Initialization of a Class Member Using std::call_once

```
class X
{
private:
    connection_info connection_details;
    connection_handle connection;
    std::once_flag connection_init_flag;

    void open_connection()
    {
        connection=connection_manager.open(connection_details);
    }
public:
    X(connection_info const& connection_details_):
        connection_details(connection_details_)
    {}
    void send_data(data_packet const& data)                          #1
    {
        std::call_once(connection_init_flag,&X::open_connection,this); #3
        connection.send_data(data);
    }
    data_packet receive_data()                                       #2
    {
        std::call_once(connection_init_flag,&X::open_connection,this); #3
        return connection.receive_data();
    }
};
```

Cueballs in code and preceding text

One scenario where there is a potential race condition over initialization is that of a local variable declared with **static**. The initialization of such a variable is defined to occur the first time control passes through its declaration: for multiple threads calling the function, this means there is the potential for a race condition to define “first”. On many pre-C++0x compilers this race condition is problematic in practice, as multiple threads may believe they are first, and try to initialize the variable, or threads may try to use it after initialization has started on another thread, but before it is finished. In C++0x this problem is solved: the initialization is defined to happen on exactly one thread, and no other threads will proceed until that initialization is complete, so the race condition is just over which thread gets to do the initialization, rather than anything more problematic. This can be used as an alternative to **std::call_once** for those cases where a single global instance is required:

```
class my_class;
my_class& get_my_class_instance()
{
    static my_class instance; #1
    return instance;
}
```

#1 Initialization of instance is guaranteed to be thread-safe.

Multiple threads can then call **get_my_class_instance()** safely, without having to worry about race conditions on the initialization.

Protecting data only for initialization is a special case of a more general scenario: that of a rarely-updated data structure. For most of the time, such a data structure is read-only, and can therefore be merrily read by multiple threads concurrently, but on occasion the data structure may need updating. What is needed here is a protection mechanism that acknowledges this fact.

3.3.2 Protecting Rarely-Updated Data Structures

Consider a table used to store a cache of DNS entries for resolving domain names to their corresponding IP addresses. Typically, a given DNS entry will remain unchanged for a long period of time — in many cases DNS entries remain unchanged for years. Though new entries may be added to the table from time to time as users access different websites, this data will therefore remain largely unchanged throughout its life. It is of course important that the validity of the cached entries is checked periodically, but this still only requires an update if the details have actually changed.

Though updates are rare, they can still happen, and if this cache is to be accessed from multiple threads, it will need to be appropriately protected during updates to ensure that none of the threads reading the cache will see a broken data structure.

In the absence of a special-purpose data structure that exactly fits the desired usage, and which is specially designed for concurrent updates and reads (such as those in chapters 6 and 7), such an update requires that the thread doing the update has exclusive access to the data structure until it has completed the operation. Once the change is complete, the data structure is again safe for multiple threads to access concurrently. Using a **std::mutex** to protect the data structure is therefore overly pessimistic, as it will eliminate the possible concurrency in reading the data structure when it is not undergoing modification: what is needed is a different kind of mutex. This new kind of mutex is typically called a reader-writer mutex, as it allows for two different kinds of usage: exclusive access by a single “writer” thread, or shared, concurrent access by multiple “reader” threads.

Unfortunately, the new C++ Standard Library does not provide such a mutex out of the box, though one was proposed to the Standards Committee [Hinnant]. Since the proposal was not accepted, the examples in this section use the implementation provided by the Boost Library, which is based on the proposal.

Rather than using an instance of **std::mutex** for the synchronization, instead we use an instance of **boost::shared_mutex**. For the update operations, **std::lock_guard< boost::shared_mutex>** and **std::unique_lock<boost::shared_mutex>** can be used for the locking, in place of the corresponding **std::mutex** specializations. These ensure exclusive access, just as with **std::mutex**. Those threads that don't need to update the data structure can instead use **boost::shared_lock<boost::shared_mutex>**, to obtain *shared* access. This is used just the same as **std::unique_lock**, except that multiple threads may have a shared lock on the same **boost::shared_mutex** at the same time. The only constraint is that if any thread has a shared lock, a thread that tries to acquire an exclusive lock will block until all other threads have relinquished their locks, and likewise if any thread has an exclusive lock, no other thread may acquire a shared or exclusive lock until the first thread has relinquished its lock.

Listing 3.13 shows a simple DNS cache like the one described above, 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;
    boost::shared_mutex entry_mutex;
public:
    dns_entry find_entry(std::string const& domain)
    {
        boost::shared_lock<boost::shared_mutex> lk(entry_mutex);   #1
        std::map<std::string,dns_entry>::const_iterator const it=
            entries.find(domain);
        return (it==entries.end())?dns_entry():*it;
    }
    void update_or_add_entry(std::string const& domain,
                             dns_entry const& dns_details)
    {
        std::lock_guard<boost::shared_mutex> lk(entry_mutex);     #2
        entries[domain]=dns_details;
    }
};
```

Cueballs in code and text

In listing 3.13, **find_entry()** uses an instance of **boost::shared_lock<>** to protect it for shared, read-only access (#1): 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 whilst the table is updated: 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.

3.4 Summary

In this chapter we've discussed how problematic race conditions can be disastrous when sharing data between threads, and how to uses **std::mutex** and careful interface design to avoid them. We've also seen that mutexes aren't a panacea, and do have their own problems in the form of deadlock, though the C++ Standard Library provides us with a tool to help avoid that in the form of **std::lock()**. We then looked at some further techniques for avoiding deadlock, followed by a brief look at transferring lock ownership, and issues surrounding choosing the appropriate granularity for locking. Finally, we've covered the alternative data protection facilities provided for specific scenarios, such as **std::call_once()**, and **boost::shared_mutex**.

One thing that we haven't covered yet however is waiting for input from other threads: our thread-safe stack just throws an exception if the stack is empty, so if one thread wanted to wait for another thread to push a value on the stack (which is, after all, one of the primary uses for a thread-safe stack) it would have to repeatedly try and pop a value, retrying if an exception got thrown. This consumes valuable processing time in performing the check, without actually making any progress: indeed the constant checking might *hamper* progress by preventing the other threads in the system from running. What is needed is some way for a thread to wait for another thread to complete a task without consuming CPU time in the process. Chapter 4 builds on the facilities we've discussed for protecting shared data, and introduces the various mechanisms for synchronizing operations between threads in C++.

4

Synchronizing Concurrent Operations

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

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

4.1 Waiting for an Event or other Condition

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

How does that relate to threads? Well, if one thread is waiting for a second thread to complete a task, it has several options. Firstly, it could just keep checking a flag in shared data (protected by a mutex), and have the second thread set the flag when it completed 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 these work against the very thread doing the waiting, as they limit the resources available to the thread being waited for, and even prevent it from setting the flag when it is 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()`** function:

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

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();                                               #1
        std::this_thread::sleep(std::chrono::milliseconds(100));   #2
        lk.lock();                                                 #3
    }
}
```

#1 Unlock the mutex whilst we sleep, so another thread can acquire it and set the flag
#2 Sleep for 100ms
#3 Lock the mutex again before we loop round to check the flag

This is an improvement, since the thread doesn't waste processing time whilst it is sleeping, but it's hard to get the sleep period right: too short a sleep in between checks and the thread still wastes processing time checking, too long a sleep and the thread will carry on sleeping even when the task it is waiting for is complete, introducing a delay. It's rare that this "oversleeping" will have a direct impact on the operation of the program, but it could mean dropped frames in a fast-paced game, or overrunning a time slice in a real-time application.

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

4.1.1 Waiting for a Condition with Condition Variables

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

So, how do we use a **std::condition_variable** to handle the example in the introduction: how do we let the thread that's waiting for work sleep until there is data to process? Listing 4.1 shows one way we could do this with a condition variable.

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

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

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

void data_processing_thread()
{
    while(true)
    {
        std::unique_lock<std::mutex> lk(mut);                 #4
        data_cond.wait(lk,[]{return !data_queue.empty();});   #5
        data_chunk data=data_queue.front();
        data_queue.pop();
        lk.unlock();                                          #6
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

Cueballs in code and text

First off, we've got a queue (#1) that is used to pass the data between the two threads. When the data is ready, the thread preparing the data locks the mutex protecting the flag using a **std::lock_guard**, and pushes the data onto the queue (#2). It then calls the **notify_one()** member function on the **std::condition_variable** instance to notify the waiting thread (#3).

On the other side of the fence, we've got the processing thread. This thread first locks the mutex, but this time with a **std::unique_lock** rather than a **std::lock_guard** (#4) — we'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 (#5). Lambda functions are a new feature in C++0x which allow you to write an anonymous function as part of another expression, and they are ideally suited for specifying predicates for standard library functions such as **wait()**. In this case, the simple lambda function **[]{return !data_queue.empty();}** checks to see if the **data_queue** is not **empty()** — i.e. there is some data in the queue ready for processing. Lambda functions are described in more detail in appendix A.

The implementation of **wait()** then checks the condition (by calling the supplied lambda function), and returns if it is satisfied. If the condition is not satisfied, **wait()** unlocks the mutex and puts the thread in a “waiting” state. When the condition variable is notified by a call to **notify_one()** from the data preparation thread, the thread wakes from its slumber, re-acquires the lock on the mutex, and checks the condition again, returning from **wait()** with the mutex still locked if the condition has been satisfied. If the condition has not been satisfied, the thread unlocks the mutex and resumes waiting. This is why we need the **std::unique_lock** rather than the **std::lock_guard** — the waiting thread must unlock the mutex whilst it is waiting , and lock it again afterwards, and **std::lock_guard** does not provide that flexibility. If the mutex remained locked whilst the thread was sleeping, the data preparation thread would not be able to lock the mutex to add an item to the queue, and the waiting thread would never be able to see its condition satisfied.

Listing 4.1 uses a simple lambda function for the wait (#5) which checks to see if the queue is not empty, but any function or callable object could be passed: if you already have a function to check the condition (perhaps because it is more complicated than a simple test like this), then this function can be passed in directly; there is no need to wrap it in a lambda. During a call to **wait()**, a condition variable may check the supplied condition any number of times, however it always does so with the mutex locked, and will return immediately if (and only if) the function provided to test the condition returns true. This means that it is not advisable to use a function with side effects for the condition check.

The flexibility to unlock a **std::unique_lock** is not just used for the call to **wait()**, it is also used once we've got the data to process, but before processing it (#6): processing data can potentially be a time-consuming operation, and as we saw in chapter 3, it is a bad idea to hold a lock on a mutex for longer than necessary.

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

4.1.2 Building a Thread-safe Queue with Condition Variables

If we're going to be designing a generic queue, it's worth spending a few minutes thinking about the operations that are likely to be required, like we did with the thread-safe stack back in section 3.2.3. Let's look at the C++ Standard Library for inspiration, in the form of the **std::queue<>** container adaptor shown in listing 4.2.

Listing 4.2: std::queue interface

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

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

    queue& operator=(queue&& q);
    void swap(queue&& q);

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

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

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

If we ignore the construction, assignment and swap operations, we're left with three groups of operations: those that query the state of the whole queue (**empty()** and **size()**), those that query the elements of the queue (**front()** and **back()**), and those that modify the queue (**push()** and **pop()**). This is the same as we had back in section 3.2.3 for the stack, and therefore we've got the same issues regarding race conditions inherent in the interface. Consequently, we need to combine **front()** and **pop()** into a single function call, much as we combined **top()** and **pop()** for the stack. The code from listing 4.1 adds a new nuance, though: when using a queue to pass data between threads, the receiving thread often needs to wait for the data. Let us therefore provide two variants on **pop()**: **try_pop()** which tries to pop the value from the queue, but always returns immediately (with an indication of failure) even if there wasn't a value to retrieve, and **wait_and_pop()** which will wait until there is a value to retrieve. If we take our lead for the signatures from the stack example, our interface looks like listing 4.3.

Listing 4.3: The interface of our thread-safe queue

```
    template<typename T>
    class queue
    {
    public:
        queue();
        queue(const queue&);
        queue& operator=(const queue&) = delete;          #1

        void push(T new_value);

        bool try_pop(T& value);                           #2
        std::shared_ptr<T> try_pop();                     #3

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

        bool empty() const;
    };
```

Cueballs in code and text

#1 Disallow assignment for simplicity
#3 Return NULL if there is no value to retrieve

As we did for the stack, we've cut down on the constructors, and eliminated assignment in order to simplify the code. We've also provided two versions of both **try_pop()** and **wait_for_pop()**, as before. The first overload of **try_pop()** (#2) stores the retrieved value in the referenced variable, so it can use the return value for status: it returns **true** if it retrieved a value, and **`false`** otherwise. The second overload (#3) cannot do this, as it returns the retrieved value directly. However, the returned pointer can be set to **`NULL`** if there is no value to retrieve.

So, how does all this relate back to listing 4.1? Well, we can extract the code for **`push()`** and **`wait_and_pop()`** from there, as shown in listing 4.4:

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

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

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

queue<data_chunk> data_queue;                #1

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

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

#1 We only need our new `queue` now, as the mutex and condition variable are contained within
#2 No external synchronization required for `push()`
#3 `wait_and_pop()` takes care of the condition variable wait

The other overload of **`wait_and_pop()`** is now trivial to write, and the remaining functions can be copied almost verbatim from the stack example in listing 4.4. The final queue implementation is shown in listing 4.5.

Listing 4.5: The full class definition for a thread-safe queue using condition variables

```
template<typename T>
class queue
{
private:
    mutable std::mutex mut;                    #1
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    queue()
    {}
    queue(queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }

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

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

    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
```

```
            data_cond.wait(lk,[]{return !data_queue.empty();});
            std::shared_ptr<T> res(new T(data_queue.front()));
            data_queue.pop();
            return res;
        }

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

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

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

#1 The mutex must be `mutable` so we can lock it in `empty()` and in the copy constructor

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

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

If the waiting thread is only ever going to wait the once, so when the condition is true it will never wait on this condition variable again, a condition variable might not be the best choice of synchronization mechanism. This is especially true if the condition being waited for is the availability of a particular piece of data. In this scenario, a *future* might be more appropriate.

4.2 Waiting for One-off Events with Futures

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

The C++ Standard Library models this sort of one-off event with something called a *future*. If a thread needs to wait for a specific one-off event, then it somehow obtains a future representing this event. The thread can then poll the future to see if the event has occurred (check the departures board), whilst performing some other task (eating in the overpriced café), or it can just wait for the event to happen. A future may have data associated with it (such as which gate your flight is boarding at), or it may not. Once an event has happened (and the future has become *ready*), then the future cannot be reset.

There are two sorts of futures in the C++ Standard Library, implemented as two class templates: *unique futures* (**std::unique_future<>**), and *shared futures* (**std::shared_future<>**). These are modelled after **std::unique_ptr** and **std::shared_ptr**: an instance of **std::unique_future** is the one and only instance that refers to its associated event, whereas multiple instances of **std::shared_future** may refer to the same event. In the latter case, all the instances will of course become *ready* at the same time, and they may all access any data associated with the event. This associated data is the reason these are templates: just like **std::unique_ptr** and **std::shared_ptr**, the template parameter is the type of the associated data. The **std:unique_future<void>** and **std::shared_future<void>** template specializations should be used where there is no associated data.

Listing 4.6 models a couple of passengers waiting for their flight as threads in some C++ code. The first passenger thread, running **wait_for_flight1()**, just obtains a

std::shared_future<boarding_information> with the boarding information (#1), and calls **get()**, which waits for the future to become ready before returning the associated data. The second passenger thread, running **wait_for_flight2()** isn't content to just sit and wait. Instead, after obtaining the future (#2), this passenger gets on with eating in the café and buying duty-free goods (#3), whilst periodically checking to see if the flight is ready to board by calling **is_ready()** on the future (#4).

Listing 4.6: Waiting for an event with a std::unique_future

```
void wait_for_flight1(flight_number flight)
{
    std::shared_future<boarding_information>  #1
        boarding_info=get_boarding_info(flight);
    board_flight(boarding_info.get());
}

void wait_for_flight2(flight_number flight)
{
    std::shared_future<boarding_information>  #2
        boarding_info=get_boarding_info(flight);
    while(!boarding_info.is_ready())          #4
    {
        eat_in_cafe();                        #3
        buy_duty_free_goods();
    }
    board_flight(boarding_info.get());
}
```

Cueballs in code and preceding text

This code uses **std::shared_future** rather than **std::unique_future**, since more than one passenger thread will be waiting on any given flight: all passengers need to be notified at the same time, and all need the same boarding information. In other cases, there will only be one thread waiting for a particular future — this is common where the future represents the completion of a sub-task in a parallel algorithm, for example — and in those cases, **std::unique_future** is a better fit. **std::unique_future** should be your first choice wherever it is appropriate, as its simpler requirements offer the potential for a lower overhead than **std::shared_future**. Also, whereas **std::shared_future** only allows you to *copy* the data associated with the future, **std::unique_future** allows you to *move* the data out, transferring ownership from the future into the waiting code. This can be important where the data is potentially expensive to copy, but cheap to move (such as a **std::vector** with a million elements), and is vital where the data *cannot* be copied, only moved (such as **std::thread**, or **std::ifstream**).

Such an example is shown in listing 4.7: in this case, the future holds the response from a request to download a (potentially large) file in a background thread (#1). The use of the future allows this thread to continue processing GUI messages in the mean time (#3). Once the future is ready (#2), the file data can be extracted from the future (#4) and then processed (#5). Because this is a subsidiary task, we know there's only the one thread waiting, so **std::unique_future** is appropriate. **unique_future<>::get()** returns an rvalue, so we can also avoid copying the data when we extract it from the future if **file_contents** has a move constructor.

listing 4.7: Using std::unique_future to transfer a large block of data

```
void process_remote_file(
    std::string const& filename)
{
    std::unique_future<
        file_contents> contents=
        server.download_file_in_background(    #1
            filename);
    while(!contents.is_ready())                #2
    {
        process_gui_events();                  #3
    }
    file_contents file_data(contents.get());   #4
    process_file(file_data);                   #5
}
```

Cueballs in code and preceding text

So, that's futures from the point of view of the waiting thread, but what about the thread that triggers the event? How do we make a future ready? How do we store the associated data? The C++ Standard Library provides the answers to these questions in the form of two function templates: **std::packaged_task<>** and **std::promise<>**.

Packaged Tasks

std::packaged_task<> ties a future to the result of a function call: when the function completes, the future is *ready*, and the associated data is the value returned by the function. This is ideal for overall operations that can be subdivided into a set of self-contained tasks: these tasks can be written as functions, packaged up so each task is associated with a future using **std::packaged_task**, and then executed on separate threads. The driving function can then wait for the futures before processing the results of these subtasks.

Promises

std::promise<> is a slightly lower-level tool that provides explicit functions for setting the data associated with a future, and making the future *ready*. This is good for the case where the data may come from multiple sources: in a parallel search algorithm, for example, the search space may be divided between threads, but only one result is required. A **std::unique_future**/**std::promise** pair can be used to manage the return value, and the first thread to find a match can set the future value through the **std::promise** instance, and instruct the other threads to stop searching.

4.2.1 Associating a Task with a Future

Let's look back at the example in listing 4.7. The details of creating the **std::unique_future** that will contain the contents of the downloaded file are wrapped up in the **server** member function **download_file_in_background()**. If it was not a background task but a foreground task, this function would probably be something along the lines of the listing 4.8: open a connection, request the file, receive chunks of data until done, and return the result to the caller.

Listing 4.8: Downloading a file from a remote server

```
file_contents download_file(std::string const& filename)
{
    Connection con(remote_server);
    con.send_file_request(filename);
    file_contents contents;
    while(!con.eof())
    {
        std::vector<unsigned char> chunk=
            con.receive_data_chunk();
        contents.append(chunk);
    }
    return contents;
}
```

If we already have a function such as **download_file()**, or we're willing to write one, writing **download_file_in_background()** becomes remarkably easy to write using **std::packaged_task**: wrap **download_file()** in a **std::packaged_task** (#1), using **std::bind()** to pass in the name of the file to download, get the associated future (#2), start a thread to run the packaged task and finally transfer the future back to the caller.

```
std::unique_future<file_contents>
    download_file_in_background(
        std::string const& filename)
{
    std::packaged_task<file_contents()>     #1
        task(std::bind(download_file,filename)));
    std::unique_future<file_contents>       #2
        res(task.get_future());
    std::thread(std::move(task));           #3
    return res;                             #4
}
```

Cueballs in code and preceding and following text

Note the use of **`std::move()`** when starting the thread (#3): **`std::packaged_task`** cannot be copied, so the ownership of the task must be explicitly transferred into the **`std::thread`** object. Even though **`std::unique_future`** cannot be copied either, since the value returned (#4) is a local variable (#2), the ownership of the result is implicitly transferred into the function return value.

What about those tasks that can't be expressed as a simple function call, or those tasks where the result may come from more than one place? As already mentioned in the introduction to section 4.2, **`std::promise`** provides a better fit for these cases than **`std::packaged_task`**.

4.2.2 Making (std::)Promises

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

Consider one of these threads handling the connections. Data packets will come in from the various connections being handled in essentially random order, and likewise data packets will be queued to be sent in random order. In many cases, other parts of the application will be waiting either for data to be successfully sent, or for a new batch of data to be successfully received via a specific network connection. In these cases, we could use a **`std::promise`**/**`std::unique_future`** pair to identify the successful transmission of a block of data, in which case the data associated with the future would be a simple “success/failure” flag, or this could be rolled into the future itself: if it's ready with a value (any value), it's a success, if it's ready with an exception, it's a failure. For incoming packets, the data associated with the future would be the data packet itself. Listing 4.9 shows an example of this scenario.

Listing 4.9: Handling Multiple Connections from a Single Thread using Promises

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

Cueballs in code and text

The function **process_connections()** loops until **done()** returns **true** (#1). Every time through the loop, it checks each connection in turn (#2), retrieving incoming data if there is any (#3) or sending any queued outgoing data (#4). This assumes that an incoming packet has some ID and a payload with the actual data in it. The ID is mapped to a **std::promise** (perhaps by a lookup in an associative container) (#5), and the value set to the packet's payload. For outgoing packets, the packet is retrieved from the outgoing queue, and actually sent through the connection. Once the send has completed, the promise associated with the outgoing data is set to **true** to indicate successful transmission (#6). Whether this maps nicely to the actual network protocol depends on the protocol: this promise/future style structure may not work for a particular scenario, though it does have a similar structure to the asynchronous IO support of some operating systems.

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

4.2.3 Saving an Exception for the Future

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

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

Now suppose that instead of just invoking **square_root()** from the current thread:

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

the call is packaged up and run on another thread:

```
std::packaged_task<double()>
    task(std::bind(square_root,-1));
std::unique_future<double> f=task.get_future();
std::thread task_thread(std::move(task));
double y=f.get();
```

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

Well, that's exactly what happens: when a **std::packaged_task** is invoked, if the wrapped function throws an exception then that exception is stored in the future in place of a stored value, the future becomes *ready*, and a call to **get()** re-throws that stored exception.

Just as you can query whether or not a future **is_ready()**, there are also query functions to identify whether a future has a stored value or a stored exception: **has_value()**, and **has_exception()**. Of course, they can't both return **true**, since a function call either returned a value *or* it threw an exception, but they *can* both return

false: if the future is not ready, then it neither has a stored value, nor has a stored exception.

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

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

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

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

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

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

Another way to store an exception in a future is to destroy the **std::promise** or **std::packaged_task** associated with the future without calling either of the set functions on the promise or invoking the packaged task. In either case, the destructor of the **std::promise** or **std::packaged_task** will store a **std::broken_promise** exception in the future: by creating a future you make a promise to provide a value or exception, and by destroying the source of that value or exception, you've broken that promise. If the compiler didn't store anything in the future in this case, waiting threads could potentially wait forever.

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

4.3 Using Synchronization of Operations to Simplify Code

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

4.3.1 Functional Programming with Futures

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

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

The benefits of Functional Programming are not limited to those languages where it is the default paradigm, however. C++ is a multi-paradigm language, and it is entirely possible to write programs in the FP style. This is even easier in C++0x than it was in C++98, with the advent of lambda functions, the incorporation of **std::bind** from Boost and TR1, and the introduction of automatic type deduction for variables. Futures are the final piece of the puzzle that make FP-style concurrency viable in C++: a future can be passed around between threads to allow the result of one computation to depend on the result of another, *without any explicit access to shared data*.

Listing 4.10 shows an implementation of the Quicksort algorithm using futures and a functional style. Before we start, we take off the first element as the partition value (#1). Then, the input data is partitioned in two (#2), using a lambda function (#3) to find those values less than the **partition_value**, and split off the lower part into its own list (#4). The lower half is sorted as a background task (#5), whilst the remainder (upper half) is sorted by a simple recursive call (#6) in order to make use of the existing thread whilst it is waiting for the background sort to complete. The two parts are then added either side of the partition value (#7, #8) before the result is returned to the caller (#9).

This is only semi-functional: the parameters are passed in by value, and so the input cannot be modified, but the lists are partitioned in-place internally for efficiency. Also, the lists are *moved* into the parameters for the recursive call, and **splice()** is used to assemble the final result in order to avoid constructing unnecessary copies. The call to **get()** (#8) used to retrieve the result of the future will wait until the future is ready, so the final splice can reassemble the two lists in the correct sorted order.

Listing 4.10: Parallel QuickSort using Futures

```
template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    std::list<T> result;
    result.splice(result.begin(),input,input.begin());               #1
    T const& partition_val=*result.begin();

    typename std::list<T>::iterator divide_point=                     #2
        std::partition(input.begin(),input.end(),
            [&](T const& t){return t<partition_val;});               #3

    std::list<T> lower_part;
    lower_part.splice(lower_part.end(),input,input.begin(),
        divide_point);                                                #4

    std::unique_future<std::list<T> > new_lower(                      #5
        spawn_task(&parallel_quick_sort<T>,std::move(lower_part)));

    std::list<T> new_higher(parallel_quick_sort(std::move(input)));  #6

    result.splice(result.end(),new_higher);                           #7
    result.splice(result.begin(),new_lower.get());                    #8
    return result;                                                    #9
}
```

Cueballs in code and preceding text

spawn_task() is not provided as part of the Standard Library, but it can be easily written as a simple wrapper around **std::packaged_task** and **std::thread** similar to the example at the start of section 4.2.1, as shown in listing 4.11: create a **std::packaged_task** for the result of the function call, get the future from it, run it on a thread and return the future. A more sophisticated implementation could add the task to a queue to be run by a pool of worker threads, thus limiting the number of threads created.

Listing 4.11: A sample implementation of spawn_task

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

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

4.3.2 Synchronizing Operations with Message Passing

The idea of Communicating Sequential Processes is simple: if there is no shared data, each thread can be reasoned about entirely independently, purely on the basis of how it behaves in response to the messages that it received. Each thread is therefore effectively a state machine: when it receives a message, it updates its state in some manner, and maybe sends one or more messages to other threads, with the processing performed depending on the initial state. One way to write such threads would be to formalize this and implement a Finite State Machine model, but this is not the only way: the state machine can be implicit in the structure of the application. Which method works better in any given scenario depends on the exact behavioural requirements of the situation, and the expertise of the programming team. However you choose to implement each thread, the separation into independent processes has the potential to remove much of the complication from shared-data concurrency, and therefore make programming easier and thus lower the bug rate.

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

Imagine a mail-order business for a moment. As a customer, you send in your order, typically by fax or through an internet-based ordering service. The business then processes your order and sends your products through the mail. Unless the business is in a very poor state, they don't sit by the fax machine waiting for your order: they've got other orders to process. Likewise you don't sit by the door waiting for the delivery: you get on with your life. This is a basic message passing system: you send a message (your order) to the business, along with enough information (your address) that they can send a message back to you (the delivery of your products). How would this look in C++?

Well, let's start with a pair of classes: one representing the business, and one representing a customer, and start them off doing their activities on separate threads, as in listing 4.12.

Listing 4.12: A mail order business, and a customer

```
class mail_order_business
{
public:
    void run_business();
};
class customer
{
public:
    void live_life();
};

mail_order_business widgets_inc;          #1
customer fred;                            #2

std::thread business_thread(              #3
    &mail_order_business::run_business,
    &widgets_inc);

std::thread customer_thread(              #4
    &customer::live_life,&fred);
```

#1 Our business object
#2 Our customer object
#3 Run the business on a thread
#4 Start a thread for the customer to live his life

This is all very well, but how can we send messages between the threads: how can **fred** place an order with **widgets_inc**? To do this, we need a messaging subsystem. This will then be the sole shared data structure in the application. Firstly, when the business starts up, it registers itself with the messaging subsystem. We then retrieve that ID and pass it in to the customer, when we create the customer. The customer then also registers with the messaging subsystem, and everything is set up. This is shown in listing 4.13.

Listing 4.13: Registering with the messaging subsystem

```
class mail_order_business
{
private:
    messaging::receiver_id id;
public:
    mail_order_business():
        id(messaging::register_receiver()) #1
    {}
    messaging::receiver_id get_id() const
    {
        return id;
    }
    // as before
};

class customer
{
private:
    messaging::receiver_id id;
    messaging::receiver_id business_id;
public:
    explicit
    customer(message::receiver_id business_id_):
        id(messaging::register_receiver()), #2
        business_id(business_id_)
    {}
    // as before
};

mail_order_business widgets_inc;
customer fred(widgets_inc.get_id());        #3
```

#1 Register business with messaging subsystem
#2 Register customer with messaging subsystem
#3 Pass the business ID into the customer

Since **fred** now knows about **widgets_inc**, he can place an order for 3 widgets to be delivered to himself. The business can then respond to that order by retrieving the **order_message** from the queue, and sending a delivery if there's enough widgets available. If there aren't enough widgets, then the order is placed on a “pending orders” list, presumably to be fulfilled when more widgets become available. Listing 4.14 shows how this might work. Of course, the customer would then receive the delivery message, and so on. A sample implementation of this message subsystem is included in Appendix B.

Listing 4.14: Sending an order for widgets through the messaging subsystem

```
void customer::live_life()
{
    // ...
    order_message order;                 #1
    order.deliver_to=id;                 #2
    order.order_widgets(3);

    messaging::send_message(business_id,order);#3
    // ...
}

void mail_order_business::run_business()
{
    // ...
    if(messaging::incoming<order_message>(id))#4
    {
        order_message order=
            messaging::get<order_message>(id);#5
        if(widgets_available()>=
           order.num_widgets())
        {
            delivery_message delivery;
            delivery.from=id;
            delivery.add_widgets(get_widgets(
                order.num_widgets()));
            messaging::send_message(
                order.deliver_to,delivery); #6
        }
        else
        {
            add_to_pending_orders(order);
        }
    }
    // ...
}
```

#1 Create an order
#2 Identify the customer as the recipient
#3 Send the order to the business
#4 Check for an incoming order
#5 Get the order
#6 Send the delivery to the customer

This example is only simple, but it does begin to show the power of the messaging paradigm: **widgets_inc** and **fred** do not share any data except the messaging subsystem, and can communicate freely. Indeed **widgets_inc** doesn't even know of the existence of **fred** until he sends the order. As mentioned earlier, this separation is very helpful when reasoning about the application: the behaviour of a **mail_order_processor** can be evaluated in isolation, without thinking about how a **customer** will behave. The sequence of incoming messages is the only part of the external world that needs to be considered.

4.4 Summary

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

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

5

The C++ Memory Model and Operations on Atomic Types

One of the most important features of the C++0x Standard is something most programmers won't even notice. It's not the new syntax features, nor is it the new library facilities, but the new multi-threading-aware memory model. Without the memory model to define exactly how the fundamental building blocks work, none of the facilities we've covered could be relied upon to work. Of course, there's a reason that most programmers won't notice: if you use mutexes to protect your data, and condition variables or futures to signal events, the details of *why* they work aren't important. It's only when you start trying to get “close to the machine” that the precise details of the memory model matter.

However, whatever else it is, C++ is a systems programming language. One of the goals of the Standards committee is that there shall be no need for a lower-level language than C++. Programmers should be provided with enough flexibility within C++ to do whatever they need without the language getting in the way, allowing them to get “close to the machine” when the need arises. The atomic types and operations allow just that, providing facilities for low-level synchronization operations that will commonly reduce to one or two CPU instructions.

In this chapter, I will start by covering the basics of the memory model, and then move onto the atomic types and operations, and finally cover the various types of synchronization available with the operations on atomic types. This is quite complex: unless you're planning on writing code that uses the atomic operations for synchronization (such as the lock-free data structures in chapter 7), you won't need to know these details.

Let's ease into things with a look at the basics of the memory model.

5.1 C++ Memory Model Basics

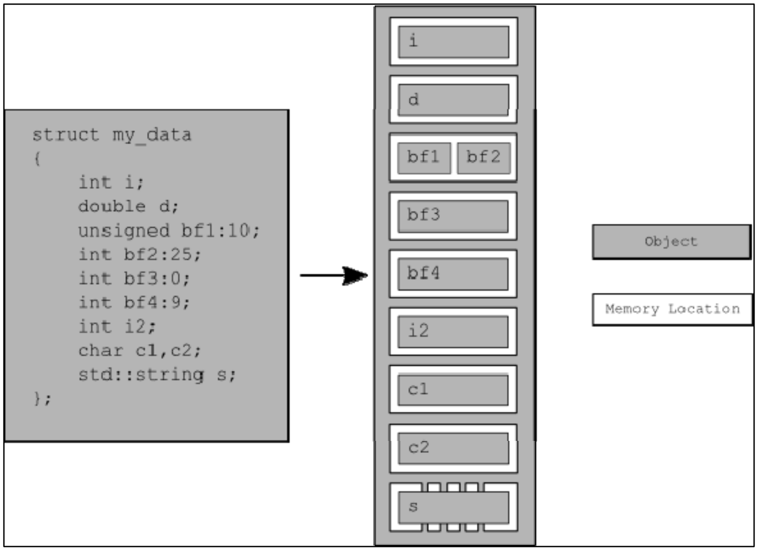
There are two aspects to the memory model: the basic *structural* aspects which relate to how things are laid out in memory, and then the *concurrency* aspects. The structural aspects are important for concurrency, especially when you're looking at low level atomic operations, so we'll start with those. In C++, it's all about objects and memory locations.

5.1.1 Objects and Memory Locations

All data in a C++ program is made up of *objects*. This is not to say that you can create a new class derived from **`int`**, or that the fundamental types have member functions, or any of the other consequences often implied when people say “everything is an object” when discussing a language like Smalltalk or Ruby: it is just a statement about the building blocks of data in C++. The C++ Standard defines an object as “a region of storage”, though it goes on to assign properties to these objects, such as their type and lifetime.

Some of these objects are simple values of a fundamental type such as **`int`** or **`float`**, whilst others are instances of user-defined classes. Some objects (such as arrays, instances of derived classes and instances of classes with non-**`static`** data members) have sub-objects, whilst others do not.

Whatever its type, an object is stored in one or more *memory locations*. Each such *memory location* is either an object (or sub-object) of a scalar type such as **`unsigned short`** or **`my_class*`** or a sequence of adjacent bit-fields. If you use bit fields, this is an important point to note: though adjacent bit-fields are distinct objects they still comprise the same memory location. Figure 5.1 shows how a **`struct`** divides into objects and memory locations.



```
struct my_data
{
    int i;
    double d;
    unsigned bf1:10;
    int bf2:25;
    int bf3:0;
    int bf4:9;
    int i2;
    char c1,c2;
    std::string s;
};
```

Figure 5.6 The division of a **`struct`** into objects and memory locations

Firstly, the entire **`struct`** is one object, which consists of several sub-objects, one for each data member. The bitfields **`bf1`** and **`bf2`** share a memory location, and the **`std::string`** object **`s`** consists of several memory locations internally, but otherwise each member has its own memory location. Note how the zero-length bitfield **`bf3`** separates **`bf4`** into its own memory location.

There are four important things to take away from this:

5. Every variable is an object, including those that are members of other objects;

6. Every object comprises *at least one* memory location;

7. Variables of fundamental type such as **`int`** or **`char`** are *exactly one* memory location, whatever their size, even if they are adjacent, or part of an array;

8. Adjacent bitfields are part of the same memory location.

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

5.1.2 Objects, memory locations and concurrency

Now, here's the part that's crucial for multi-threaded applications in C++: everything hinges on those *memory locations*. If two threads access *separate* memory locations, there is no problem: everything works fine. However, 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. However, 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 is 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 threads. The use of atomic operations to enforce an ordering is described in section 5.3. Of course, if more than two threads access the same memory location, each pair of accesses must have a defined ordering.

If there is no enforced ordering between two accesses to a single memory location from separate threads, one or both of those accesses is not *atomic*, and one or both is a write, this is a *data race*, and causes undefined behaviour.

This statement is crucially important: undefined behaviour is one of the nastiest corners of C++. According to the language standard, once an application contains any undefined behaviour all bets are off: the behaviour of the complete application is now undefined, and it may do anything at all. I know of one case where a particular instance of undefined behaviour caused someone's monitor to set on fire. Though this is rather unlikely to happen to you, a *data race* is definitely a serious bug, and should be avoided at all costs.

There's another important point in that highlighted statement — you can also avoid the undefined behaviour by using *atomic* operations to access the memory location involved in the race. This doesn't prevent the race itself — which of the atomic operations touches the memory location first is still not specified — but it does bring the program back into the realm of defined behaviour.

So, what constitutes an atomic operation, and how can these be used to enforce ordering?

5.2 Atomic Operations and Types in C++

An *atomic operation* is an indivisible operation: you cannot observe such an operation half-done from any thread in the system; it is 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 either retrieve the initial value of the object, or the value stored by one of the modifications.

The flip side of this is that a non-atomic operation might be seen 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 non-atomic operation is a load, it might retrieve part of the object, and then 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 just 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 behaviour.

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

5.2.1 The Standard Atomic Types

The Standard *atomic types* can be found in the **<cstdatomic>** header. All operations on such types are *atomic*, and *only* operations on these types are atomic in the sense of the language definition, though 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 or not operations on a given object 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**).

The only type which doesn't provide an **is_lock_free()** member function is **std::atomic_flag**. This type is a really simple boolean flag, and operations on this type are *required* to be lock free: once you have a simple lock-free boolean flag, you can use that to implement a simple lock, and thus implement all the other atomic types using that as a basis. When I said really simple, I meant it: objects of type **std::atomic_flag** are initialized to clear, and they can then either be queried-and-set (with the **test_and_set()** member function), or cleared (with the **clear()** member function). That's it. No assignment, no copy-construction, no test-and-clear, no other operations at all.

The remaining atomic types are a bit more full-featured, but may not be lock-free (as explained above). Of course, on most popular platforms it is expected that the atomic variants of all the built-in types are indeed lock free, but it is not required. These types, and the built-in types they represent are shown in table 5.1.

Table 5.1 The Standard atomic types and their corresponding built-in types

Atomic type	Corresponding built-in type
atomic_bool	**bool**
atomic_char	**char**

atomic_schar	**signed char**
atomic_uchar	**unsigned char**
atomic_int	**int**
atomic_uint	**unsigned int**
atomic_short	**short**
atomic_ushort	**unsigned short**
atomic_long	**long**
atomic_ulong	**unsigned long**
atomic_llong	**long long**
atomic_ullong	**unsigned long long**
atomic_char16_t	**char16_t**
atomic_char32_t	**char32_t**
atomic_wchar_t	**wchar_t**
atomic_address	**void***

As well as the basic atomic types, the C++ Standard Library also provides a set of typedefs for the atomic types corresponding to the various non-atomic Standard Library typedefs such as **std::size_t**. These are shown in table 5.2.

Table 5.2 The Standard atomic typedefs and their corresponding built-in typedefs

Atomic typedef	**Corresponding Standard Library typedef**
atomic_int_least8_t	**int_least8_t**
atomic_uint_least8_t	**uint_least8_t**
atomic_int_least16_t	**int_least16_t**

atomic_uint_least16_t	**uint_least16_t**
atomic_int_least32_t	**int_least32_t**
atomic_uint_least32_t	**uint_least32_t**
atomic_int_least64_t	**int_least64_t**
atomic_uint_least64_t	**uint_least64_t**
atomic_int_fast8_t	**int_fast8_t**
atomic_uint_fast8_t	**uint_fast8_t**
atomic_int_fast16_t	**int_fast16_t**
atomic_uint_fast16_t	**uint_fast16_t**
atomic_int_fast32_t	**int_fast32_t**
atomic_uint_fast32_t	**uint_fast32_t**
atomic_int_fast64_t	**int_fast64_t**
atomic_uint_fast64_t	**uint_fast64_t**
atomic_intptr_t	**intptr_t**
atomic_uintptr_t	**uintptr_t**
atomic_size_t	**size_t**
atomic_ssize_t	**ssize_t**
atomic_ptrdiff_t	**ptrdiff_t**
atomic_intmax_t	**intmax_t**
atomic_uintmax_t	**uintmax_t**

That's a lot of types! Of course, there's rather a simple pattern to it: for a standard typedef **T**, the corresponding atomic type is just the same name with an **atomic_** prefix: **atomic_T**.

The same applies to the built-in types, except that **signed** is abbreviated as just **s**, **unsigned** as just **u**, and **long long** as **llong**. The only peculiarity is **std::atomic_address**, but that's easy to remember as an exception.

The Standard atomic types are not copyable or assignable in the conventional sense, in that they have no copy constructors or copy assignment operators. However, they *do* support assignment from and implicit conversion to the corresponding built-in types, as well as direct **load()** and **store()** member functions, **exchange()**, **compare_exchange_weak()** and **compare_exchange_strong()**. They also support the compound assignment operators where appropriate: **+=, -=, *=, |=,** etc., and the integral types support **++** and **--**. These operators also have corresponding named member functions with the same functionality: **fetch_add()**, **fetch_or()**, etc. The return value from the assignment operators and member functions is either the value stored (in the case of the assignment operators) or the value prior to the operation (in the case of the named functions). This avoids the potential problems that could stem from the usual habit of such assignment operators returning a reference to the object being assigned to: in order to get the stored value from such a reference, the code would have to perform a separate read, thus allowing another thread to have modified the value between the assignment and the read, and opening the door for a race condition.

In addition to these atomic types, the Standard Library also provides a generic **std::atomic<>** class template which can be used to create an atomic variant of a user-defined type. Because it's a generic class template, the operations are limited to **load()**, **store()** (and assignment from and conversion to the user-defined type), **exchange(), compare_exchange_weak()** and **compare_exchange_strong()**. There are specializations for integral types which derive from the corresponding **std::atomic_***integral-type* type (see table 5.1), and there are specializations for pointer types that allow pointer arithmetic. Thus **std::atomic<int>** is derived from **std::atomic_int** and **std::atomic<std::string*>** is derived from **std::atomic_address**. This allows you to use the extended interface of the basic atomic types with the corresponding **std::atomic<>** specializations.

Each of the operations on the atomic types has an optional memory ordering argument which can be used to specify the required memory ordering semantics. The precise semantics of the memory ordering options are covered in section 5.3. For now it suffices to know that the operations are divided into three categories:

- *store* operations, which can have **memory_order_relaxed**, **memory_order_release** or **memory_order_seq_cst** ordering;
- *load* operations, which can have **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire** or **memory_order_seq_cst** ordering; and

- *read-modify-write* operations, which can have **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel** or **memory_order_seq_cst** ordering.

The default ordering for all operations is **memory_order_seq_cst**.

Let's now look at the operations you can actually do on each of the Standard atomic types, starting with **std::atomic_flag**.

5.2.2 Operations on *std::atomic_flag*

std::atomic_flag is the simplest Standard atomic type, which represents a boolean flag. It is deliberately basic, and is intended as a building block only. As such, I would never expect to see it in use, except under very specialist circumstances. However, it will serve as a starting point for discussing the other atomic types, since it shows some of the general policies that apply to the atomic types.

Objects of type **std::atomic_flag must** be initialized with **ATOMIC_FLAG_INIT**. This initializes the flag to a *clear* state. There is no choice in the matter: the flag always starts *clear*.

```
std::atomic_flag f=ATOMIC_FLAG_INIT;
```

This applies wherever the object is declared, and whatever scope it has. It is the only atomic type to require such special treatment for initialization, but it is also the only type guaranteed to be lock free. If the **std::atomic_flag** object has static storage duration, it is guaranteed to be statically initialized, which means that there are no initialization-order issues: it will always be initialized by the time of the first operation on the flag.

Once you've got your flag object initialized, there are now only three things you can do with it: destroy it, clear it, or set it and query the previous value. These correspond to the destructor, the **clear()** member function, and the **test_and_set()** member function respectively. Both the **clear()** and **test_and_set()** member functions can have a memory order specified. **clear()** is a *store* operation, so cannot have **memory_order_acquire** or **memory_order_acq_rel** semantics, but **test_and_set()** is a read-modify-write operation, so can have any of the memory ordering tags applied. As with every atomic operation, the default for both is **memory_order_seq_cst**.

```
f.clear(std::memory_order_release);         #1
bool x=f.test_and_set();                    #2
```

#1 Clear the flag with release semantics
#2 Set the flag with the default (sequentially consistent) memory ordering, and retrieve the old value

You cannot copy-construct another **std::atomic_flag** object from the first, and you cannot assign one **std::atomic_flag** to another. This is not something peculiar to **std::atomic_flag**, but something common to all the atomic types. All operations on an atomic type are defined to be atomic, and assignment and copy-construction involve two objects. A single operation on two distinct objects cannot be atomic: in the case of copy-construction or copy-assignment, the value must first be read from one object and then written to the other. This is two separate operations on two separate objects, and the combination cannot be atomic. Therefore, these operations are not permitted.

The limited feature set makes **std::atomic_flag** ideally suited to use as a spin-lock mutex. Initially the flag is clear and the mutex is unlocked. To lock the mutex, loop on **test_and_set()** until the old value was **false**, indicating that *this* thread set the value to **true**. Unlocking the mutex is simply a matter of clearing the flag. Such an implementation is shown in listing 5.1.

Listing 5.14: Implementation of a spinlock mutex using std::atomic_flag.

```
class spinlock_mutex
{
    std::atomic_flag flag;
public:
    spinlock_mutex():
        flag(ATOMIC_FLAG_INIT)
    {}
    void lock()
    {
        while(flag.test_and_set(std::memory_order_acquire));
    }
    void unlock()
    {
        flag.clear(std::memory_order_release);
    }
};
```

Such a mutex is very basic, but it is enough to use with **std::lock_guard<>** (see chapter 3). By its very nature it does a busy-wait in **lock()**, so it is a poor choice if there is expected to be any degree of contention, but it is enough to ensure mutual exclusion. When we look at the memory ordering semantics, we'll see how this guarantees the necessary enforced ordering that goes with a mutex lock.

std::atomic_flag is so limited that it can't even be used as a general boolean flag, since it doesn't have a simple non-modifying query operation. For that we're better off using **std::atomic_bool**, so we'll cover that next.

5.2.3 Operations on std::atomic_bool

The most basic of the atomic integral types is **std::atomic_bool**. This is a more full-featured boolean flag than **std::atomic_flag**, as you might expect. Though it is still not copy-constructible or copy-assignable, you can construct it from a non-atomic **bool**, so it can be initially **true** or **false**, and you can also assign from a non-atomic **bool**.

```
std::atomic_bool b=true;
```

```
    b=false;
```

One other thing to note about the assignment operator from a non-atomic **bool** is that it differs from the general convention of returning a reference to the object assigned to: it returns a **bool** with the value assigned instead. This is another common pattern with the atomic types: the assignment operators they support return values (of the corresponding non-atomic type) rather than references. If a reference to the atomic variable was returned, any code that depended on the result of the assignment would then have to explicitly load the value, potentially getting the result of a modification by another thread. By returning the result of the assignment as a non-atomic value, this additional load can be avoided, and you know that the value obtained is the actual value stored.

Rather than the restrictive **clear()** function of **std::atomic_flag**, writes (of either **true** or **false**) are done by calling **store()**, though the memory order semantics can still be specified. Similarly, **test_and_set()** has been replaced with the more general **exchange()** member function that allows you to replace the stored value with a new one of your choosing, and atomically retrieve the original value. **std::atomic_bool** also supports a plain non-modifying query of the value with an implicit conversion to plain **bool**, or with an explicit call to **load()**. As you might expect, **store()** is a *store* operation, whilst **load()** is a *load* operation. **exchange()** is a *read-modify-write* operation.

```
    std::atomic_bool b;
    bool x=b.load(std::memory_order_acquire);
    b.store(true);
    x=b.exchange(false,std::memory_order_acq_rel);
```

exchange() isn't the only read-modify-write operation supported by **std::atomic_bool**: it also introduces an operation to store a new value if the current value is equal to an “expected” value.

Storing a new value (or not) depending on the current value

This new operation is called compare/exchange, and comes in the form of the **compare_exchange_weak()** and **compare_exchange_strong()** member functions. The compare/exchange operation is the cornerstone of programming with atomic types: it compares the value of the atomic variable with a supplied “expected” value and stores the supplied “desired” value if they are equal. If the values are not equal, the “expected” value is updated with the actual value of the atomic variable. The return type of the compare/exchange functions is a **bool** which is **true** if the store was performed, and **false** otherwise.

For **compare_exchange_weak()**, the store might not be successful even if the original value was equal to the “expected” value, in which case the value of the variable is unchanged and the return value of **compare_exchange_weak()** is **false**. This is most likely to happen on machines which lack a single compare-and-exchange instruction, if the processor cannot guarantee that the operation has been done atomically — possibly because the thread performing the operation was switched out in the middle of the necessary sequence of instructions, and another thread scheduled in its place by the operating system where there are more threads than processors. This is called a “spurious” failure, since the reason for the failure is a function of timing rather than the values of the variables.

Because **compare_exchange_weak()** can fail spuriously, it must typically be used in a loop:

```
bool expected=false;
extern atomic_bool b; // set somewhere else
while(!b.compare_exchange_weak(expected,true) && !expected);
```

In this case, we keep looping as long as **expected** is still **false**, indicating that the **compare_exchange_weak()** call failed spuriously.

On the other hand, **compare_exchange_strong()** is guaranteed to only return **false** if the actual value was not equal to the **expected** value. This can eliminate the need for loops like the above where we just want to know whether we successfully changed a variable, or whether another thread got there first.

If we want to change the variable whatever the initial value (perhaps with an updated value that depends on the current value), the update of **expected** becomes useful: each time round the loop, **expected** is reloaded, so if no other thread modifies the value in the mean time, the **compare_exchange_weak()** or **compare_exchange_strong()** call should be successful the next time round the loop. If the calculation of the value to be stored is simple, it may be beneficial to use **compare_exchange_weak()** in order to avoid a double-loop on platforms where **compare_exchange_weak() can** fail spuriously (and so **compare_exchange_strong()** contains a loop), but if the calculation of the value to be stored is itself time consuming, then it may make sense to use **compare_exchange_strong()** to avoid having to recalculate the value to store when the **expected** value hasn't changed. For **std::atomic_bool** this isn't so important — there are only two possible values after all — but for the larger atomic types this can make a difference.

The compare/exchange functions are also unusual in that they can take **two** memory ordering parameters. This allows for the memory ordering semantics to differ in the case of success and failure: it might be desirable for a successful call to have **memory_order_acq_rel** semantics whilst a failed call has **memory_order_relaxed** semantics. A failed compare/exchange did not do a store, so it cannot have **memory_order_release** or **memory_order_acq_rel** semantics. It is therefore not permitted to supply these values as the ordering for failure. You also can't supply stricter memory ordering for failure than success: if you want **memory_order_acquire** or **memory_order_seq_cst** semantics for failure, you must specify those for success as well.

If you do not specify an ordering for failure, it is assumed to be the same as that for success, except that the **release** part of the ordering is stripped: **memory_order_release** becomes **memory_order_relaxed**, and **memory_order_acq_rel** becomes **memory_order_acquire**. If you specify neither, they default to **memory_order_seq_cst** as usual. The following two calls to **compare_exchange_weak()** are equivalent:

```
std::atomic_bool b;
bool expected;
b.compare_exchange_weak(expected,true,
    memory_order_acq_rel,memory_order_acquire);
b.compare_exchange_weak(expected,true,memory_order_acq_rel);
```

I'll leave the consequences of the choice of memory ordering to section 5.3.

One further difference between **std::atomic_bool** and **std::atomic_flag** is that **std::atomic_bool** may not be lock-free: the implementation may have to acquire a mutex internally in order to ensure the atomicity of the operations. For the rare case that this matters, you can use the **is_lock_free()** member function to check whether or not operations on a particular instance of **std::atomic_bool** are lock free. Note that this is *per-instance* since it may be possible for operations on some instances (such as objects with static storage duration) to be lock free, but not others. This is another feature common to all atomic types other than **std::atomic_flag.**

The next-simplest of the atomic types is **std::atomic_address**, so we'll look at that next.

5.2.4 Operations on *std::atomic_address*

std::atomic_address has a lot in common with **std::atomic_bool** in terms of its interface. Just like **std::atomic_bool**, it is neither copy-constructible nor copy-assignable, though it can be both constructed and assigned from the associated non-atomic built-in type: in this case **void***. As well as the obligatory **is_lock_free()** member function, **std::atomic_address** also has **load()**, **store()**, **exchange()**, **compare_exchange_weak()** and **compare_exchange_strong()** member functions, with similar semantics to those of **std::atomic_bool**, except taking and returning **void*** rather than **bool**.

The new, and slightly idiosyncratic, operations provided by **std::atomic_address** are **fetch_add** and **fetch_sub**, which do atomic addition and subtraction on the stored address, and the operators **+=** and **-=** which provide convenient wrappers. These are slightly idiosyncratic because **std::atomic_address** fundamentally represents a **void***, and you cannot do pointer arithmetic on a **void***. However, they are defined to add and subtract a single byte, as if the stored pointer was an **unsigned char***. This makes sense when you think about the memory structure of C++: the object representation of an object of type **T** is the sequence of **sizeof(T) unsigned char** values that make up the memory locations for that object.

Anyway, **fetch_add** adds the specified offset to the pointer, and **fetch_sub** subtracts the offset from the pointer. In both cases they return the *original unmodified address*, not the new one. In contrast, the assignment operators **+=** and **-=** return the new value in an attempt to be as close to the semantics of a conventional assignment operator; they still return a **void*** by value rather than a reference to the **std::atomic_address** object, though.

```
    char some_array[5];
    std::atomic_address p(some_array);
    void* x=p.fetch_add(2);              #1
    assert(x==some_array);
    assert(p.load()==&some_array[2]);
    x=(p-=1);                            #2
    assert(x==&some_array[1]);
    assert(p.load()==&some_array[1]);
```

#1 Add two to p and return the old value
#2 Subtract one from p and return the new value

If you use **std::atomic_address** to index into arrays of other types, you need to be ensure that the appropriate increment is used: whereas pointer arithmetic on normal pointers will increase or decrease the pointer by the specified number of *objects*, arithmetic on **std::atomic_address** increases or decreases the pointer by the specified number of *bytes*.

```
    std::string my_strings[23];
    std::atomic_address ps(my_strings);
    ps.fetch_add(5*sizeof(std::string));
    assert(ps.load()==&my_strings[5]);
```

The function forms also allow the memory ordering semantics to be specified as an additional function call argument:

```
    p.fetch_add(3,std::memory_order_release);
```

Since both **fetch_add** and **fetch_sub** are read-modify-write operations, they can have any of the memory ordering tags, and can participate in a *release sequence*. Specifying the ordering semantics is not possible for the operator forms, since there is no way of providing the information: these forms therefore always have **memory_order_seq_cst** semantics.

The remaining basic atomic types are essentially all the same: they're all atomic integral types, and have the same interface as each other, except the associated built-in type is different. We'll look at them as a group.

5.2.5 Operations on Standard Atomic Integral Types

As well as the “usual” set of operations, the atomic integral types have quite a comprehensive set of operations available: **fetch_add**, **fetch_sub**, **fetch_and**, **fetch_or**, **fetch_xor**, compound-assignment forms of these operations (**+=**, **-=**, **&=**, **|=**, and **^=**) and pre- and post- increment and decrement. It's not quite the full set of compound-assignment operations you could do on a normal integral type, but it's close enough: only division, multiplication and shift operators are missing. Since atomic integral values are typically used either as counters or as bit masks, this is not a particularly noticeable loss.

The semantics match closely to that of **fetch_add** and **fetch_sub** for **std::atomic_address**: the named functions atomically perform their operation and return the *old* value, whereas the compound-assignment operators return the *new* value. Pre- and post- increment and decrement work as usual: **++x** increments the variable and returns the new value, whereas **x++** increments the variable and returns the old value. As you'll be expecting by now, the result is a value of the associated integral type in both cases.

We've now looked at all the basic atomic types; all that remains is the generic **std::atomic** class template, so let's look at that next.

5.2.6 The std::atomic class template

This allows a user to create an atomic variant of a user-defined type, as well as providing a uniform interface to the atomic variants of the built-in types. For most fundamental types, there is little difference to using **std::atomic<T>** rather than the corresponding **std::atomic_***integral_type*** type: the former derives from the latter, so **std::atomic<int>** gets all its member functions from **std::atomic_int**, for example. The only difference is with pointers, and pointer arithmetic: **std::atomic<std::string*>** can be used to index through an array of **std::string** objects without having to worry about the **sizeof** issues that arise when using **std::atomic_address** directly.

Though the **std::atomic** template provides the minor benefits listed above when used with the fundamental types, the main benefit to be derived from its use is the possibility of atomic variants of user-defined types.

You can't just use any user-defined type with **std::atomic<>**, though: the type has to fulfil certain criteria. In order to use **std::atomic<UDT>** for some user-defined type **UDT**, this type must have a *trivial* copy-assignment operator. This means that the type must not have any virtual functions or virtual base classes, and must use the compiler-generated copy-assignment operator. Not only that, but every base class and non-**static** data member of user-defined type must also have a trivial copy-assignment operator. This essentially permits the compiler to use **memcpy()** or an equivalent operation for assignment operations, since there is no user-written code to run.

Finally, the type must be *bitwise equality comparable*, so there can be no user-defined equality comparison operator. This goes alongside the assignment requirements: not only must you be able to copy an object of type **UDT** using **memcpy()**, but you must be able to compare instances for equality using **memcmp()**. This guarantee is required in order for compare/exchange operations to work.

The reasoning behind these restrictions goes back to one of the guidelines from chapter 3: don't pass pointers and references to protected data outside the scope of the lock, by passing them as arguments to user-supplied functions. In general, the compiler is not going to be able to generate lock-free code for **std::atomic<UDT>**, so it will have to use an internal lock for all the operations. If user-supplied copy-assignment or comparison operators were permitted, this would require passing a reference to the protected data as an argument to a user-supplied function, thus violating the guideline above. Also, the library is entirely at liberty to use a single lock for all atomic operations that need it, and allowing user-supplied functions to be called whilst holding that lock might cause deadlock, or cause other threads to block because a comparison operation took a long time. Finally, these restrictions increase the chance that the compiler will be able to make use of atomic instructions directly for **std::atomic<UDT>**, since it can just treat the user-defined type as a set of raw bytes.

These restrictions mean that you can't, for example, create a **std::atomic<std::vector<int>>**, but you can use it with classes containing counters or flags or pointers, or even just arrays of simple data elements. However, this isn't particularly a problem: the more complex the data structure, the more likely you'll want to do operations on it other than simple assignment and comparison. If that's the case, then you're better off using a **std::mutex** to ensure that the data is appropriately protected for the desired operations, as described in chapter 3.

When instantiated with a user-defined type **T**, the interface of **std::atomic<T>** is limited to the set of operations available for **std::atomic_bool**: **load()**, **store()**, **exchange()**, **compare_exchange_weak()**, **compare_exchange_strong()** and assignment from and conversion to an instance of type **T**.

5.2.7 Free functions for atomic operations

Up until now I've limited myself to describing the member function forms of the operations on the atomic types. However, there are also equivalent non-member functions for most operations on the atomic integral types and **std::atomic_address**. These functions are overloaded for each of the atomic types, and where there is opportunity for specifying a memory ordering tag they come in two varieties: one without the tag, and one with an **_explicit** suffix, and an additional parameter or parameters for the memory ordering tag or tags. Whereas the atomic object being referenced by the member functions is implicit, all the free functions take a pointer to the atomic object as the first parameter.

For example, **std::atomic_is_lock_free()** comes in just one variety (though overloaded for each type), and **std::atomic_is_lock_free(&a)** returns the same value as **a.is_lock_free()** for an object of atomic type **a**. Likewise, **std::atomic_load(&a)** is the same as **a.load()**, but the equivalent **of a.load(std::memory_order_acquire)** is **std::atomic_load_explicit(&a, std::memory_order_acquire)**.

The free functions are designed to be C compatible, so they use pointers rather than references in all cases. For example, the first parameter of the **compare_exchange_weak()** and **compare_exchange_strong()** member functions (the expected value) is a reference, whereas the second parameter of **std::atomic_compare_exchange_weak()** (the first is the object pointer) is a pointer. **std::atomic_compare_exchange_weak_explicit()** also requires both the success and failure memory orders to be specified, whereas the compare/exchange member functions have both a single memory order form (with a default of **std::memory_order_seq_cst**), and an overload that takes the success and failure memory orders separately.

Overloads for the non-member functions that apply to instances of the **std::atomic<>** template are conspicuous by their absence. For instantiations over the integral and pointer types, the overloads for the corresponding **std::atomic_xxx** types are sufficient, and the committee didn't feel they were sufficiently important to include for the general case. The operations on **std::atomic_flag** also buck the trend, in that they spell out the “flag” part in the names: **std::atomic_flag_test_and_set()**, **std::atomic_flag_test_and_set_explicit()**, **std::atomic_flag_clear()**, and **std::atomic_flag_clear_explicit()**.

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, as **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);
}
```

As with the atomic operations on other types, the **_explicit** variants are also provided to allow you to specify the desired memory ordering, and the **std::atomic_is_lock_free()** function can be used to check whether or not a particular instance uses locks to ensure the atomicity.

As described in the introduction, the Standard atomic types do more than just avoid the undefined behaviour associated with a *data race*: they allow the user to enforce an ordering of operations between threads. It is this enforced ordering that is the basis of the facilities for protecting data and synchronizing operations such as **std::mutex** and **std::unique_future<>**. With that in mind, let's move on to the real meat of this chapter: the details of the concurrency aspects of the memory model, and how atomic operations can be used to synchronize data and enforce ordering.

5.3 Synchronizing Operations and Enforcing Ordering with Atomic Types

Suppose you have two threads, one of which is populating a data structure to be read by the second. In order to avoid a problematic race condition, the first thread sets a flag to indicate that the data is ready, and the second thread doesn't read the data until the flag is set. Listing 5.2 shows such a scenario.

Listing 5.15: Reading and writing variables from different threads

```
#include <vector>
#include <cstdatomic>
#include <iostream>

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

void writer_thread()
{
    data.push_back(42);                        #3
    data_ready=true;                           #4
}
void reader_thread()
{
    while(!data_ready.load())                  #1
    {
        std::this_thread::sleep(
            std::milliseconds(1));
    }
    std::cout<<”The answer=”<<data[0]<<”\n”;  #2
}
```

Cueballs in code and text

Leaving aside the inefficiency of the loop waiting for the data to be ready (#1), we really need this to work, as otherwise sharing data between threads becomes impractical: every item of data is forced to be atomic. We've already said that it's undefined behaviour to have non-atomic reads (#2) and writes (#3) 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 (#3) *happens-before* the write to the **data_ready** flag (#4), and the read of the flag (#1) *happens-before* the read of the data (#2). When the value read from **data_ready** (#1) is **true**, the write *synchronizes-with* that read, creating a *happens-before* relationship. Since *happens-before* is transitive, the write to the data (#3) *happens-before* the write to the flag (#4), which *happens-before* the read of the **true** value from the flag (#1), which *happens-before* the read of the data (#2), and we 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.

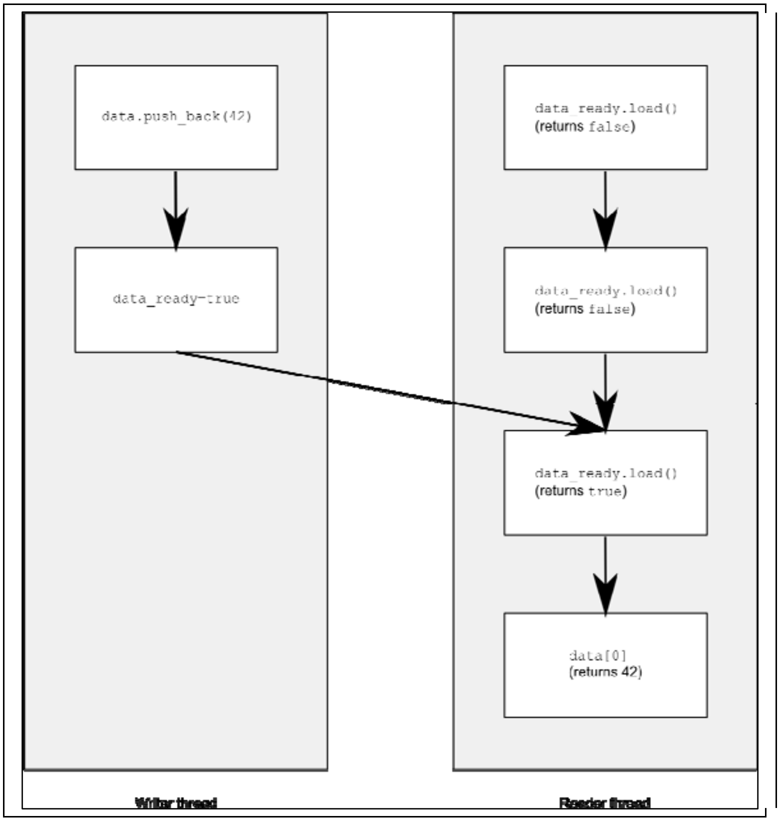


Figure 5.7 Enforcing an ordering between non-atomic operations using atomic operations.

All this might seem fairly intuitive: of course the operation that writes a value happens before an operation that reads that value! With the default atomic operations that is indeed true (which is why this is the default), but it does need spelling out: the atomic operations also have other options for the ordering requirements, which we'll come to shortly.

Now we've seen *happens-before* and *synchronizes-with* in action, its time to look at what they really mean. I'll start with *synchronizes-with*.

5.3.1 The synchronizes-with relation

The *synchronizes-with* relationship is something that you can only get between operations on atomic types. Operations on a data structure (such as locking a mutex) might provide this relationship if the data structure contains atomic types, and the operations on that data structure perform the appropriate atomic operations internally, but fundamentally it only comes from operations on atomic types.

The basic idea is this: a suitably-tagged atomic write operation on a variable **x** *synchronizes-with* a suitably-tagged atomic read operation on **x** that reads the value stored by (a) that write, (b) a subsequent atomic write operation on **x** by the same thread that performed the initial write, or (c) an atomic read-modify-write operation on **x** (such as **fetch_add()** or **compare_exchange_weak()**) by any thread, that read the value written. Leaving the "suitably tagged" part aside for now, as all operations on atomic types are "suitably tagged" by default, this essentially means what you might expect: if thread A stores a value, and thread B reads that value, there is a *synchronizes-with* relationship between the store in thread A and the load in thread B, just as in listing 5.2.

As I'm sure you've guessed, the nuances are all in the "suitably tagged" part. The C++ memory model allows various ordering constraints to be applied to the operations on atomic types, and this is the tagging to which I refer. The various options for memory ordering, and how they relate to the *synchronizes-with* relation are covered in section 5.3.3. First, let's step back and look at the *happens-before* relation.

5.3.2 The happens-before relation

The *happens-before* relation is the basic building block of operation ordering in a program: it specifies which operations see the effects of which other operations. For a single thread, it's largely straightforward: if one operation is *sequenced-before* another then it also *happens-before* it. This means that if the one operation (A) occurs in a statement prior to another (B) in the source code, then A *happens-before* B. We saw that in listing 5.2: the write to **data** (#3) happens-before the write to **data_ready** (#4). If the operations occur in the same statement, then in general there is no happens-before relation between them, as they are unordered. This is just another way of saying that the ordering is unspecified: we all know that the program in listing 5.3 will output "1,2" or "2,1", but it is unspecified which, as the order of the two calls to **get_num()** (#1) is unspecified.

Listing 5.16: The order of evaluation of arguments to a function call is unspecified.

```
#include <iostream>

void foo(int a,int b)
{
    std::cout<<a<<”,”<<b<<std::endl;
}

int get_num()
{
    static int i=0;
```

```
        return ++i;
    }

    int main()
    {
        foo(get_num(),get_num());           #1
    }
```

There are circumstances where operations within a single statement are sequenced such as where the built-in comma operator is used, or where the result of one expression is used as an argument to another expression, but in general operations within a single statement are unsequenced, and there is no sequenced-before (and thus no happens-before) relation between them. Of course, all operations in a statement happen-before all of the operations in the next statement.

This is really just a restatement of the single-threaded sequencing rules we're all used to, so what's new? The new part is the interaction between threads: if an operation A on one thread *inter-thread happens-before* an operation B on another thread, then A *happens-before* B. This doesn't really help much: we've just added a new relation (*inter-thread happens-before*), but this is an important relation when you're writing multi-threaded code.

At the basic level, *inter-thread happens-before* is relatively simple, and relies on the *synchronizes-with* relation introduced section 5.3.1: if an operation A in one thread *synchronizes-with* an operation B in another thread then A *inter-thread happens-before* B. It is also a transitive relation: if A inter-thread happens-before B and B inter-thread happens-before C, then A inter-thread happens-before C. We saw this in listing 5.2 as well.

Inter-thread happens-before also combines with the *sequenced-before* relation: if operation A is sequenced-before operation B, and operation B inter-thread happens-before operation C, then A inter-thread happens-before C. Similarly, if A synchronizes-with B and B is sequenced-before C, then A inter-thread happens-before C. These two together mean that if you make a series of changes to data in a single thread, you only need one synchronization path for the data to be visible to subsequent operations on the thread that executed C.

These are the crucial rules that enforce ordering of operations between threads, and make everything in listing 5.2 work. There are some additional nuances with data dependency, as we'll see shortly. In order to understand this, we need to cover the memory ordering tags used for atomic operations and how they relate to the *synchronizes-with* relation.

5.3.3 Memory Ordering for Atomic Operations

There are six memory ordering options that can be applied to operations on atomic types. They are: **memory_order_relaxed**, **memory_order_consume**, **memory_order_acquire**, **memory_order_release**, **memory_order_acq_rel**, and **memory_order_seq_cst**. Unless you specify otherwise for a particular operation, the memory ordering option for all operations on atomic types is **memory_order_seq_cst**, which is the most stringent of the available options. Though there are six ordering options, they represent three models: *sequentially-consistent* ordering (**memory_order_seq_cst**), *acquire-release* ordering (**memory_order_consume**, **memory_order_acquire**, **memory_order_release**, and **memory_order_acq_rel**), and *relaxed* ordering (**memory_order_relaxed**). What are the consequences of each choice for operation ordering and *synchronizes-with*?

Sequentially-Consistent Ordering

The default ordering is named *sequentially-consistent* because it implies that the behaviour of the program is consistent with a simple sequential view of the world. **If all operations on instances of atomic types are sequentially consistent, then the behaviour of a multi-threaded program is as-if all these operations were performed in some particular sequence by a single thread**. This is by far the easiest memory ordering to understand, which is why it is the default: all threads must see the same order of operations. This makes it easy to reason about the behaviour of code written with atomic variables — you can just write down all the possible sequences of operations by different threads, eliminate those that are inconsistent and verify that your code behaves as expected in the others. It also means that operations cannot be re-ordered — if your code has one operation before another in one thread, then that ordering must be seen by all other threads.

From the point of view of synchronization, a sequentially-consistent store *synchronizes-with* a sequentially-consistent load of the same variable that reads the value stored. This provides one ordering constraint on the operation of two (or more) threads, but sequential consistency is more powerful than that — any operations done after that load must also appear after the store to other threads in the system. The example below demonstrates this ordering constraint in action.

This ease of understanding can come at a price, though. On a machine with many processors it can impose a noticeable performance penalty, as the overall sequence of operations must be kept consistent between the processors, possibly requiring extensive (and expensive!) synchronization operations between the processors.

Listing 5.4 shows sequential-consistency in action. The loads and stores to **x** and **y** are explicitly tagged with **memory_order_seq_cst**, though this tag could be omitted in this case as it is the default. The **assert** (#1) can never fire, since either the store to **x** (#2) or the store to **y** (#3) must happen first, even though it is not specified which.

Listing 5.17: Sequential Consistency implies a total ordering

```
#include <cstdatomic>
#include <thread>
#include <assert.h>
```

```
std::atomic_bool x,y;
std::atomic_int z;

void write_x()
{
    x.store(true,std::memory_order_seq_cst); #2
}

void write_y()
{
    y.store(true,std::memory_order_seq_cst); #3
}

void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst))    #4
        ++z;
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst))    #5
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);                #1
}
```

Cueballs in code and preceding and following text

If the load of **y** in **read_x_then_y** (#4) returned **false** then the store to **x** must occur before the store to **y**, in which case the load of **x** in **read_y_then_x** (#5) must return **true**, since the **while** loop ensures that the **y** is **true** at this point. Since the semantics of **memory_order_seq_cst** require a single total ordering over all operations tagged **memory_order_seq_cst**, there is an implied ordering relationship between a load of **y** that returns **false** (#4) and the store to **y** (#2). For there to be a single total order, if one thread sees **x==true** and then subsequently sees **y==false** then this implies that the store to **x** occurs before the store to **y** in this total order.

Of course, because everything is symmetrical, it could also happen the other way round, with the load of **x** (#5) returning **false**, forcing the load of **y** (#4) to return **true**. In both cases, **z** is equal to 1. Of course, both loads can return **true**, leading to **z** being 2, but under no circumstances can **z** be zero.

The operations and *happens-before* relationships are shown in figure 5.3. The dashed line shows the implied ordering relationship required in order to maintain sequential consistency.

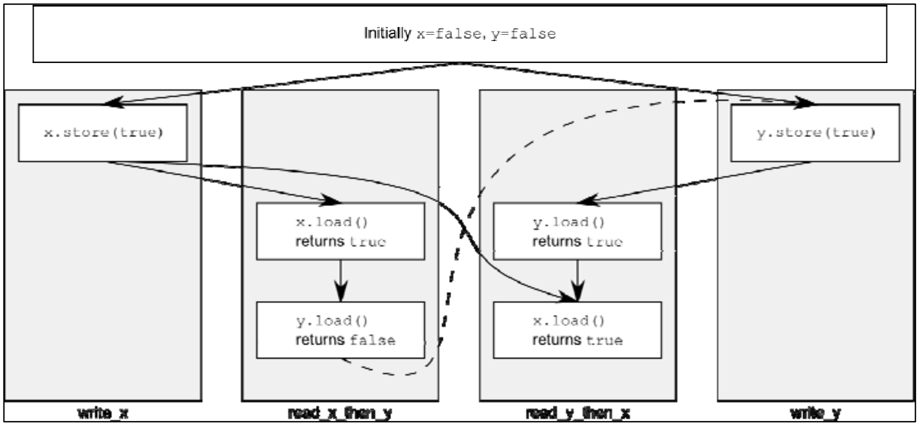


Figure 5.8 Sequential Consistency and *happens-before*

Sequential Consistency is the most straightforward and intuitive ordering, but also the most “expensive” memory ordering as it requires global synchronization between all threads. On a multi-processor system this may require quite extensive and time-consuming communication between processors.

In order to avoid this synchronization cost, we need to step outside the world of sequential consistency, and consider using other memory orderings.

Non-sequentially-consistent Memory Orderings

Once you step outside the nice sequentially-consistent world, things start to get complicated. Probably the single biggest issue to get to grips with is the fact that **there is no longer a single global order of events**. This means that different threads can see different views of the same operations, and any mental model we have operations from different threads neatly interleaved one after the other must be thrown away. Not only do we have to account for things happening truly concurrently, but **threads don't have to agree on the order of events**. In order to write (or even just to understand) any code that uses a memory ordering other than the default **memory_order_seq_cst** it is absolutely vital to get your head round this. It's so important I'll say it again: **threads don't have to agree on the order of events**.

This is best demonstrated by stepping completely outside the sequentially consistent world and using **memory_order_relaxed** for all operations. Once we've got to grips with that, we can then move back to acquire-release ordering which allows us to selectively introduce ordering relationships between operations and claw back some of our sanity.

Relaxed Ordering

Operations on atomic types performed with relaxed ordering do not participate in *synchronizes-with* relationships. Operations on the same variable within a single thread still obey *happens-before* relationships, but there is almost no requirement on ordering relative to other threads. The only requirement is that accesses to a single atomic variable from the same thread cannot be reordered: once a given thread has seen a particular value of an atomic variable, a subsequent read by that thread cannot retrieve an earlier value of the variable. Without any additional synchronization, the *modification order* of each variable is the only thing shared between threads that are using **memory_order_relaxed**.

To demonstrate quite how relaxed our relaxed operations can be we only need two threads, as in listing 5.5.

Listing 5.18: Relaxed operations have very little ordering requirements

```
#include <cstdatomic>
#include <thread>
#include <assert.h>

std::atomic_bool x,y;
std::atomic_int z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);    #4
    y.store(true,std::memory_order_relaxed);    #5
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));  #3
    if(x.load(std::memory_order_relaxed))       #2
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);                        #1
}
```

Cueballs in code and text

The assert (#1) *can* fire, because the load of **x** (#2) can read **false**, even though the load of **y** (#3) read **true** and the store of **x** (#4) *happens-before* the store of **y** (#5). **x** and **y** are different variables, so there are no ordering guarantees relating the visibility of values arising from operations on each.

Relaxed operations on different variables can be freely reordered provided they obey any *happens-before* relationships they are bound by (e.g. within the same thread). They do not introduce *synchronizes-with* relationships. The *happens-before* relationships from listing 5.5 are shown in figure 5.4, along with a possible outcome. Even though there is a *happens-before* relationship between the stores, and between the loads, there is not one between either store and either load, and so the loads can see the stores out of order.

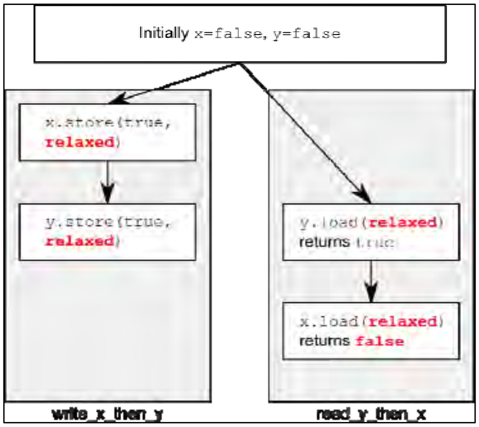

Initially x=false, y=false
x.store(true, relaxed)
y.store(true, relaxed)
y.load(relaxed) returns true
x.load(relaxed) returns false
write_x_then_y
read_y_then_x

Figure 5.9 Relaxed atomics and *happens-before*

To try and understand how this works, imagine that each variable is a man in a cubicle with a notepad. On his notepad he has a list of values. You can phone him up and ask him to give you a value, or you can tell him to write down a new value. If you tell him to write down a new value then he writes it down at the bottom of the list. If you ask him for a value then he reads you a number from the list.

The first time you talk to this man, if you ask him for a value he may give you *any* value from the list he has on his pad at the time. If you then ask him for another value he may give you the same one again, or a value from further down the list. He will never give you a value from further up the list. If you tell him to write a number down, and then subsequently ask him for a value he will either give you the number you told him to write down, or a number below *that* on the list.

So, imagine for a moment that his list starts with the values “5, 10, 23, 3, 1, 2”. If you ask for a value you could get any of those. If he gives you “10”, then the next time you ask he could give you “10” again, or any of the later ones, but not “5”. If you call him five times, he could say “10”, “10”, “1”, “2”, “2”, for example. If you tell him to write down “42” then he will add it to the end of the list. If you ask him for a number again, he'll keep telling you “42” until he (a) has another number on his list, and (b) he feels like telling it to you.

Now, imagine your friend Carl *also* has this chap's number. Carl can also phone him up and either ask him to write down a number, or ask for one, and this chap applies the same rules to Carl as he does to you. He only has one phone, so he can only deal with one of you at a time, so the list on his pad is a nice straightforward list. However, just because you got him to write down a new number doesn't mean he has to tell it to Carl, and vice-versa. If Carl asked him for a number and was told “23”, then just because you asked the man to write down “42” doesn't mean he'll tell that to Carl next time — he may tell Carl any of the numbers 23, 3, 1, 2, 42, or even the 67 that Fred told him to write down after you called. He could very well tell Carl “23”, “3”, “3”, “1”, “67” without being inconsistent with what he told you. It's like he keeps track of which number he told to whom with a little movable sticky note for each person, like in figure 5.5.

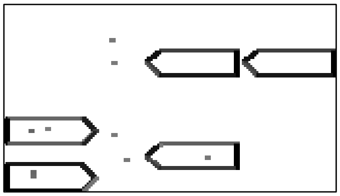


Figure 5.10 The notebook for the man in the cubicle

Now imagine that there's not just one man in a cubicle, but a whole cubicle farm, with loads of men with phones and notepads. These are all our atomic variables. Each variable has its own modification order (the list of values on the pad), but there's no relationship between them at all. If each caller (you, Carl, Anne, Dave and Fred) is a thread, then this is what you get when every operation uses **memory_order_relaxed**. There's a few additional things you can ask the man in the cubicle, such as “write down this number, and tell me what was bottom of the list” (**exchange**), and “write down *this* number if the number on the bottom of the list is *that*, otherwise tell me what I should have guessed” (**compare_exchange_strong**), but that doesn't affect the general principle.

If you think about the program logic from listing 5.5, then **write_x_then_y** is like some guy calling up the man in cubicle **x** and telling him to write **true**, then calling up the man in cubicle **y** and telling *him* to write **true**. The thread running **read_y_then_x** repeatedly calls up the man in cubicle **y** asking for a value until he says **true**, and then calls the man in cubicle **x** to ask for a value. The man in cubicle **x** is under no obligation to tell you any specific value off his list, and is quite within his rights to say **false**.

This makes relaxed atomic operations *very* hard to deal with. They must be used in combination with atomic operations that feature stronger ordering semantics in order to be useful for inter-thread synchronization. I strongly recommend avoiding relaxed atomic operations unless they are absolutely necessary, and even then only using them with extreme caution. Given the unintuitive results that can be achieved with just two threads and two variables in listing 5.5, it's not hard to imagine the possible complexity when more threads and more variables are involved.

One way to achieve additional synchronization without the overhead of full-blown sequential consistency is to use acquire-release ordering.

Acquire-Release Ordering

Acquire-release ordering is a step up from relaxed ordering: there is still no total order of operations, but it does introduce some synchronization. Under this ordering model, atomic loads are *acquire* operations (**memory_order_acquire**), atomic stores are *release* operations (**memory_order_release**), and atomic read-modify-write operations (such as **fetch_add** or **exchange**) are either *acquire*, *release* or both (**memory_order_acq_rel**). Synchronization is pairwise, between the thread that does the *release* and the thread that does the *acquire* — **a *release* operation *synchronizes-with* an *acquire* operation that reads the value written**. This means that different threads can *still* see different orderings, but these orderings are restricted. Listing 5.6 is a rework of listing 5.4 using acquire-release semantics rather than sequentially-consistent ones. In this case the assert (#1) *can* fire (just like in the relaxed ordering case), because it is possible for both the load of **x** (#2) and the load of **y** (#3) to read **false**. **x** and **y** are written by different threads, so the ordering from the *release* to the *acquire* in each case has no effect on the operations in the other threads.

Listing 5.19: Acquire-Release does not imply a total ordering

```
#include <cstdatomic>
#include <thread>
#include <assert.h>

std::atomic_bool x,y;
std::atomic_int z;

void write_x()
{
    x.store(true,std::memory_order_release);
}

void write_y()
{
    y.store(true,std::memory_order_release);
}

void read_x_then_y()
{
    while(!x.load(std::memory_order_acquire));
    if(y.load(std::memory_order_acquire))    #3
        ++z;
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));
    if(x.load(std::memory_order_acquire))    #2
```

```
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load()!=0);                #1
}
```

Cueballs in code and preceding text

Figure 5.6 shows the *happens-before* relationships from listing 5.6, along with a possible outcome where the two reading threads each have a different view of the world. This is possible because there is no *happens-before* relationship to force an ordering, as described above.

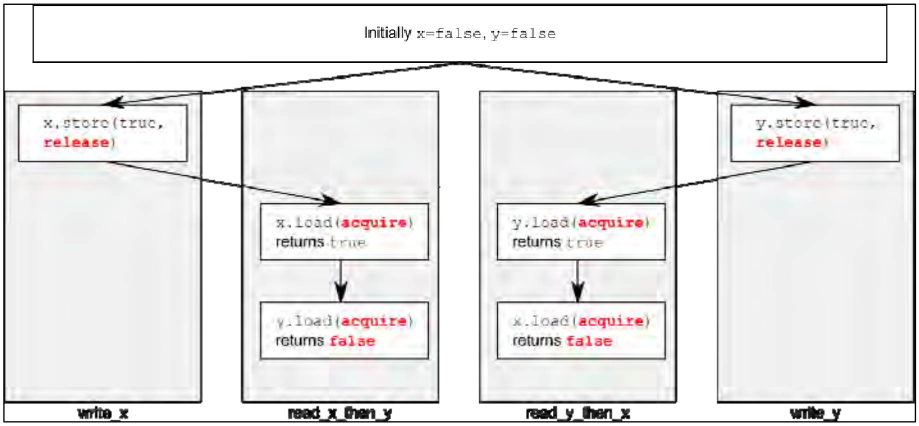

Initially x=false, y=false
x.store(true, release)
x.load(acquire) returns true
y.load(acquire) returns false
y.load(acquire) returns true
x.load(acquire) returns false
y.store(true, release)
write_x
read_x_then_y
read_y_then_x
write_y

Figure 5.11 Acquire-Release and *happens-before*

In order to see the benefit of acquire-release ordering, we need to consider two stores from the same thread, like in listing 5.5. If we change the store to **y** to use

memory_order_release and the load from **y** to use **memory_order_acquire** like in listing 5.7 then we actually impose an ordering on the operations on **x**.

Listing 5.20: Acquire-release operations can impose ordering on relaxed operations

```
#include <cstdatomic>
#include <thread>
#include <assert.h>

std::atomic_bool x,y;
std::atomic_int z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);    #3
    y.store(true,std::memory_order_release);    #2
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_acquire));  #1
    if(x.load(std::memory_order_relaxed))       #4
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);                        #5
}
```

Cueballs in code and text

Eventually, the load from **y** (#1) will see **true** as written by the store (#2). Since the store uses **memory_order_release**, and the load uses **memory_order_acquire** the store *synchronizes-with* the load. The store to **x** (#3) *happens-before* the store to **y** (#2), since they are in the same thread. Since the store to **y** *synchronizes-with* the load from **y**, the store to **x** also *happens-before* the load from **y**, and by extension *happens-before* the load from **x** (#4). Thus the load from **x** **must** read **true**, and the assert (#5) **cannot** fire. If the load from **y** was not in a **while** loop, this would not necessarily be the case — the load from

y might read **false**, in which case there would be no requirement on the value read from **x**. In order to provide any synchronization, acquire and release operations must be paired up — the value stored by a release operation must be seen by an acquire operation for either to have any effect. If either the store at #2 or the load at #1 was a relaxed operation, there would be no ordering on the accesses to **x**, so no guarantee that the load at #4 would read **true**, and the **assert** could fire.

We can still think about acquire-release ordering in terms of our men with notepads in their cubicles, but we have to add more to the model. First, imagine that every store that is done is part of some batch of updates, so when you call a man to tell him to write a number down you also tell him which batch this update is part of “please write down 99, as part of batch 423”. For the last store in a batch, you tell this to the man too: “please write down 147, which is the last store in batch 423”. The man in the cubicle will then duly write down this information, along with who gave him the value. This models a store-release operation. Next time you tell someone to write down a value you increase the batch number: “please write down 41, as part of batch 424”.

When you ask for a value, you now have a choice: you can either just ask for a value (which is a relaxed load), in which case the man just gives you the number, or you can ask for a value, and information about whether or not it is the last in a batch (which models a load-acquire). If you ask for the batch information, and the value was not last in a batch then the man will tell you something like “the number is 987, which is just a 'normal' value”, whereas if it *was* the last in a batch then he'll tell you something like “the number is 987, which is the last number in batch 956 from Anne”. Now, here's where the acquire-release semantics kick in: if you tell the man all the batches you know about when you ask for a value then he'll look down his list for the last value from any of the batches you know about, and either give you that number of one further down the list.

How does this model acquire-release semantics? Let's take a look at our example and see. First off, thread **a** is running **write_x_then_y** and says to the man in cubicle **x** “please write **true** as part of batch 1 from thread **a**”, which he duly writes down. Thread **a** then says to the man in cubicle **y** “please write **true** as the last write of batch 1 from thread **a**”, which he duly writes down. In the mean time, thread **b** is running **read_y_then_x**. Thread **b** keeps asking the man in box **y** for a value with batch information until he says “**true**”. He may have to ask many times, but eventually the main will say “**true**”. The man in box **y** doesn't *just* say “**true**” though — he also says “this is the last write in batch 1 from thread **a**”.

Now, thread **b** then goes on to ask the man in box **x** for a value, but this time he says “please can I have a value, and by the way I know about batch 1 from thread **a**”. So now, the man from cubicle **x** has to look down his list for the last mention of batch 1 from thread **a**. The only mention he has is the value **true**, which is also the last value on his list, so he **must** read out that value, otherwise he's breaking the rules of the game.

If we look at the definition of *happens-before* back in section 5.3.2, one of the important properties is that it is *transitive*: if A inter-thread happens-before B and B inter-thread happens-before C, then A inter-thread happens-before C. This means that acquire-release ordering can be used to synchronize data across several threads, even when the "intermediate" threads haven't actually touched the data.

Transitive Synchronization with Acquire-Release Ordering

In order to think about transitive ordering, you need at least three threads. The first thread modifies some shared variables and does a store-release to one of them. A second thread then reads the variable subject to the store-release with a load-acquire, and performs a store-release on a second shared variable. Finally, a third thread does a load-acquire on that second shared variable. Provided that the load-acquire operations see the values written by the store-release operations to ensure the *synchronizes-with* relationships, this third thread can read the values of the other variables stored by the first thread, even if the intermediate thread didn't touch any of them. This scenario is shown in listing 5.8.

Listing 5.21: Transitive synchronization using `std::memory_order_acquire` and `std::memory_order_release`.

```
    std::atomic<int> data[5];
    std::atomic<bool> sync1(false),sync2(false);

    void thread_1()
    {
        data[0].store(42,std::memory_order_relaxed);
        data[1].store(97,std::memory_order_relaxed);
        data[2].store(17,std::memory_order_relaxed);
        data[3].store(-141,std::memory_order_relaxed);
        data[4].store(2003,std::memory_order_relaxed);
        sync1.store(true,std::memory_order_release);                #3
    }

    void thread_2()
    {
        while(!sync1.load(std::memory_order_acquire));              #1
        sync2.store(std::memory_order_release);                     #2
    }

    void thread_3()
    {
        while(!sync2.load(std::memory_order_acquire));              #4
        assert(data[0].load(std::memory_order_relaxed)==42);
        assert(data[1].load(std::memory_order_relaxed)==97);
        assert(data[2].load(std::memory_order_relaxed)==17);
        assert(data[3].load(std::memory_order_relaxed)==-141);
        assert(data[4].load(std::memory_order_relaxed)==2003);
    }
```

Cueballs in code and text

Even though **thread_2** only touches the variables **sync1** (#1) and **sync2** (#2), this is enough for synchronization between **thread_1** and **thread_3** to ensure that the **assert**s don't fire. First off, the stores to **data** from **thread_1** *happen-before* the store to **sync1** (#3), since they are *sequenced-before* it in the same thread. Because the load from **sync1** (#3) is in a **while** loop, it will eventually see the value stored from **thread_1**, and thus form the second half of the release-acquire pair. Therefore the store to **sync1** *happens-before* the final load from **sync1** in the **while** loop. This load is *sequenced-before* (and thus *happens-before*) the store to **sync2** (#2) which forms a release-acquire pair with the final load from the **while** loop in **thread_3** (#4). The store to **sync2** (#2) thus *happens-before* the load (#4), which *happens-before* the loads from **data**. Because of the transitive nature of *happens-before* we can chain it all together: the stores to **data** *happen-before* the store to **sync1** (#3), which *happens-before* the load from **sync1** (#1), which *happens-before* the store to **sync2** (#2), which *happens-before* the load from **sync2** (#4), which *happens-before* the loads from **data**. Thus the stores to **data** in **thread_1** *happen-before* the loads from **data** in **thread_3**, and the **assert**s cannot fire.

In this case, we could combine **sync1** and **sync2** into a single variable by using a read-modify-write operation with **memory_order_acq_rel** in **thread_2**. One option would be to use **compare_exchange_strong()** to ensure that the value is only updated once the store from **thread_1** has been seen:

```
std::atomic<int> sync(0);
void thread_1()
{
    // ...
    sync.store(1,std::memory_order_release);
}
void thread_2()
{
    int expected=1;
    while(!sync.compare_exchange_strong(expected,2,
                                        std::memory_order_acq_rel))
        expected=1;
}
void thread_3()
{
    while(sync.load(std::memory_order_acquire)<2);
    // ...
}
```

If you use read-modify-write operations, it is important to pick which semantics you desire. In this case, we want both *acquire* and *release* semantics, so **memory_order_acq_rel** is appropriate, but you can use other orderings too. A **fetch_sub** operation with **memory_order_acquire** semantics does not *synchronize-with* anything, even though it stores a value, since it is not a release operation. Likewise, a store cannot *synchronize-with* a **fetch_or** with **memory_order_release** semantics, since the read part of the **fetch_or** is not an acquire operation. Read-modify-write operations with **memory_order_acq_rel** semantics behave as both an acquire and a release, so a prior store can synchronize-with such an operation, and it can synchronize-with a subsequent load, as is the case in this example.

If you mix acquire-release operations with sequentially-consistent operations, the sequentially-consistent loads behave like loads with acquire semantics, and sequentially-consistent stores behave like stores with release semantics. Sequentially-consistent read-modify-write operations behave as both acquire and release operations. Relaxed operations are still relaxed, but are bound by the additional *synchronizes-with* and consequent *happens-before* relationships introduced through the use of acquire-release semantics.

Despite the potentially non-intuitive outcomes, anyone who's used locks has had to deal with the same ordering issues: locking a mutex is an *acquire* operation, and unlocking the mutex is a *release* operation. With mutexes, we learn that we must ensure that the same mutex is locked when we read a value as was locked when we wrote it, and the same applies here — your acquire and release operations have to be on the same variable to ensure an ordering. If data is protected with a mutex, then the exclusive nature of the lock means that the result is indistinguishable from that had the lock and unlock been sequentially-consistent operations. Similarly, if you use acquire and release orderings on atomic variables to build a simple lock then from the point-of-view of code that *uses* the lock the behaviour will appear sequentially consistent, even though the internal operations are not.

If you don't need the stringency of sequentially-consistent ordering for your atomic operations, the pair-wise synchronization of acquire-release ordering has the potential for a much lower synchronization cost than the global ordering required for sequentially-consistent operations. The trade-off here is the mental cost required to ensure that the ordering works correctly, and that the non-intuitive behaviour across threads is not problematic.

*Data-dependency with acquire-release ordering and **memory_order_consume***

In the introduction to this section I said that **memory_order_consume** was part of the acquire-release ordering model, but it was conspicuously absent from the preceding description. This is because **memory_order_consume** is special: it is all about data dependencies, and introduces the data-dependency nuances to the *inter-thread happens-before* relation mentioned in section 5.3.2.

There are two new relations that deal with data dependencies: *dependency-ordered-before* and *carries-a-dependency-to*. Just like *sequenced-before*, *carries-a-dependency-to* applies strictly within a single thread and essentially models the data dependency between operations: if the result of an operation A is used as an operand for an operation B, then A *carries-a-dependency-to* B. If the result of operation A is a value of a scalar type such as an **int**, then the relation still applies if the result of A is stored in a variable, and that variable is then used as an operand for operation B. This operation is also transitive, so if A *carries-a-dependency-to* B, and B *carries-a-dependency-to* C, then A *carries-a-dependency-to* C.

On the other hand, the *dependency-ordered-before* relation can apply between threads. It is introduced by using atomic load operations tagged with **memory_order_consume**. This is a special case of **memory_order_acquire** that limits the synchronized data to direct dependencies: a store operation A tagged with **memory_order_release, memory_order_acq_rel** or **memory_order_seq_cst** is *dependency-ordered-before* a load operation B tagged with **memory_order_consume** if the consume reads the value stored. This is as opposed to the *synchronizes-with* relationship you get if the load uses **memory_order_acquire**. If this operation B then *carries-a-dependency-to* some operation C, then A is also *dependency-ordered-before* C.

This wouldn't actually do us any good for synchronization purposes if it didn't affect the *inter-thread happens-before* relation, but it does: if A is *dependency-ordered-before* B, then A also *inter-thread happens-before* B.

One important use for this kind of memory ordering is where the atomic operation loads a pointer to some data. By using **memory_order_consume** on the load, and **memory_order_release** on the prior store, you ensure that the pointed-to data is correctly synchronized, without imposing any synchronization requirements on any other non-dependent data. Listing 5.9 shows an example of this scenario.

Listing 5.22: Using std::memory_order_consume to synchronize data in a dynamically allocated object

```
struct X
{
    int i;
    std::string s;
};

std::atomic<X*> p;
std::atomic_int a;

void create_x()
```

```
{
    X* x=new X;
    x->i=42;
    x->s=”hello”;
    a.store(99,std::memory_order_relaxed);                      #1
    p.store(x,std::memory_order_release);                       #2
}

void use_x()
{
    X* x;
    while(!(x=p.load(std::memory_order_consume)))               #3
        std::this_thread::sleep(std::chrono::microseconds(1));
    assert(x->i==42);                                           #4
    assert(x->s==”hello”);                                      #5
    assert(a.load(std::memory_order_relaxed)==99);              #6
}

int main()
{
    std::thread t1(create_x);
    std::thread t2(use_x);
    t1.join();
    t2.join();
}
```

Cueballs in code and text

Even though the store to **a** (#1) is *sequenced-before* the store to **p** (#2), and the store to **p** is tagged **memory_order_release**, the load of **p** (#3) is tagged **memory_order_consume**. This means that the store to **p** only *happens-before* those expressions which are dependent on the value loaded from **p**. This means that the asserts on the data members of the **X** structure (#4, #5) are guaranteed not to fire, since the load of **p** carries a dependency to those expressions through the variable **x**. On the other hand, the assert on the value of **a** (#6) may or may not fire: this operation is not dependent on the value loaded from **p**, and so there is no guarantee on the value that is read. This is particularly apparent since it is tagged with **memory_order_relaxed**, as we shall see.

Now I've covered the basics of the memory orderings, it's time to look at the more complex parts of the *synchronizes-with* relation, which manifest in the form of *release sequences*.

5.3.4 Release Sequences and synchronizes-with

Back in 5.3.1, I mentioned that you could get a *synchronizes-with* relationship between a store to an atomic variable and a load of that atomic variable from another thread, even when there was a sequence of read-modify-write operations between the store and the load, provided all the operations were “suitably tagged”. Now we've covered the possible memory ordering “tags”, I can elaborate on this. If the store is tagged with **memory_order_release** or **memory_order_seq_cst**, and the load is tagged with **memory_order_consume**, **memory_order_acquire** or **memory_order_seq_cst**, and each operation in the chain loads the value written by the previous operation, then the chain of operations constitutes a *release sequence* and the initial store *synchronizes-with* (for **memory_order_acquire** or **memory_order_seq_cst**) or is *dependency-ordered-before* (for **memory_order_consume**) the final load. Any atomic read-modify-write operations in the chain can have *any* memory ordering (even **memory_order_relaxed**).

To see what this means, and why it's important, consider an **atomic_int** being used as a **count** of the number of items in a shared queue, as in listing 5.10. One way to handle things would be to have the thread producing the data store the items in a shared buffer, and then do **count.store(number_of_items,memory_order_release)** (#1) to let the other threads know that there is data available. The threads consuming the queue items might then do **count.fetch_sub(1,memory_order_acquire)** (#2) to claim an item from the queue, prior to actually reading the shared buffer (#3). Once the **count** becomes zero, there are no more items, and the thread must wait (#4).

Listing 5.23: Reading values from a queue with atomics

```
#include <cstdatomic>
#include <thread>

std::vector<int> queue_data;
std::atomic_int count;

void populate_queue()
{
    unsigned const number_of_items=20;
    queue_data.clear();
    for(unsigned i=0;i<number_of_items;++i)
    {
        queue_data.push_back(i);
    }

    count.store(number_of_items,std::memory_order_release);             #1
}

void consume_queue_items()
{
    while(true)
    {
        int item_index;
```

```
            if((item_index=count.fetch_sub(1,std::memory_order_acquire))<=0)#2
            {
                wait_for_more_items();                                    #4
                continue;
            }
            process(queue_data[item_index-1]);                            #3
        }
    }

    int main()
    {
        std::thread a(populate_queue);
        std::thread b(consume_queue_items);
        std::thread c(consume_queue_items);
        a.join();
        b.join();
        c.join();
    }
```

Cueballs in code and preceding text

If there is one consumer thread, this is fine: the **fetch_sub** is a read, with **memory_order_acquire** semantics, and the store had **memory_order_release** semantics, so the store synchronizes-with the load and the thread can read the item from the buffer. If there are two threads reading, the second **fetch_sub** will see the value written by the first, and not the value written by the **store**. Without the rule about the *release sequence* this second thread would therefore not have a happens-before relationship with the first thread, and it would not be safe to read the shared buffer unless the first **fetch_sub** also had **memory_order_release** semantics, which would introduce unnecessary synchronization between the two consumer threads. Thankfully, the first **fetch_sub does** participate in the *release sequence*, and so the **store** synchronizes-with the second **fetch_sub**. There is still no *synchronizes-with* relationship between the two consumer threads. This is shown in figure 5.7. The dotted lines in figure 5.7 show the *release-sequence*, and the solid lines show the *happens-before* relationships.

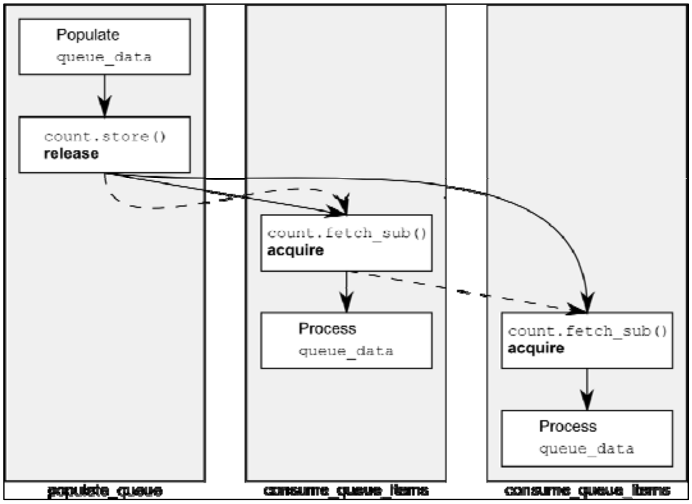


Figure 5.12 The *release-sequence* for the queue operations from listing 5.10

There can be any number of links in the chain, but provided they are all read-modify-write operations such as **fetch_sub**, and none of them are tagged with **memory_order_relaxed**, the **store** will still *synchronize-with* each one that is tagged **memory_order_acquire**. In this example, all the links are all the same, and all are acquire operations, but they could be a mix of different operations, with different memory ordering semantics.

Whilst most of the synchronization relationships come from the memory ordering semantics applied to operations on atomic variables, it is also possible to introduce additional ordering constraints by using *fences*.

5.3.5 Fences

An atomic operations library would not be complete without a set of fences. These are operations that enforce memory ordering constraints without modifying any data, and are typically combined with atomic operations that use the **memory_order_relaxed** ordering constraints. Fences are global operations and affect the ordering of other atomic operations in the thread that executed the fence. Fences are also commonly called *memory barriers*, and get their name because they put a line in the code which certain operations cannot cross. As you may recall from section 5.3.3, relaxed operations on separate variables can usually be freely reordered by the compiler or the hardware. Fences restrict this freedom and introduce *happens-before* and *synchronizes-with* relationships that were not present before.

Let's start by adding a fence between the two atomic operations on each thread in listing 5.5. This is shown in listing 5.11.

Listing 5.24: Relaxed operations can be ordered with fences

```
#include <cstdatomic>
#include <thread>
#include <assert.h>

std::atomic_bool x,y;
std::atomic_int z;

void write_x_then_y()
{
    x.store(true,std::memory_order_relaxed);              #3
    std::atomic_thread_fence(std::memory_order_release);   #1
    y.store(true,std::memory_order_relaxed);              #4
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));            #5
    std::atomic_thread_fence(std::memory_order_acquire);   #2
    if(x.load(std::memory_order_relaxed))                 #6
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);                                  #7
}
```

Cueballs in code and text

The release fence (#1) *synchronizes-with* the acquire fence (#2), since the load from **y** at #5 reads the value stored at #4. This means that the store to **x** at #3 *happens-before* the load from **x** at #6, so the value read must be **true** and the assert at #7 will not fire. This is in contrast to the original case without the fences where the store to and load from **x** where not ordered, and so the assert could fire. Note that both fences are necessary: you need a release in one thread and an acquire in another to get *a synchronizes-with* relationship.

In this case, the release fence (#1) has the same effect as if the store to **y** (#4) was tagged with **memory_order_release** rather than **memory_order_relaxed**. Likewise, the acquire fence (#2) makes it as-if the load from **y** (#5) was tagged with **memory_order_acquire**. This is the general idea with fences: if an acquire operation that sees the result of a store that takes place after a release fence, then the fence *synchronizes-with* that acquire operation, and if a load that takes place before an acquire fence sees the result of a release operation then the release operation *synchronizes-with* the acquire fence. Of course, you can have fences on both sides, as in the example here, in which case if a load that takes place before the acquire fence sees a value written by a store that takes place after the release fence then the release fence *synchronizes-with* the acquire fence.

Though the fence synchronization depends on the values read or written by operations before or after the fence, it is important to note that the synchronization point is the fence itself. If we take **write_x_then_y** from listing 5.11 and move the write to **x** after the fence as shown below, then the assert is no longer guaranteed to fire, even though the write to **x** comes before the write to **y**.

```
void write_x_then_y()
{
    std::atomic_thread_fence(std::memory_order_release);
    x.store(true,std::memory_order_relaxed);                    #1
    y.store(true,std::memory_order_relaxed);                    #2
}
```

#1, #2 These two operations are no longer separated by the fence, and so are no longer ordered

It is only when the fence comes *between* the store to **x** and the store to **y** that it imposes an ordering. Of course, the presence or absence of a fence doesn't affect any enforced orderings on *happens-before* relations that exist due to other atomic operations.

This example, and almost every other example so far in this chapter is built entirely from variables with an atomic type. However, the real benefit to using atomic operations to enforce an ordering is that they can enforce an ordering on non-atomic operations, and thus avoid the undefined behaviour of a *data race*, as we saw back in listing 5.2.

5.3.6 Ordering non-atomic operations with atomics

If we replace **x** from listing 5.11 with an ordinary non-atomic **bool** (as in listing 5.12), the behaviour is guaranteed to be the same. The fences still provide an enforced ordering of the store to **x** (#1) and the store to **y** (#2), and the load from **y** (#3) and the load from **x** (#4), and there is still a *happens-before* relationship between the store to **x** and the load from **x**, so the assert (#6) still won't fire.

Listing 5.25: Atomic operations and fences can enforce an ordering on non-atomic operations

```
#include <cstdatomic>
#include <thread>
#include <assert.h>

bool x=false;                                                  #5
std::atomic_bool y;
std::atomic_int z;

void write_x_then_y()
{
    x=true;                                                     #1
    std::atomic_thread_fence(std::memory_order_release);
    y.store(true,std::memory_order_relaxed);                    #2
}

void read_y_then_x()
{
    while(!y.load(std::memory_order_relaxed));                  #3
    std::atomic_thread_fence(std::memory_order_acquire);
    if(x)                                                       #4
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0);                                        #6
}
```

Cueballs in code and preceding and following text

#5 x is a now plain non-atomic variable

The store to (#2) and load from (#3) **y** still have to be atomic, otherwise there would be a *data race* on **y**, but the fences enforce an ordering on the operations on **x**, once the reading thread has seen the stored value of **y**. This enforced ordering means that there is no data race on **x**, even though it is modified by one thread and read by another.

It's not just fences that can order non-atomic operations — we saw the ordering effects back in listing 5.9 with a **memory_order_release** / **memory_order_consume** pair ordering non-atomic accesses to a dynamically allocated object, and many of the examples in this chapter could be rewritten with some of the **memory_order_relaxed** operations replaced with plain non-atomic operations instead.

Ordering of non-atomic operations through the use of atomic operations is where the *sequenced-before* part of *happens-before* becomes so important. If a non-atomic operation is *sequenced-before* an atomic operation, and that atomic operation *happens-before* an operation in another thread, the non-atomic operation also *happens-before* that operation in the other thread. This is where the ordering on the operations on **x** in listing 5.12 comes from, and why the example in listing 5.2 works. This is also the basis for the higher-level synchronization facilities in the C++ Standard Library, such as mutexes and condition variables: to see how this works, consider the simple spin-lock from listing 5.1.

The **lock()** operation is a loop on **flag.test_and_set()** using **std::memory_order_acquire** ordering, and the **unlock()** is a call to **flag.clear()** with **std::memory_order_release** ordering. When the first thread calls **lock()**, the flag is initially clear, so the first call to **test_and_set()** will set the flag and return **false**, indicating that this thread now has the lock, and terminating the loop. The thread is then free to modify any data protected by the mutex. Any other thread that calls **lock()** at this time will find the flag already set, and will be blocked in the **test_and_set()** loop.

When the thread with the lock is done modifying the protected data, it calls **unlock()**, which calls **flag.clear()** with **std::memory_order_release** semantics. This then *synchronizes-with* a subsequent call to **flag.test_and_set()** from an invocation of **lock()** on another thread, since this call has **std::memory_order_acquire** semantics.

Since the modification of the protected data is necessarily *sequenced-before* the **unlock()** call, this modification *happens-before* the **unlock()**, and thus *happens-before* the subsequent **lock()** call from the second thread, and *happens-before* any accesses to that data from this second thread once it has acquired the lock. Though other mutex implementations will have different internal operations, the basic principle is the same: **lock()** is an acquire operation on an internal memory location, and **unlock()** is a release operation on that same memory location.

5.4 Summary

In this chapter we've covered the low level details of the C++0x memory model and the atomic operations that provide the basis for synchronization between threads. This includes the basic atomic types and the generic **std::atomic<>** template, the operations on them, and the complex details of the various memory ordering options.

We've also looked at fences, and how they can be paired with operations on atomic types to enforce an ordering. Finally, we've come back to the beginning with a look at how the atomic operations can be used to enforce an ordering between non-atomic operations on separate threads.

In the next chapter we'll look at using the high level synchronization facilities alongside atomic operations to design efficient containers for concurrent access, and write algorithms that process data in parallel.

6

Designing Data Structures for Concurrency I: Lock-based Data Structures

In the last chapter we looked at the low level details of atomic operations and the memory model. In this chapter we'll take a break from the low level details (though we'll be needing them for chapter 7), and think about data structures.

The choice of data structure to use for a programming problem can be a key part of the overall solution, and parallel programming problems are no exception. If a data structure is to be accessed from multiple threads then either it must be completely immutable so the data never changes and no synchronization is necessary, or the program must be designed to ensure that changes are correctly synchronized between threads. One option is to use a separate mutex and external locking to protect the data, using the techniques we looked at in chapters 3 and 4, and another is to design the data structure itself for concurrent access.

When designing a data structure for concurrency, we can use the basic building blocks of multi-threaded applications from earlier chapters such as mutexes and condition variables. Indeed, we've already seen a couple of examples showing how to combine these building blocks to write data structures that are safe for concurrent access from multiple threads.

In this chapter we'll start by looking at some general guidelines for designing data structures for concurrency. We'll then take the basic building blocks of locks and condition variables and revisit the design of those basic data structures before moving on to more complex data structures. In chapter 7 we'll then look at how to go right back to basics and use the atomic operations described in chapter 5 to build data structures without locks.

So, without further ado, let's take a look at what is involved in designing a data structure for concurrency.

6.1 What does it mean for a Data Structure to be designed for concurrency?

At the basic level, designing a data structure for concurrency means that multiple threads can access the data structure concurrently, either performing the same or distinct operations, and each thread will see a consistent view of the data structure. No data will be lost or corrupted, all invariants will be upheld and there will be no problematic race conditions. Such a data structure is said to be *thread-safe*.

Truly designing for concurrency means more than that though: it means providing the *opportunity for concurrency* to threads accessing the data structure. By it's very nature, a mutex provides *mutual exclusion*: only one thread can acquire a lock on the mutex at a time. The use of a mutex to protect a data structure does so by explicitly *preventing* true concurrent access to the data it protects.

This is called *serialization*: threads take turns to access the data protected by the mutex; they must access it serially rather than concurrently. Consequently, careful thought must go into the design of the data structure to enable true concurrent access. Some data structures have more scope for true concurrency than others, but in all cases the idea is the same: the smaller the protected region, the fewer operations are serialized, the greater the potential for concurrency.

Before we go on to look at some data structure designs, let's have a quick look at some simple guidelines for what to consider when designing for concurrency.

6.1.1 Guidelines for Designing Data Structures for Concurrency

As I just mentioned, there are two aspects to consider when designing data structures for concurrent access: ensuring that the accesses are *safe* and *enabling* genuine concurrent access. The basics of how to make the data structure thread-safe were covered back in chapter 3:

- Ensure that no thread can see a state where the *invariants* of the data structure have been broken by the actions of another thread.
- Take care to *avoid race conditions* inherent in the interface to the data structure by providing functions for complete operations rather than for operation steps.
- Pay attention to how the data structure behaves in the presence of exceptions to ensure that the invariants are not broken.
- *Minimize the opportunities for deadlock* when using the data structure by restricting the scope of locks and avoiding nested locks where possible.

However, before you think about that, it's also important to think about what constraints you wish to put on the users of the data structure:

- If one thread is accessing the data structure through a particular function, which functions are safe to call from other threads?

This is actually quite a crucial question to think about. Generally constructors and destructors require exclusive access to the data structure, but it is up to the user to ensure that they are not accessed before construction is complete or after destruction has started. If the data structure supports assignment, **`swap()`** or copy-construction then as the designer of the data structure you need to decide whether these operations are safe to call concurrently with other operations, or whether they require the user to ensure exclusive access even though the majority of functions for manipulating the data structure may be called from multiple threads concurrently without problem.

The second aspect to consider is that of enabling genuine concurrent access. I can't offer much in the way of guidelines here; instead here's a list of questions to ask yourself as the data structure designer:

- Can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
- Can different parts of the data structure be protected with different mutexes?
- Do all operations require the same level of protection?
- Can a simple change to the data structure improve the opportunities for concurrency without affecting the operational semantics?

All these questions are guided by a single idea: how can we minimize the amount of serialization that must occur and enable the greatest amount of true concurrency? It is not uncommon for data structures to allow concurrent access from multiple threads that merely read the data structure, whilst a thread that can modify the data structure must have exclusive access. This is supported by using constructs like **`boost::shared_mutex`**. Likewise, as we shall see shortly, it is quite common for a data structure to support concurrent access from threads performing different operations, whilst serializing threads that try to perform the same operation.

The simplest thread-safe data structures typically use mutexes and locks to protect the data — though there are issues with this, as we saw in chapter 3, it is relatively easy to ensure that only one thread is accessing the data structure at a time. To ease us into the design of thread-safe data structures, we'll stick to looking at such lock-based data structures in this chapter, and leave the design of concurrent data structures without locks for chapter 7.

6.2 Lock-based Concurrent Data Structures

The design of lock-based concurrent data structures is all about ensuring that the right mutex is locked when accessing the data, and ensuring that the lock is held for the minimum amount of time. This is hard enough when there is just one mutex protecting a data structure — we need to ensure that data cannot be accessed outside the protection of the mutex lock, and that there are no race conditions inherent in the interface, as we saw in chapter 3. If we use separate mutexes to protect separate parts of the data structure then these issues are compounded, and there is now also the possibility of deadlock if the operations on the data structure require more than one mutex to be locked. We therefore need to consider the design of a data structure with multiple mutexes even more carefully than the design of a data structure with a single mutex.

In this section we're going to apply the guidelines from section 6.1.1 to the design of several simple data structures, using mutexes and locks to protect the data. In each case we shall seek out the opportunities for enabling greater concurrency whilst ensuring that the data structure remains thread-safe.

Let's start by looking at the stack implementation from chapter 3: it's one of the simplest data structures around, and it only uses a single mutex. Is it really thread-safe? How does it fare from the point of view of achieving true concurrency?

6.2.1 A Thread-safe Stack using Locks

The thread-safe stack from chapter 3 is reproduced in listing 6.1. The intent was to write a thread-safe data structure akin to **std::stack<>**, which supported pushing data items onto the stack, and popping them off again.

Listing 6.26: A class definition for a thread-safe stack

```
#include <exception>

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    stack(){}
    stack(const stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }
    stack& operator=(const stack&) = delete;

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(new_value);                                #1
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();                #2
        std::shared_ptr<T> const res(new T(data.top()));     #3
        data.pop();                                          #4
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=data.top();                                    #5
        data.pop();                                          #6
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

Cueballs in code and text

Let's look at each of the guidelines in turn, and see how they apply here.

Firstly, as you can see, the basic thread-safety is provided by protecting each member function with a lock on the mutex, **m**. This ensures that only one thread is actually accessing the data at any one time, so provided each member function maintains the invariants, no thread can see a broken invariant.

Secondly, there is a potential for a race condition between **empty()** and either of the **pop()** functions, but since the code explicitly checks for the contained stack being empty whilst holding the lock in **pop()**, this race condition is not problematic. By returning the popped data item directly as part of the call to **pop()** we avoid a potential race condition that would be present with separate **top()** and **pop()** member functions such as those in **std::stack<>**.

Next, there are a few potential sources of exceptions. Locking a mutex may throw an exception, but not only is this likely to be exceedingly rare (as it indicates a problem with the mutex or a lack of system resources), it is also the first operation in each member function. Since no data has been modified, this is safe. Unlocking a mutex cannot fail, so that is always safe, and the use of **std::lock_guard<>** ensures that the mutex is never left locked.

The call to **data.push()** (#1) may throw an exception either due to the copying of the data value, or because it couldn't allocate enough memory to extend the underlying data structure. Either way, **std::stack<>** guarantees it will be safe, so that's not a problem either.

In the first overload of **pop()**, the code itself might throw an **empty_stack** exception (#2), but nothing has been modified, so that's safe. The creation of **res** (#3) might thrown an exception though for several reasons: first, the **new** might throw due to lack of memory, secondly the copy constructor of the data item to be returned might throw, and thirdly the constructor of **std::shared_ptr** might throw if it cannot allocate memory for its internal data. In all three cases, the C++ runtime and Standard Library ensure that there are no memory leaks and the new object (if any) is correctly destroyed. Since we *still* haven't modified the underlying stack, we're still OK. The call to **data.pop()** (#4) is guaranteed not to throw, as is the return of the result, so this overload of **pop()** is exception safe.

The second overload of **pop()** is similar, except this time it's the copy assignment operator that can throw (#5) rather than the construction of a new object and a **std::shared_ptr** instance. Again, we don't actually modify our data structure until the call to **data.pop()** (#6), which is still guaranteed not to throw, so this overload is exception safe too.

Finally, **empty()** doesn't modify any data, so that's exception safe.

There are a couple of opportunities for deadlock here, since we call user code whilst holding a lock: the copy constructor (#1,#3) and copy assignment operator (#5) on the contained data items. If these functions either call member functions on the same stack as the item is being inserted into or removed from, or they require a lock of any kind and another lock was held when the stack member function was invoked then there is the possibility of deadlock. However, it is sensible to require that users of the stack are responsible for ensuring this: you can't reasonably expect to add an item onto a stack or remove it from a stack without copying it.

Because all the member functions use a **std::lock_guard<>** to protect the data, it is safe for any number of threads to call the **stack** member functions. The only member functions which are not safe are the constructors and destructor, but this isn't a particular problem: the object can only be constructed once, and destroyed once. Calling member functions on an incompletely-constructed object or a partially-destructed object is never a good idea whether done concurrently or not. As a consequence, the user must ensure that other threads are not able to access the stack until it is fully constructed, and must ensure that all threads have ceased accessing the stack before it is destructed.

Though it is safe for multiple threads to call the member functions concurrently, due to the use of locks, only one thread is ever actually doing any work in the stack data structure at a time. This *serialization* of threads can potentially limit the performance of an application where there is significant contention on the **stack**: whilst a thread is waiting for the lock, it is not doing any useful work. Also, the stack does not provide any means for waiting for an item to be added, so if a thread needs to wait it must periodically call **empty()**, or just call **pop()** and catch the **empty_stack** exceptions. This makes this stack implementation a poor choice if such a scenario is required as a waiting thread must either consume precious resources checking for data, or the user must write external wait and notification code (e.g. using condition variables) which might therefore render the internal locking unnecessary and therefore wasteful. The queue from chapter 4 shows a way of incorporating such waiting into the data structure itself using a condition variable inside the data structure, so let's look at that next.

6.2.2 A Thread-Safe Queue Using Locks and Condition Variables

The thread-safe queue from chapter 4 is reproduced in listing 6.2. Much like the stack was modelled after **std::stack<>**, this queue is modelled after **std::queue<>**. Again, the interface differs from that of the Standard container adaptor due to the constraints of writing a data structure that is safe for concurrent access from multiple threads.

Listing 6.27: The full class definition for a thread-safe queue using condition variables

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

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();                             #1
    }

    void wait_and_pop(T& value)                             #2
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()                       #3
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[]{return !data_queue.empty();}); #5
        std::shared_ptr<T> res(new T(data_queue.front()));
        data_queue.pop();
        return res;
    }

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

    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();                    #4
        std::shared_ptr<T> res(new T(data_queue.front()));
        data_queue.pop();
        return res;
    }

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

Cueballs in code and text

The structure of the queue implementation shown in listing 6.2 is very similar to the stack from listing 6.1, except for the call to **data_cond.notify_one()** in **push()** (#1) and the **wait_and_pop()** functions (#2, #3). The two overloads of **try_pop()** are almost identical to the **pop()** functions from listing 6.1, except that they don't throw an exception if the queue is empty — instead they return either a **bool** value indicating whether or not a value was retrieved or a **NULL** pointer if no value could be retrieved by the pointer-returning overload (#4). This would actually have been a valid way of implementing the stack too. So, if we exclude the **wait_for_pop()** functions, then the analysis we did for the stack applies just as well here.

The new **wait_for_pop()** functions are a solution to the problem of waiting for a queue entry that we saw with the stack: rather than continuously calling **empty()**, the waiting thread can just call **wait_for_pop()** and the data structure will handle the waiting with a condition variable. The call to **data_cond.wait()** will not return until the underlying queue has at least one element, so we don't have to worry about the possibility of an empty queue at this point in the code, and the data is still protected with the lock on the mutex. These functions don't therefore add any new race conditions or possibilities for deadlock, and the invariants will be upheld.

There is a slight twist with regards to exception safety in that if there is more than one thread waiting when an entry is pushed onto the queue only one thread will be woken by the call to **data_cond.notify_one()**. However, if that thread then throws an exception in **wait_and_pop()**, such as when the new **std::shared_ptr<>** is constructed (#5), then none of the other threads will be woken. If this is not acceptable, the call is readily replaced with **data_cond.notify_all()**, which will wake all the threads, but at the cost of most of them then going back to sleep when they find that the queue is empty after all. A second alternative is to have **wait_and_pop()** call **notify_one()** if an exception is thrown, so that another thread can attempt to retrieve the stored value. A third alternative is to move the **std::shared_ptr<>** initialization to the **push()** call and store **std::shared_ptr<>** instances rather than direct data values. Copying the **std::shared_ptr<>** out of the internal **std::queue<>** then cannot throw an exception, so **wait_and_pop()** is safe again. Listing 6.3 shows the queue implementation revised with this in mind:

Listing 6.28: A thread-safe queue holding std::shared_ptr<> instances for exception-safety

```
    template<typename T>
```

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

    void push(T new_value)
    {
        std::shared_ptr<T> data(new T(new_value));            #5
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }

    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[]{return !data_queue.empty();});
        value=*data_queue.front();                            #1
        data_queue.pop();
    }

    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[]{return !data_queue.empty();});
        std::shared_ptr<T> res=data_queue.front();            #3
        data_queue.pop();
        return res;
    }

    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=*data_queue.front();                            #2
        data_queue.pop();
    }

    std::shared_ptr<T> try_pop()
```

```
        {
            std::lock_guard<std::mutex> lk(mut);
            if(data_queue.empty())
                return std::shared_ptr<T>();
            std::shared_ptr<T> res=data_queue.front();           #4
            data_queue.pop();
            return res;
        }

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

Cueballs in code and text

The basic consequences of holding the data by **std::shared_ptr<>** are straightforward: the pop functions that take a reference to a variable to receive the new value now have to dereference the stored pointer (#1, #2), and the pop functions that return a **std::shared_ptr<>** instance can just retrieve it from the queue (#3, #4) before returning it to the caller.

If the data is held by **std::shared_ptr<>** then there is also an additional benefit: the allocation of the new instance can now be done outside the lock in **push()** (#5), whereas in listing 6.2 it had to be done whilst the lock in **pop()**. Since memory allocation is typically quite an expensive operation this can be quite beneficial for the performance of the queue, as it reduces the time the mutex is held, allowing other threads to perform operations on the queue in the mean time.

Just like in the stack example, the use of a mutex to protect the entire data structure limits the concurrency supported by this queue: though multiple threads might be blocked on the queue in various member functions, only one thread can be doing any work at a time. However, part of this restriction comes from the use of **std::queue<>** in the implementation: by using the standard container we now have essentially one data item which is either protected or not. By taking control of the detailed implementation of the data structure we can provide more fine-grained locking, and thus allow a higher level of concurrency.

6.2.3 A Thread-Safe Queue Using Fine-grained Locks and Condition Variables

In listings 6.2 and 6.3 we have one protected data item (**data_queue**), and thus one mutex. In order to use finer-grained locking we need to look inside the queue at its constituent parts and associate one mutex with each distinct data item.

The simplest data structure for a queue is a singly-linked list. The queue contains a "head" pointer which points to the first item in the list, and each item then points to the next item. Data items are removed from the queue by replacing the "head" pointer with the pointer to the next item and then returning the data from the old "head".

Items are added to the queue at the other end. In order to do this, the queue also contains a "tail" pointer which refers to the last item in the list. New nodes are added by changing the "next" pointer of the last item to point to the new node and then updating the "tail" pointer to refer to the new item. When the list is empty both the "head" and "tail" pointers are **NULL**.

Listing 6.4 shows a simple implementation of such a queue based on a cut-down version of the interface to the queue in listing 6.2: we only have one **try_pop()** function and no **wait_and_pop()** as this queue only supports single-threaded use.

Listing 6.29: A simple queue implementation

```
template<typename T>
class queue
{
private:
    struct node
    {
        T data;
        node* next;

        node(T data_):
            data(data_),next(0)
        {}
    };

    node* head;                                                  #1
    node* tail;                                                  #2

public:
    queue():
        head(0),tail(0)
    {}

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
```

```
        ~queue()
        {
            while(head)
            {
                node* const old_head=head;
                head=old_head->next;
                delete old_head;
            }
        }

        std::shared_ptr<T> try_pop()
        {
            if(!head)
            {
                return std::shared_ptr<T>();
            }
            std::shared_ptr<T> const res(new T(head->data));
            node* const old_head=head;
            head=old_head->next;                                        #6
            delete old_head;
            return res;
        }

        void push(T new_value)
        {
            std::auto_ptr<node> p(new node(new_value));
            if(tail)
            {
                tail->next=p.get();                                    #5
            }
            else
            {
                head=p.get();                                          #3
            }
            tail=p.release();                                          #4
        }
    };
```

Though this works fine in a single-threaded context, there's a couple of things that will cause us problems if we try to use fine-grained locking in a multi-threaded context. Since we have two data items (**head** (#1) and **tail** (#2)), we could in principle use two mutexes: one to protect **head**, and one to protect **tail**. However, there are a couple of problems with that.

The most obvious problem is that **push()** can modify both **head** (#3) and **tail** (#4), so it would have to lock both mutexes. This isn't too much of a problem, though it is unfortunate, as locking both mutexes would be possible. The critical problem is that both **push()** and **pop()** access the **next** pointer of a node: **push**() updates **tail->next** (#5), and **try_pop()** reads **head->next** (#6). If there is a single item in the queue then

head==tail, so both **head->next** and **tail->next** are the same object, which therefore requires protection. Since we can't tell if it's the same object without reading both **head** and **tail**, now have to lock the same mutex in both **push()** and **try_pop()**, so we're no better than before. Is there a way out of this dilemma?

Enabling concurrency by separating data

We can solve this problem by pre-allocating a dummy node with no data to ensure that there is always at least one node in the queue to separate the node being accessed at the head from that being accessed at the tail. For an empty queue, **head** and **tail** now both point to the dummy node rather than being **NULL**. This is fine, since **try_pop()** doesn't access **head->next** if the queue is empty. If we add a node to the queue (so there is one real node), then **head** and **tail** now point to separate nodes, so there is no race on **head->next** and **tail->next**. The downside is that we have to add an extra level of indirection to store the data by pointer in order to allow the dummy nodes. Listing 6.5 shows how the implementation looks now.

Listing 6.30: A simple queue with a dummy node

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;                        #2
        node* next;

        node():
            next(0)
        {}
    };

    node* head;
    node* tail;

public:
    queue():
        head(new node),tail(head)                       #7
    {}

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    ~queue()
    {
        while(head)
        {
            node* const old_head=head;
            head=old_head->next;
            delete old_head;
        }
    }

    std::shared_ptr<T> try_pop()
    {
        if(head==tail)                                   #1
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(head->data);        #3
        node* const old_head=head;
        head=old_head->next;                             #8
        delete old_head;                                 #9
        return res;
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(new T(new_value));   #4
        std::auto_ptr<node> p(new node);                 #5
        tail->data=new_data;                             #6
        tail->next=p.get();
        tail=p.release();
    }
};
```

The changes to **try_pop()** are fairly minimal. Firstly, we're comparing **head** against **tail** (#1) rather than checking for **NULL**, since the dummy node means that **head** is never **NULL.** Secondly, because the **node** now stores the data by pointer (#2) we can just retrieve the pointer directly (#3) rather than having to construct a new instance of **T**. The big changes are in **push()**: we must first create a new instance of **T** on the heap and take ownership of it in a **std::shared_ptr<>** (#4). The new node we're creating is actually going to be the new dummy node, so we don't need to supply the **new_value** to the constructor (#5). Instead, we set the data on the old dummy node to our newly allocated copy of the **new_value** (#6). Finally, in order to have a dummy node we have to create it in the constructor (#7).

By now, I'm sure you're wondering what these changes buy us, and how they help with making the queue thread-safe. Well, **push()** now only accesses **tail**, not **head**, which is an improvement. **try_pop()** accesses both **head** and **tail**, but **tail** is only needed for the initial comparison, so the lock is only very short lived. The big gain is that the dummy node means **try_pop()** and **push()** are never operating on the same node, so we no longer need an overarching mutex. So, we can have one mutex for **head** and one for **tail**. Where do we put the locks?

We're aiming for the maximum opportunities for concurrency, so we want to hold the locks for the smallest possible length of time. **push()** is easy: the mutex needs to be locked across all accesses to **tail**, which means we lock the mutex after the new node is allocated (#5) and before we assign the data to the current tail node (#6). The lock then needs to be held until the end of the function.

try_pop() is not so easy. First off we need to lock the mutex on **head**, and hold it until we're done with **head**. In essence this is the mutex to determine which thread does the popping, so we want to do that first. Once **head** is changed (#8) we can unlock the mutex: it doesn't need to be locked whilst we delete the old head node (#9). That leaves the access to **tail** needing a lock on the tail mutex. Since we only need to access **tail** once, we can just acquire the mutex for the time it takes to do the read. This is best done by wrapping it in a function. In fact, since the code that needs the **head** mutex locked is only a subset of the member it is clearer to wrap that in a function too. The final code is shown in listing 6.6.

Listing 6.31: A thread-safe queue with fine-grained locking

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;

        node():
            next(NULL)
        {}
    };

    std::mutex head_mutex;
    node* head;
    std::mutex tail_mutex;
    node* tail;

    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    node* pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);
        if(head==get_tail())
        {
            return NULL;
        }
        node* const old_head=head;
        head=old_head->next;
        return old_head;
    }


public:
    queue():
        head(new node),tail(head)
    {}

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    ~queue()
    {
        while(head)
        {
            node* const old_head=head;
            head=old_head->next;
            delete old_head;
        }
    }

    std::shared_ptr<T> try_pop()
    {
        node* old_head=pop_head();
        if(!old_head)
        {
            return std::shared_ptr<T>();
        }

        std::shared_ptr<T> const res(old_head->data);
        delete old_head;
        return res;
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(new T(new_value));
        std::auto_ptr<node> p(new node);
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        tail->data=new_data;
        tail->next=p.get();
        tail=p.release();
    }
};
```

So, let us look at this code with a critical eye, thinking about the guidelines listed in 6.1.1. Before we look for broken invariants, we'd best be sure what they are:

- **tail->next==NULL**
- **tail->data==NULL**
- **head==tail** implies an empty list
- A single element list has **head->next==tail**
- For each node **x** in the list, where **x!=tail**, **x->data** points to an instance of **T**, and **x->next** points to the next node in the list. **x->next==tail** implies **x** is the last node in the list.
- Following the **next** nodes from **head** will eventually yield **tail**.

On it's own, **push()** is straightforward: the only modifications to the data structure are protected by **tail_mutex**, and they uphold the invariant, since the new tail node is an empty node, and **data** and **next** are correctly set for the old tail node, which is now the last real node in the list.

The interesting part is **try_pop()**. It turns out that not only is the lock on **tail_mutex** necessary to protect the read of **tail** itself, it's also necessary to ensure that we don't get a data race reading the data from the head. If we didn't have that mutex then it would be quite possible for a thread to call **try_pop()** and a thread to call **push()** concurrently, and there would be no defined ordering on their operations. Even though each member function holds a lock on a mutex, they hold locks on *different* mutexes, and they potentially access the same data: all data in the queue originates from a call to **push()** after all. Since the threads would be potentially accessing the same data without a defined ordering, this would be a data race as we saw in chapter 5, and undefined behaviour. Thankfully the lock on the **tail_mutex** in **get_tail()** solves everything. Since the call to **get_tail()** locks the same mutex as the call to **push()**, there is a defined order between the two calls. Either the call to **get_tail()** occurs *before* the call to **push()**, in which case it sees the old value of **tail**, or it occurs *after* the call to **push()**, in which case it sees the new value of **tail**, *and the new data attached to the previous value of* *tail*.

It is also important that the call to **get_tail()** occurs *inside* the lock on **head_mutex**. If it did not, then the call to **pop_head()** could be stuck in between the call to **get_tail()** and the lock on the **head_mutex**, as other threads called **try_pop()** (and thus **pop_head()**) and acquired the lock first, thus preventing our initial thread from making progress.

```
        node* pop_head()                                            #A
        {
            node* const old_tail=get_tail();                        #1
            std::lock_guard<std::mutex> head_lock(head_mutex);

            if(head==old_tail)                                     #2
            {
                return NULL;
            }
            node* const old_head=head;
            head=old_head->next;                                   #3
            return old_head;
        }
```

#1 Get the old tail value outside the lock on `head_mutex`
#A This is a broken implementation

Cueballs in code and text

In this scenario we might find that both **head** and **tail** have changed by the time our initial thread can acquire the lock on **head_mutex**, and not only is the returned tail node no longer the tail, it is no longer even part of the list. This could then mean that the comparison of **head** to **old_tail** (#2) fails, even if **head** really is the last node. Consequently when we update **head** (#3) we may end up moving **head** beyond **tail**, and off the end of the list, destroying the data structure. By moving the call to **get_tail()** inside the lock on **head_mutex** (as in listing 6.6), we ensure that no other threads can changed **head**, and **tail** only ever moves further away, which is perfectly safe: **head** can never pass the value returned from **get_tail()**, so the invariants are upheld.

Once **pop_head()** has removed the node from the queue by updating **head**, the mutex is unlocked, and **try_pop()** can extract the data and delete the node if there was one, and return a **NULL** instance of **std::shared_ptr<>** if not, all in complete safety, knowing it is the only thread that can access this node.

Next up, the external interface is a subset of that from listing 6.2, so the same analysis applies: there are no race conditions inherent in the interface.

Exceptions are more interesting. Since we've changed the data allocation patterns, the exceptions can now come from different places. The only operations in **try_pop()** that can throw exceptions are the mutex locks, and the data is not modified until the locks are acquired. Therefore **try_pop()** is exception safe. On the other hand, **push()** allocates a new instance of **T** on the heap, and a new instance of **node**, either of which might throw an exception. However, both of the newly-allocated objects are assigned to smart pointers, so they will be freed if an exception is thrown. Once the lock is acquired, none of the remaining operations in **push()** can throw an exception, so again we're home and dry and **push()** is exception safe too.

Because we've not changed the interface, there are no new external opportunities for deadlock. There are no internal opportunities either: the only place that two locks are acquired is in **pop_head()**, which always acquires the **head_mutex** and then the **tail_mutex**, so this will never deadlock.

The remaining question is regards to the actual possibilities for concurrency. This data structure actually has considerably more scope for concurrency than that from listing 6.2, since the locks are more fine-grained, and *more is done outside the locks*. For example, in **push()**, the new node and new data item are allocated with no locks held. This means that multiple threads can be allocating new nodes and data items concurrently without a problem. Only one thread can add its new node to the list at a time, but the code to do so is only a few simple pointer assignments, so the lock is not held for much time at all compared to the **std::queue<>**-based implementation where the lock is held around all the memory allocation operations internal to the **std::queue<>**.

Also, **try_pop()** only holds the **tail_mutex** for a very short time, simply to protect a read from **tail**. Consequently a call to **try_pop()** can occur almost entirely concurrently with a call to **push()**. Also, the operations performed whilst holding the **head_mutex** are also quite minimal: the expensive **delete** is outside the lock. This will increase the number of calls to **try_pop()** that can happen concurrently: only one thread can call **pop_head()** at a time, but multiple threads can then delete their old nodes and return the data safely.

Waiting for an item to pop

OK, so listing 6.6 provides a thread-safe queue with fine-grained locking, but it only supports **try_pop()** (and only one overload at that). What about the handy **wait_and_pop()** functions we had back in listing 6.2? Can we implement an identical interface with our fine-grained locking?

Of course, the answer is “yes”, but the real question is “how?” Modifying **push()** is easy: just add the **data_cond.notify_one()** call at the end of the function, just like in listing 6.2. Actually, it's not quite that simple: we're using fine-grained locking because we want the maximum possible amount of concurrency. If we leave the mutex locked across the call to **notify_one()** (as in listing 6.2) then if the notified thread wakes up before the mutex has been unlocked it will have to wait for the mutex. On the other hand, if we unlock the mutex *before* we call **notify_one()** then the mutex is available for the waiting thread to acquire when it wakes up (assuming no other thread locks it first). This is a minor improvement, but it might be important in some cases.

wait_and_pop() is more complicated, as we have to decide where to wait, what the predicate is, and which mutex needs to be locked. The condition we're waiting for is “queue not empty”, which is represented by **head!=tail**. Written like that it would require both **head_mutex** and **tail_mutex** to be locked, but we've already decided in listing 6.6 that we only need to lock **tail_mutex** for the read of **tail**, and not for the comparison itself, so we can apply the same logic here. If we make the predicate **head!=get_tail()** then we only need to hold the **head_mutex**, so we can use our lock on that for the call to **data_cond.wait()**. Once we've added the wait logic, the implementation is the same as **try_pop()**.

The second overload of **try_pop()** and the corresponding **wait_for_pop()** overload requires careful thought. If we just replace the return of the **std::shared_ptr<>** retrieved from **old_head** with a copy-assignment to the **value** parameter then there is a potential exception-safety issue. At this point, the data item has been removed from the queue and the mutex unlocked; all that remains is to return the data to the caller. However, if the copy-assignment throws an exception (as it very well might), then the data item is lost, as it can't be returned to the queue in the same place.

If the actual type **T** used for the template argument has a no-throw move-assignment operator, or a no-throw swap operation then we could use that, but we'd really like a general solution that could be used for any type **T**. In which case, we have to move the potentially-throwing inside the locked region, before the node is removed from the list. This means we need an extra overload of **pop_head()** which retrieves the stored value prior to modifying the list.

In comparison, **empty()** is trivial: just lock **head_mutex** and check for **head==get_tail()**. The final code is shown in listing 6.7.

Listing 6.32: A thread-safe queue with fine-grained locking and waiting

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;

        node():
            next(NULL)
        {}
    };

    std::mutex head_mutex;
    node* head;
    std::mutex tail_mutex;
    node* tail;
    std::condition_variable data_cond;

    node* get_tail()
    {
        std::lock_guard<std::mutex> tail_lock(tail_mutex);
        return tail;
    }

    node* pop_head()                                            #1
    {
        node* const old_head=head;
        head=old_head->next;
        return old_head;
    }

    node* try_pop_head()
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if(head==get_tail())
        {
            return NULL;
        }
        return pop_head();
    }

    node* try_pop_head(T& value)
    {
        std::lock_guard<std::mutex> head_lock(head_mutex);

        if(head==get_tail())
        {
            return NULL;
        }
        value=*head->data;
        return pop_head();
    }

    std::unique_lock<std::mutex> wait_for_data()                     #2
    {
        std::unique_lock<std::mutex> head_lock(head_mutex);
        data_cond.wait(head_lock,[&]{return head!=get_tail();});
        return head_lock;                                            #3
    }

    node* wait_pop_head()
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());  #4
        return pop_head();
    }

    node* wait_pop_head(T& value)
    {
        std::unique_lock<std::mutex> head_lock(wait_for_data());  #5
        value=*head->data;
```

```
            return pop_head();
        }

    public:
        queue():
            head(new node),tail(head)
        {}

        queue(const queue& other)=delete;
        queue& operator=(const queue& other)=delete;

        ~queue()
        {
            while(head)
            {
                node* const old_head=head;
                head=old_head->next;
                delete old_head;
            }
        }

        std::shared_ptr<T> try_pop()
        {
            node* const old_head=try_pop_head();
            if(!old_head)
            {
                return std::shared_ptr<T>();
            }

            std::shared_ptr<T> const res(old_head->data);
            delete old_head;
            return res;
        }

        bool try_pop(T& value)
        {
            node* const old_head=try_pop_head(value);
            if(!old_head)
            {
                return false;
            }
            delete old_head;
            return true;
        }

        std::shared_ptr<T> wait_and_pop()
        {
            node* const old_head=wait_pop_head();
            std::shared_ptr<T> const res(old_head->data);
            delete old_head;
            return res;
        }

        void wait_and_pop(T& value)
        {
            node* const old_head=wait_pop_head(value);
            delete old_head;
        }

        void push(T new_value)
        {
            std::shared_ptr<T> new_data(new T(new_value));
            std::auto_ptr<node> p(new node);
            {
                std::lock_guard<std::mutex> tail_lock(tail_mutex);
                tail->data=new_data;
                tail->next=p.get();
                tail=p.release();
            }
            data_cond.notify_one();
        }

        void empty()
        {
            std::lock_guard<std::mutex> head_lock(head_mutex);
            return (head==get_tail());
        }
    };
```

Cueballs in code and text

The full implementation shown in listing 6.7 has several little helper functions to simplify the code and reduce duplication, such as **pop_head()** (#1) and **wait_for_data()** (#2), which actually modify the list to remove the head item and wait for the queue to have some data to pop respectively. **wait_for_data()** is particularly noteworthy, as not only does it wait on the condition variable using a lambda function for the predicate, but it returns the lock instance to the caller (#3). This is to ensure that the same lock is held whilst the data is modified by the relevant **wait_pop_head()** overload (#4, #5).

This queue implementation will serve as the basis for the lock-free queue I'll be covering in chapter 7, but for now let's move beyond queues and onto more complex data structures.

6.3 Designing More Complex Lock-based Data Structures

Stacks and queues are simple: the interface is exceedingly limited, and they are very tightly focused on a specific purpose. Not all data structures are that simple; most data structures support a variety of operations. In principle, this can then lead to greater opportunities for concurrency, but it also makes the task of protecting the data that much harder as the multiple access patterns need to be taken into account. The precise nature of the various operations that can be performed is important when designing such data structures for concurrent access.

To see some of the issues involved, let's take a look at the design of a lookup table.

6.3.1 Writing a thread-safe Lookup Table Using Locks

A lookup table or dictionary associates values of one type (the key type) with values of either the same or a different type (the mapped type). In general, the intention behind such a structure is to allow code to query the data associated with a given key. In the C++ standard library this facility is provided by the associative containers: **`std::map<>`**, **`std::multimap<>`**, **`std::unordered_map<>`**, and **`std::unordered_multimap<>`**.

A lookup table has a different usage pattern to a stack or a queue. Whereas almost every operation on a stack or a queue modifies it in some way, either to add an element or remove one, a lookup table might be modified very rarely. The simple DNS cache in listing 3.11 is one example of such a scenario, which features a hugely reduced interface compared to **`std::map<>`**. As we saw with the stack and queue, the interfaces of the standard containers are not suitable when the data structure is to be accessed from multiple threads concurrently, as there are inherent race conditions in the interface design, so they need to be cut down and revised.

The biggest problem with the **`std::map<>`** interface from a concurrency perspective is the iterators: though it is possible to have an iterator that provides safe access into a container even when other threads can access (and modify) the container, this is a tricky proposition. Correctly handling iterators requires you to deal with issues such as another thread deleting the element that the iterator is referring to, which can get rather involved. For the first cut at a thread-safe lookup table interface we'll skip the iterators. Given that the interface to **`std::map<>`** (and the other associative containers in the standard library) is so heavily iterator-based, it's probably worth leaving them aside, and designing the interface from the ground up.

There are only a few basic operations on a lookup table:

- Add a new key/value pair;

- Change the value associated with a given key;
- Remove a key and its associated value; and
- Obtain the value associated with a given key *if any.*

There's also a few container-wide operations that might be useful, such as a check on whether or not the container is empty, a snapshot of the complete list of keys, or a snapshot of the complete set of key/value pairs.

If we stick to the simple thread-safety guidelines such as not returning references, and put a simple mutex lock around the entirety of each member function, then all of these are safe: they either come before some modification from another thread, or after it. The biggest potential for a race condition is when a new key/value pair is being added: if two threads add a new value, only one will be first, and the second will therefore fail. One possibility is to combine add and change into a single member function, as we did for the DNS cache in listing 3.11.

The only other interesting point from an interface perspective is the *if any* part of obtaining an associated value. One option is to allow the user to provide a “default” result which is returned in the case that the key is not present:

```
    mapped_type get_value(key_type const& key, mapped_type default_value);
```

In this case, a default-constructed instance of **mapped_type** could be used if the **default_value** was not explicitly provided. This could also be extended to return a **std::pair<mapped_type,bool>** instead of just an instance of **mapped_type**, where the **bool** indicates whether or not the value was present. Another option is to return a smart pointer referring to the value: if the pointer value is **NULL** then there was no value to return.

As already mentioned, once the interface has been decided the thread-safety could be guaranteed by using a single mutex and a simple lock around each and every member function to protect the underlying data structure. However, this would squander the possibilities for concurrency provided by the separate functions for reading the data structure and modifying it. One option is to use a mutex that supports multiple reader threads or a single writer thread, such as the **boost::shared_mutex** used in listing 3.11. Though this would indeed improve the possibilities for concurrent access, only one thread could modify the data structure at a time. Ideally, we'd like to do better than that.

Designing a Map Data Structure for Fine-grained Locking

As with the queue discussed in section 6.2.3, in order to permit fine-grained locking we need to look carefully at the details of the data structure rather than just wrapping a pre-existing container such as **std::map<>**. There are three common ways of implementing an associative container like our lookup table:

- A binary tree, such as a red-black tree;

- A sorted array; and
- A hash table.

A binary tree doesn't really provide much scope for extending the opportunities for concurrency: every lookup or modification has to start by accessing the root node, which therefore has to be locked. Though this lock can be released as the accessing thread moves down the tree, this is therefore not much better than a single lock across the whole data structure.

A sorted array is even worse, as you cannot tell in advance where in the array a given data value is going to be, so you need a single lock for the whole array.

That leaves the hash table. Assuming a fixed number of buckets, which bucket a key belongs to is purely a property of the key and its hash function. This therefore means we can safely have a separate lock per bucket. If we again use a mutex that supports multiple readers or a single writer then we've just increased the opportunities for concurrency *N*-fold, where *N* is the number of buckets. The downside is that we need a good hash function for the key. The C++ standard library provides us with the **std::hash<>** template which we can use for this purpose: it is already specialized for the fundamental types such as **int**, and common library types such as **std::string**, and the user can easily specialize it for other key types. If we follow the lead of the standard unordered containers and take the type of the function object to use for doing the hashing as a template parameter then the user can choose whether to specialize **std::hash<>** for their key type or provide a separate hash function.

So, let's look at some code. What might the implementation of a thread-safe lookup table look like? One possibility is shown in listing 6.8.

Listing 6.33: A thread-safe lookup table

```
template<typename Key,typename Value,typename Hash=std::hash<Key> >
class lookup_table
{
private:
    typedef std::pair<Key,Value> bucket_value;
    typedef std::list<bucket_value> bucket_data;
    typedef typename bucket_data::iterator bucket_iterator;

    struct bucket_type
    {
        bucket_data data;
        boost::shared_mutex mutex;                                  #2

        bucket_iterator find_entry_for(Key const& key)              #10
        {
            return std::find_if(data.begin(),data.end(),
                                [&](bucket_value const& item)
                                {return item.first==key;});
        }
    };

    std::vector<bucket_type*> buckets;                                #1
    Hash hasher;

    bucket_type& get_bucket(Key const& key) const                     #3
    {
        std::size_t const bucket_index=hasher(key)%buckets.size();
        return *buckets[bucket_index];
    }

public:
    typedef Key key_type;
    typedef Value mapped_type;
    typedef Hash hash_type;

    lookup_table(unsigned num_buckets=19, Hash const& hasher_=Hash()):
        buckets(num_buckets),hasher(hasher_)
    {
        for(unsigned i=0;i<num_buckets;++i)
        {
            buckets[i]=new bucket_type;
        }
    }

    lookup_table(lookup_table const& other)=delete;
    lookup_table& operator=(lookup_table const& other)=delete;

    ~lookup_table()
    {
        for(unsigned i=0;i<buckets.size();++i)
        {
            delete buckets[i];
        }
    }

    Value value_for(Key const& key,
                    Value const& default_value=Value()) const
    {
        bucket_type& bucket=get_bucket(key);                           #4
        boost::shared_lock<boost::shared_mutex> lock(bucket.mutex);    #7
        bucket_iterator const found_entry=bucket.find_entry_for(key);
        return (found_entry==bucket.data.end())?
            default_value : found_entry->second;
    }
    void add_or_update_mapping(Key const& key,Value const& value)
    {
        bucket_type& bucket=get_bucket(key);                           #5
        std::unique_lock<boost::shared_mutex> lock(bucket.mutex);      #8
        bucket_iterator const found_entry=bucket.find_entry_for(key);
        if(found_entry==bucket.data.end())
        {
            bucket.data.push_back(bucket_value(key,value));
        }
        else
        {
            found_entry->second=value;
        }
    }

    void remove_mapping(Key const& key)
    {
        bucket_type& bucket=get_bucket(key);                           #6
        std::unique_lock<boost::shared_mutex> lock(bucket.mutex);      #9
        bucket_iterator const found_entry=bucket.find_entry_for(key);
        if(found_entry!=bucket.data.end())
        {
            bucket.data.erase(found_entry);
        }
    }
};
```

Cueballs in code and text

This implementation uses a **std::vector<bucket_type*>** (#1) to hold the buckets, which allows the number of buckets to be specified in the constructor. The default is 19, which is an arbitrary prime number — hash tables work best with a prime number of buckets. Each bucket is protected with an instance of **boost::shared_mutex** (#2) to allow many concurrent reads or a single call to either of the modification functions *per bucket*.

Since the number of buckets is fixed, the **get_bucket()** function (#3) can be called without any locking (#4, #5, #6), and then the bucket mutex can be locked either for shared (read-only) ownership (#7) or unique (read/write) ownership (#8, #9) as appropriate for each function.

All three functions make use of the **find_entry_for()** member function (#10) on the bucket to determine whether or not the entry is in the bucket. Each bucket just contains a **std::list<>** of key/value pairs so adding and removing entries is easy.

We've already covered the concurrency angle, and everything is suitably protected with mutex locks, so what about exception safety? **value_for** doesn't modify anything, so that's fine: if it throws an exception it won't affect the data structure. **remove_mapping** modifies the list with the call to **erase**, but this is guaranteed not to throw, so that's safe. This leaves **add_or_update_mapping**, which might throw in either of the two branches of the **if**. **push_back** is exception-safe, and will leave the list in the original state if it throws, so that branch is fine. The only problem is with the assignment in the case that we're replacing an existing value: if the assignment throws then we're relying on it leaving the original unchanged. However, this doesn't affect the data structure as a whole, and is entirely a property of the user-supplied type, so we can safely leave it up to the user to handle this.

At the beginning of this section, I mentioned that one nice-to-have feature of such a lookup table would be the option of retrieving a snapshot of the current state into for example a **std::map<>**. This would require locking the entire container, in order to ensure that a consistent copy of the state was retrieved, which requires locking all the buckets. Since the “normal” operations on the lookup table only require a lock on one bucket at a time, this would be the only operation that required a lock on all the buckets. Therefore, provided we lock them in the same order every time (e.g. increasing bucket index) there will be no opportunity for deadlock. Such an implementation is shown in listing 6.9.

Listing 6.34: Obtaining the contents of a `lookup_table` as a `std::map<>`

```
std::map<Key,Value> lookup_table::get_map() const
{
    std::vector<std::unique_lock<boost::shared_mutex> > locks;
    for(unsigned i=0;i<buckets.size();++i)
    {
        locks.push_back(
            std::unique_lock<boost::shared_mutex>(buckets[i].mutex));
    }
    std::map<Key,Value> res;
    for(unsigned i=0;i<buckets.size();++i)
    {
        for(bucket_iterator it=buckets[i].data.begin();
            it!=buckets[i].data.end();
            ++it)
        {
            res.insert(*it);
        }
    }
    return res;
}
```

The lookup table implementation from listing 6.8 increases the opportunity for concurrency of the lookup table as a whole by locking each bucket separately, and by using a **boost::shared_mutex** to allow reader concurrency on each bucket. However, what if we could increase the potential for concurrency on a bucket by even finer grained locking? In the next section we'll do just that by looking at a thread-safe list container with iterator support.

6.3.2 Writing a Thread-Safe List Using Locks

A list is one of the most basic data structures, so it should be straightforward to write a thread-safe one, shouldn't it? Well, that depends on what facilities you're after, and we need one that offers iterator support — something we shied away from adding to our map on the basis that it was too complicated. The basic issue with STL-style iterator support is that the iterator must hold some kind of reference into the internal data structure of the container. If the container can be modified from another thread, this reference must somehow remain valid, which essentially requires the iterator hold a lock on some part of the structure. Since the lifetime of an STL-style iterator is completely outside the control of the container, this is a bad idea.

The alternative is to provide iteration functions such as **`for_each`** as part of the container itself. This puts the container squarely in charge of the iteration and locking, but does fall foul of our deadlock avoidance guidelines from chapter 3: in order for **`for_each`** to do anything useful, it must call user-supplied code whilst holding the internal lock. Not only that, but it must also pass a reference to each item to this user-supplied code in order that the user-supplied code might work on this item. This could be avoided by passing a copy of each item to the user-supplied code, but that would be expensive if the data items were large.

So, for now we'll leave it up to the user to ensure that they don't cause deadlock by acquiring locks in the user-supplied operations, and don't cause data races by storing the references for access outside the locks. In the case of the list being used by the lookup table this is perfectly safe, as we know we're not going to do anything naughty.

That leaves us with the question of which operations to supply for our list. If we cast our eyes back on listings 6.8 and 6.9, then we can see the sorts of operations we require:

- add an item to the list;
- remove an item from the list if it meets a certain condition;
- find an item in the list that meets a certain condition;
- update an item that meets a certain condition;
- copy each item in the list to another container.

For this to be a good general-purpose list container, it would be helpful to add further operations such as a positional insert, but this is unnecessary for our lookup table, so I will leave it as an exercise for the reader.

The basic idea with fine-grained locking for a linked list is to have one mutex per node. If the list gets big, that's a lot of mutexes! However, the benefit here is that operations on separate parts of the list are truly concurrent: each operation only holds the locks on the nodes it's actually interested in, and unlocks each node as it moves on to the next. Listing 6.10 shows an implementation of just such a list.

Listing 6.35: A thread-safe list with iteration support

```
template<typename T>
class concurrent_list
{
    struct node                                                    #1
    {
        std::mutex m;
        std::shared_ptr<T> data;
        node* next;

        node():                                                    #2
            next(NULL)
        {}

        node(T const& value):                                      #4
            data(new T(value))
        {}
    };

    node head;

public:
    concurrent_list()
    {}

    concurrent_list(concurrent_list const& other)=delete;
    concurrent_list& operator=(concurrent_list const& other)=delete;

    ~concurrent_list()
    {
        remove_if([](T const&){return true;});
    }

    void push_front(T const& value)
    {
        std::auto_ptr<node> new_node(new node(value));             #3
        std::lock_guard<std::mutex> lk(head.m);
        new_node->next=head.next;                                  #5
        head.next=new_node.release();                              #6
    }

    template<typename Function>
    void for_each(Function f)                                      #7
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);                  #8
        while(node* const next=current->next)                     #9
        {
            std::unique_lock<std::mutex> next_lk(next->m);        #10
            lk.unlock();                                          #11
            f(*next->data);                                       #12
            current=next;
            lk=std::move(next_lk);                                #13
        }
    }

    template<typename Predicate>
    std::shared_ptr<T> find_first_if(Predicate p)                #14
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next)
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            lk.unlock();
            if(p(*next->data))                                   #15
            {
                return next->data;                               #16
            }
            current=next;
            lk=std::move(next_lk);
        }
        return std::shared_ptr<T>();
    }

    template<typename Predicate>
    void remove_if(Predicate p)                                  #17
    {
        node* current=&head;
        std::unique_lock<std::mutex> lk(head.m);
        while(node* const next=current->next)
        {
            std::unique_lock<std::mutex> next_lk(next->m);
            if(p(*next->data))                                   #18
            {
                current->next=next->next;                        #19
                next_lk.unlock();
                delete next;                                     #20
            }
            else
            {
                lk.unlock();                                     #21
                current=next;
                lk=std::move(next_lk);
            }
```

```
            }
        }
    };
```

Cueballs in code and text

The **concurrent_list<>** from listing 6.10 is a singly-linked list, where each entry is a **node** structure (#1). A default-constructed **node** is used for the **head** of the list, which starts with a **NULL next** pointer (#2). New nodes are added with the **push_front()** function: first a new node is constructed (#3), which allocates the stored data on the heap (#4), whilst leaving the **next** pointer uninitialized. We then need to acquire the lock on the mutex for the **head** node in order to get the appropriate **next** value (#5), and insert the node at the front of the list by setting **head.next** to point to our new node (#6). So far so good: we only need to lock one mutex in order to add a new item to the list, so there's no risk of deadlock; and the slow memory allocation happens outside the lock so the lock is only protecting the update of a couple of pointer values which can't fail. On to the iterative functions.

First up let's take a look at **for_each()** (#7). This operation takes a **Function** of some type to apply to each element in the list; in common with most standard library algorithms it takes this function by value, and will work with either a genuine function or an object of a type with a function call operator. In this case the function must accept a value of type **T** as the sole parameter. Here's where we do the hand-over-hand locking. To start with, we lock the mutex on the **head** node (#8). It's then safe to obtain the pointer to the **next** node. If that pointer is not **NULL** (#9) then we lock the mutex on that node (#10) in order to process the data. Once we've got the lock on that node we can release the lock on the previous node (#11) and call the specified function (#12). Once the function completes we can update the **current** pointer to the node we just processed and **move** the ownership of the lock from **next_lk** out to **lk** (#13). Since **for_each** passes each data item directly to the supplied **Function**, this can be used to update the items if necessary, or copy them into another container, or whatever. This is entirely safe, because the mutex for the node holding the data item is held across the call.

find_first_if() (#14) is very similar to **for_each()**: the crucial difference is that the supplied **Predicate** must return **true** to indicate a match or **false** to indicate no match (#15). Once we've got a match then we just return the found data (#16) rather than continuing to search. We could do this with **for_each()**, but it would needlessly continue processing the rest of the list even once a match had been found.

remove_if() (#17) is slightly different, because this function has to actually update the list: we can't use **for_each()** for this. If the **Predicate** returns **true** (#18) then we remove the node from the list by updating **current->next** (#19). Once we've done that, then we can release the lock held on the mutex for the **next** node and then delete the node

(#20). In this case, we don't update **current** because we need to check the new **next** node. If the **Predicate** returns **false** then we just want to move on as before (#21).

So, are there any deadlocks or race conditions with all these mutexes? The answer here is quite definitely **no**, provided that the supplied predicates and functions are well-behaved. The iteration is always one way, always starting from the **head** node, and always locks the next mutex before releasing the current one, so there is no possibility of different lock orders in different threads. The only potential candidate for a race condition is the deletion of the removed node in **remove_if()** (#20), as we do this after we've unlocked the mutex (it's undefined behaviour to destroy a locked mutex). However, a few moments thought reveals that this is indeed safe, since we *still* hold the mutex on the previous node (**current**), so no new thread can try and acquire the lock on the node we're deleting.

What about opportunities for concurrency? The whole point of such fine-grained locking was to improve the possibilities for concurrency over a single mutex, so have we achieved that? Yes, we have: different threads can be working on different nodes in the list at the same time, whether they are just processing each item with **for_each()**, searching with **find_first_if()** or removing items with **remove_if()**. However, since the mutex for each node must be locked in turn, the threads cannot pass each other: if one thread is spending a long time processing a particular node then other threads will have to wait when they reach that particular node.

6.4 Summary

This chapter started by looking at what it meant to design a data structure for concurrency, and some guidelines for doing so. We've then worked through several common data structures (stack, queue, hash map and linked list) looking at how to apply those guidelines to implement them in a way designed for concurrent access, using locks to protect the data and prevent data races. You should now be able to look at the design of your own data structures to see where the opportunities for concurrency lie, and where there is potential for race conditions.

In chapter 7 we'll look at ways of avoiding locks entirely, using the low-level atomic operations to provide the necessary ordering constraints, whilst sticking to the same set of guidelines.

7

Designing Data Structures for Concurrency II: Lock-free Concurrent Data Structures

In the last chapter we looked at general aspects of designing data structures for concurrency, with guidelines for thinking about the design to ensure they are safe. We then examined several common data structures and looked at example implementations that used mutexes and locks to protect the shared data. The first couple of examples used one mutex to protect the entire data structure, but later ones used more than one to protect various smaller parts of the data structure and allow greater levels of concurrency in accesses to the data structure.

Mutexes are a powerful mechanism for ensuring that multiple threads can safely access a data structure without encountering race conditions or broken invariants. It is also relatively straightforward to reason about the behaviour of code that uses them: either the code has the lock on the mutex protecting the data or it doesn't. However, it's not all a bed of roses: we saw in chapter 3 how incorrect use of locks can lead to deadlock, and we've just seen with the lock-based queue and lookup table examples how the granularity of locking can affect the potential for true concurrency. If we can write data structures that are safe for concurrent access without locks then there is the potential to avoid these problems. Such a data structure is called a *lock-free* data structure.

In this chapter we'll look at how the memory ordering properties of the atomic operations introduced in chapter 5 can be used to build lock-free data structures. Extreme care needs to be taken when designing such data structures, as they are very hard to get right, and the conditions that cause the design to fail may only occur very rarely. We'll start by looking at the reasons for using them before working through some examples and drawing out some general guidelines.

7.1 The Pros and Cons of Lock-free Data Structures

Given how hard it is to get a lock-free data structure right, you need some pretty good reasons to write one. When it comes down to it, the primary reason is to enable maximum concurrency. With lock-based containers, there is always the potential for one thread to have to block, and wait for another to complete it's operation before the first thread can proceed; with a suitably-designed lock-free data structure every thread can make forward progress, regardless what the other threads are doing.

The flip side here is that if we cannot exclude threads from accessing the data structure we must be careful to ensure that the invariants are upheld, or choose alternative invariants that can be upheld. Also, we must pay attention to the ordering constraints we impose on the operations. To avoid the undefined behaviour associated with a *data race* we must use atomic operations for the modifications, but that alone is not enough: we must ensure that changes become visible to other threads in the correct order. All this means that writing thread-safe data structures without using locks is considerably harder than writing them with locks.

Since there aren't any locks, deadlocks are simply impossible with lock-free data structures, though there is the possibility of *live-locks* instead. A *live-lock* occurs when two threads each try and change the data structure, but for each thread the changes made by the other require the operation to be restarted, so both threads loop and try again. Imagine two people trying to go through a narrow gap. If they both go at once, then they get stuck, so they have to come out and try again. Unless someone gets there first (either by agreement, by being quicker or by sheer luck) the cycle will repeat. As in this simple example, live-locks are typically short lived because they depend on the exact scheduling of threads. They therefore sap performance rather than causing long-term problems, but they are still something to watch out for.

Good Lock-free code is non-blocking and allows every thread to make progress, regardless what other thread are doing. This can increase the potential concurrency of operations on a data structure and reduce the time an individual thread spends waiting. However, this may well *decrease* overall performance since the atomic operations used for lock-free code can be much slower than non-atomic operations. Not only that, but the hardware must synchronize data between threads that access the same atomic variables. As with everything, it's important to check the relevant performance aspects (whether that's worst-case wait time, average wait-time, overall execution time or something else) both with a lock-based data structure and a lock-free one before committing either way.

Now let's look at some examples.

7.2 Worked Examples of Lock-free Data Structures

In order to demonstrate some of the techniques used in designing lock-free data structures, we'll look at the lock-free implementation of a series of simple data structures. Not only will each example describe the implementation of a useful data structure, but I will use the examples to highlight particular aspects of lock-free data structure design.

As already mentioned, lock-free data structures rely on the use of atomic operations and the associated memory ordering guarantees in order to ensure that data becomes visible to other threads in the correct order. Initially, we'll use the default **`memory_order_seq_cst`** memory ordering for all atomic operations, as that is the easiest to reason about (remember that all **`memory_order_seq_cst`** operations form a total order), but for the later examples we'll look at reducing some of the ordering constraints to **`memory_order_acquire`**, **`memory_order_release`**, or even **`memory_order_relaxed`**. Though none of these examples will use mutex locks directly, it's worth bearing in mind that only **`std::atomic_flag`** is guaranteed not to use locks in the implementation, so on some platforms what appears to be lock-free code might actually be using locks internal to the C++ Standard Library implementation. On these platforms, a simple lock-based data structure might actually be more appropriate, but there's more to it than that: before choosing an implementation, identify your requirements, and profile the various options that meet those requirements.

So, back to the beginning with the simplest of data structures: a stack.

7.2.1 Writing a Thread-safe Stack without Locks

The basic premise of a stack is relatively simple: nodes are retrieved in the reverse order to which they were added: Last In, First Out (LIFO). It is therefore important to ensure that once a value is added to the stack it can safely be retrieved *immediately* by another thread, and it is also important to ensure that only one thread returns a given value. The simplest stack is just a linked list: the **head** pointer identifies the first node (which will be the next to retrieve), and each node then points to the next node in turn.

Under such a scheme, adding a node is relatively simple:

9. Create a new node,
10. set its **`next`** pointer to the current **head** node, and
11. set the **head** node to point to it.

This works fine in a single-threaded context, but if other threads are also modifying the stack it's not enough. Crucially, if two threads are adding nodes there is a race condition between steps 2 and 3: a second thread could modify the value of **head** between when our thread reads it in step 2, and we update it in step 3. This would then result in the changes made by that other thread being discarded, or even worse consequences. Before we look at addressing this race condition, it is also important to note that once **head** has been updated to point to our new node, another thread could read that node. It is therefore vital that our new node is thoroughly prepared *before* **head** is set to point to it: we cannot modify the node afterwards.

OK, so what can we do about this nasty race condition? The answer is to use an atomic compare/exchange operation at step 3 to ensure that **head** hasn't been modified since we read it in step 2. If it has, then we can loop and try again. Listing 7.1 shows how we can implement a thread-safe **push()** without locks.

Listing 7.36: Implementing push() without locks

```
template<typename T>
class stack
{
private:
    struct node
    {
        T data;
        node* next;

        node(T const& data_):                                     #4
            data(data_)
        {}
    };

    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data);                      #1
        new_node->next=head.load();                               #2
        while(!head.compare_exchange_weak(new_node->next,new_node)); #3
    }
};
```

Cueballs in code and text

This code neatly matches our 3-point-plan from above: create a new node (#1), set the node's **next** pointer to the current **head** (#2), and set the **head** pointer to our new node (#3). By populating the data in the **node** structure itself from the **node** constructor (#4) we've ensured that the node is ready to roll as soon as it's constructed, so that's the easy problem down. Then, we use **compare_exchange_weak()** to ensure that the **head** pointer still has the same value as we stored in **new_node->next** (#3), and set it to **new_node** if so. This bit of code also uses a nifty part of the compare/exchange functionality: if it returns **false** to indicate that the comparison failed (e.g. because **head** was modified by another thread), then the value supplied as the first parameter (**new_node->next**) is *updated* to the current value of **head**. We therefore don't have to re-load **head** each time through the loop, as the compiler does that for us. Also, because we're just looping directly on failure, we can use **compare_exchange_weak**, which can result in more optimal code on some architectures.

So, we might not have a **pop()** operation yet, but we can quickly check **push()** against our guidelines. The only place that can throw an exception is the construction of the new **node** (#1), but this will clean up after itself, and the list hasn't been modified yet, so that's perfectly safe. Since we build the data to be stored as part of the **node**, and we use **compare_exchange_weak()** to update the **head** pointer, there are no problematic race conditions here. Once the compare/exchange succeeds, the node is on the list, and ready for the taking. Since there's no locks, there's no possibility of deadlock, and our **push()**function passes with flying colours.

Of course, now we've got a means of adding data to the stack, we need a way of getting them off again. On the face of it, this is quite simple:

12. Read the current value of **head**;
13. Read **head->next**;
14. Set **head** to **head->next**;
15. Return the **data** from the retrieved **node**;
16. Delete the retrieved node.

However in the presence of multiple threads this is not so simple. If there are two threads removing items from the stack, then they both might read the same value of **head** at step 1. If one thread then proceeds all the way through to step 5 before the other gets to step 2, then the second thread will be dereferencing a dangling pointer. This is one of the biggest issues in writing lock-free code, so for now we'll just leave out step 5 and leak the nodes.

This doesn't resolve all the problems though, because there's another problem: if two threads read the same value of **head** then they will return the same node. This violates the intent of the stack data structure, so we need to avoid this. We can resolve this the same way we resolved the race in **push()**: use compare/exchange to update **head**. If the compare/exchange fails then either a new node has been pushed on, or another thread just popped the node we were trying to pop. Either way we need to return to step 1 (though the compare/exchange call re-reads **head** for us).

Once the compare/exchange call succeeds, we know we are the *only* thread that is popping the given node off the stack, so we can safely execute step 4. Here's a first cut at **pop()**:

```
template<typename T>
class stack
{
public:
    void pop(T& result)
    {
        node* old_head=head.load();
        while(!head.compare_exchange_weak(old_head,old_head->next));
        result=old_head->data;
    }
};
```

Though this is nice and succinct, there are *still* a couple of problems aside from the leaking node. Firstly, it doesn't work on an empty list: if **head** is **NULL** then it will cause undefined behaviour as it tries to read the **next** pointer. This is easily fixed by checking for **NULL** in the **while** loop, and either throwing an exception on an empty stack or returning a **bool** to indicate success or failure.

The second problem is an exception-safety issue. When we first introduced the thread-safe stack back in chapter 3, we saw how just returning the object by value left us with an exception safety issue: if an exception is thrown copying the return value then the value is lost. In that case, passing in a reference to the result was an acceptable solution since we could ensure that the stack was left unchanged if an exception got thrown. Unfortunately, here we don't have that luxury: we can only safely copy the data once we know we're the only thread returning the node, *which means the node has already been removed from the queue*. Consequently, passing in the target for the return value by reference is no longer an advantage: we might as well just return by value. If we want to return the value safely we have to use the other option from chapter 3: return a (smart) pointer to the data value.

If we return a smart pointer, we can just return **NULL** to indicate that there is no value to return, however this requires that the data is allocated on the heap. If we do the heap allocation as part of the **pop()** then we're *still* no better off, as the heap allocation might throw an exception. Instead, we can allocate the memory when we **push()** the data onto the stack — we have to allocate memory for the **node** anyway. Returning a **std::shared_ptr<>** will not throw an exception, so **pop()** is now safe. Putting all this together gives us listing 7.2.

Listing 7.37: A lock-free stack that leaks nodes

```
template<typename T>
class stack
{
```

```
private:
    struct node
    {
        std::shared_ptr<T> data;                                   #1
        node* next;

        node(T const& data_):
            data(new T(data_))                                     #2
        {}
    };

    std::atomic<node*> head;
public:
    void push(T const& data)
    {
        node* const new_node=new node(data);
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=head.load();
        while(old_head &&                                            #3
            !head.compare_exchange_weak(old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();     #4
    }
};
```

Cueballs in code and text

The data is held by pointer now (#1), so we have to allocate the data on the heap in the node constructor (#2). We've also got a check for **NULL** before we dereference **old_head** in the **compare_exchange_weak()** loop. Finally, we either return the data associated with our node if there is one, or a **NULL** pointer if not.

If you've got a garbage collector picking up after you (like in managed languages such as C# or Java), we're done: the old node will be collected and recycled once it is no longer being accessed by any threads. However, not many C++ compilers ship with a garbage collector so we generally have to tidy up after ourselves.

7.2.2 Stopping those pesky leaks: managing memory in lock-free data structures

When we first looked at **pop()**, we opted to leak nodes in order to avoid the race condition where one thread deletes a node whilst another thread still holds a pointer to it that it is just about to dereference. However, leaking memory is not acceptable in any sensible C++

program, so we have to do something about that. Now it's time to look at the problem, and work out a solution.

The basic problem is that we want to free a node, but we can't do so until we're sure there are no other threads that still hold pointers to it. This essentially means we need to write a special-purpose garbage collector just for **node**s. Now, this might sound scary, but whilst it's certainly tricky it's not *that* bad: we're only checking for **node**s, and we're only checking for nodes accessed from **pop()**. We're not worried about nodes in **push()**, because they're only accessible from one thread until they're on the stack, whereas multiple threads might be accessing the same node in **pop()**.

If there are no threads calling **pop()**, then it's perfectly safe to delete all the nodes currently awaiting deletion. Therefore, if we add the nodes to a “to be deleted” list when we've extracted the data, then we can delete them all when there are no threads calling **pop()**. How do we know there aren't any threads calling **pop()**? Simple — count them. If we increment a counter on entry, and decrement that counter on exit then it is safe to delete the nodes from the “to be deleted” list when the counter is zero. Of course, it will have to be an atomic counter so it can safely be accessed from multiple threads. Listing 7.3 shows such an implementation.

Listing 7.38: Reclaiming nodes when no threads are in pop()

```
template<typename T>
class stack
{
private:
    std::atomic<unsigned> threads_in_pop;                            #1
    std::atomic<node*> to_be_deleted;

    static void delete_nodes(node* nodes)                            #7
    {
        while(nodes)
        {
            node* next=nodes->next;
            delete nodes;
            nodes=next;
         }
    }
public:
    std::shared_ptr<T> pop()
    {
        ++threads_in_pop;                                            #2
        node* old_head=head.load();
        while(old_head &&
            !head.compare_exchange_weak(old_head,old_head->next));
        std::shared_ptr<T> res;
```

```
            if(old_head)
            {
                res.swap(old_head->data);
            }
            node* nodes_to_delete=to_be_deleted.load();                  #5
            if(!--threads_in_pop)                                        #3
            {
                if(to_be_deleted.compare_exchange_strong(                #6
                    nodes_to_delete,NULL))
                {
                    delete_nodes(nodes_to_delete);
                }
                delete old_head;                                         #4
            }
            else if(old_head)                                            #8
            {
                old_head->next=nodes_to_delete;                          #9
                while(!to_be_deleted.compare_exchange_weak(              #10
                    old_head->next,old_head));
            }
            return res;
        }
    };
```

The atomic variable **threads_in_pop** (#1) is used to count the threads currently trying to pop an item off the stack. It is incremented at the start of **pop()** (#2) and decremented once the node has been removed (#3). If the count is decremented to zero, then it is safe to delete the node we just removed (#4), because no other thread can possibly be referencing it. It *may* also be safe to delete the other pending nodes, but only if no other nodes have been added to the list — if other nodes have been added to the list, that means another thread has called **pop()** since we decreased **threads_in_pop** to zero, so there may now still be threads in **pop()**. In order to check for this, we load the current head of the list of nodes that are waiting to be deleted *before* we decrease the count of **threads_in_pop** (#5), and can then use **compare_exchange_strong()** to atomically check whether it's still the node at the head of the **to_be_deleted** list, and set the list pointer to **NULL** if so (#6). We must use **compare_exchange_strong()** here, since **compare_exchange_weak()** may fail even if the pointer is unchanged.

If the **compare_exchange_strong()** succeeds, no other threads have added any nodes to the pending list, so it is safe to delete the whole list. We accomplish this by simply following the **next** pointers and deleting the nodes as we go (#7).

If we *weren't* the last thread in **pop()** when we decremented the count (#3), then we need to add the node we removed (if any (#8)) to the **to_be_deleted** list. We do this by reusing the **next** field of the node to point to the current head of the list (#9), and looping on a **compare_exchange_weak()** call until it succeeds (#10). Finally, we can return the data we extracted from the node.

This works reasonably well in low-load situations, where there are suitable *quiescent* points at which no threads are in **pop()**. However, this is potentially a transient situation, which is why we need to test that the **to_be_deleted** list is unchanged (#6), and why this test occurs *before* we delete the just-removed node (#4) — deleting a node is potentially a time-consuming operation, and we want the window in which other threads can modify the list to be as small as possible.

In high-load situations, there may *never* be such a quiescent state, since other threads have entered **pop()** before all the threads initially in **pop()** have left. Under such a scenario, the **to_be_deleted** list would grow without bounds, and we'd be essentially leaking memory again. If there's not going to be any quiescent periods, we need to find an alternative mechanism for reclaiming the nodes. The key is to identify when no more threads are accessing a particular node in order that it can be reclaimed. By far the easiest such mechanism to reason about is the use of *hazard pointers*.

7.2.3 Detecting nodes that can't be reclaimed using hazard pointers

The term *Hazard Pointers* is a reference to a technique discovered by Maged Michael [Michael2002]. They are so-called because deleting a node that might still be referenced by other threads is *hazardous* — if other threads do indeed hold references to that node and proceed to access the node through that reference then you've got undefined behaviour. The basic idea is that if a thread is going to access an object that another thread might want to delete then it first sets a *hazard pointer* to reference the object, thus informing the other thread that deleting the object would indeed be hazardous. Once the object is no longer needed, the hazard pointer is cleared. If you've ever watched the Oxford/Cambridge boat race, they use a similar mechanism when starting the race: the cox of either boat can raise their hand to indicate that they are not ready. Whilst either cox has their hand raised the umpire may not start the race. If both coxes have their hands down the race may start, but a cox may raise their hand again if the race has not started and they feel the situation has changed.

When a thread wishes to delete an object it must first check the hazard pointers belonging to the other threads in the system. If none of the hazard pointers reference the object it can safely be deleted. Otherwise it must be left until later. Periodically, the list of objects that have been “left until later” is checked to see if any of them can now be deleted.

Described at such a high level it sounds relatively straightforward, so how do we do this in C++?

Well, first off we need a location in which to store the pointer to the object we're accessing, the *hazard pointer* itself. This location must be visible to all threads, and we need one of these for each thread that might access the data structure. Allocating them correctly and efficiently can be a challenge, so we'll leave that for later and assume we've got a function **get_hazard_pointer_for_current_thread()** that returns a reference to our hazard pointer. We then need to set it when we read a pointer that we intend to dereference — in this case the **head** value from the list:

```
std::shared_ptr<T> pop()
{
    std::atomic_address& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();                                       #1
    node* temp;
    do
    {
        temp=old_head;
        hp.store(old_head);                                           #2
        old_head=head.load();
    } while(old_head!=temp);                                          #3
    // ...
}
```

Cueballs in code and text

We have to do this in a **while** loop to ensure that the **node** hasn't been deleted between the reading of the old **head** pointer (#1) and the setting of the hazard pointer (#2). During this window no other thread knows we're accessing this particular node. Fortunately, if the old **head** node *is* going to be deleted then **head** itself must have changed, so we can check this and keep looping until we know that the **head** pointer still has the same value we set our hazard pointer to (#3).

Now we've got our hazard pointer set, we can proceed with the rest of **pop()**, safe in the knowledge that no other thread will delete the nodes from under us. Well, almost: every time we reload **old_head**, we need to update the hazard pointer before we dereference it the pointer. Once we've extracted a node from the list, we can then clear our hazard pointer. If there are no other hazard pointers referencing our node, we can safely delete it, otherwise we have to add it to a list of nodes to be deleted later. Listing 7.4 shows a full implementation of **pop()** using such a scheme.

Listing 7.39: An implementation of pop() using hazard pointers

```
std::shared_ptr<T> pop()
{
    std::atomic_address& hp=get_hazard_pointer_for_current_thread();
    node* old_head=head.load();
    do
    {
        node* temp;
        do                                                             #1
```

```
            {
                temp=old_head;
                hp.store(old_head);
                old_head=head.load();
            } while(old_head!=temp);
        }
        while(old_head &&
              !head.compare_exchange_strong(old_head,old_head->next));
        hp.store(NULL);                                                   #2
        std::shared_ptr<T> res;
        if(old_head)
        {
            res.swap(old_head->data);
            if(outstanding_hazard_pointers_for(old_head))                #3
            {
                reclaim_later(old_head);                                  #4
            }
            else
            {
                delete old_head;                                          #5
            }
            delete_nodes_with_no_hazards();                               #6
        }
        return res;
    }
```

Cueballs in code and text

First off, we've moved the loop that sets the hazard pointer *inside* the outer loop for reloading **old_head** if the compare/exchange fails (#1). We're using **compare_exchange_strong()** here since we're actually doing work inside the while loop: a spurious failure on **compare_exchange_weak()** would result in resetting the hazard pointer unnecessarily. This ensures that the hazard pointer is correctly set before we dereference **old_head**. Once we've claimed the node as ours, we can clear our hazard pointer (#2). If we did get a node, then we need to check the hazard pointers belonging to other threads to see if they reference it (#3). If so, we can't delete it just yet, so must put it on a list to be reclaimed later (#4), otherwise we can delete it right away (#5). Finally, we put in a call to check for any nodes for which we had to call **reclaim_later()** — if there are no longer any hazard pointers referencing those nodes, we can safely delete them now (#6). Any nodes for which there are still outstanding hazard pointers will be left for the next thread that calls **pop()**.

Of course, there's still a lot of detail hidden in these new functions: **get_hazard_pointer_for_current_thread()**, **reclaim_later()**,

outstanding_hazard_pointers_for(), and **delete_nodes_with_no_hazards()**, so let's draw back the curtain and look at how they work.

The exact scheme for allocating hazard pointer instances to threads used by **get_hazard_pointer_for_current_thread()** doesn't really matter for the program logic (though it can affect the efficiency, as we'll see later), so for now we'll go with a really simple structure: a fixed-size array of pairs of thread IDs and pointers. **get_hazard_pointer_for_current_thread()** then searches through the array to find the first free slot, and sets the ID entry of that slot to the ID of the current thread. When the thread exits, the slot is freed by resetting the ID entry to a default-constructed **std::thread::id()**. This is shown in listing 7.5.

Listing 7.40: A simple implementation of get_hazard_pointer_for_current_thread()

```
unsigned const max_hazard_pointers=100;
struct hazard_pointer
{
    std::atomic<std::thread::id> id;
    std::atomic_address pointer;
};
hazard_pointer hazard_pointers[max_hazard_pointers];

class hp_owner
{
    hazard_pointer* hp;

public:
    hp_owner(hp_owner const&)=delete;
    hp_owner operator=(hp_owner const&)=delete;

    hp_owner():                                                    #4
        hp(NULL)
    {
        for(unsigned i=0;i<max_hazard_pointers;++i)
        {
            std::thread::id old_id;
            if(hazard_pointers[i].id.compare_exchange_strong(          #5
                old_id,std::this_thread::get_id()))
            {
                hp=&hazard_pointers[i];                                #6
                break;
            }
        }
        if(!hp)                                                        #7
        {
            throw std::runtime_error("No hazard pointers available");
        }
    }

    std::atomic_address& get_pointer()
    {
        return hp->pointer;
    }

    ~hp_owner()                                                         #8
    {
        hp->pointer.store(NULL);
        hp->id.store(std::thread::id());
    }
};

std::atomic_address& get_hazard_pointer_for_current_thread()           #1
{
    thread_local static hp_owner hazard;                                #2
    return hazard.get_pointer();                                        #3
}
```

Cueballs in code and text

The actual implementation of **get_hazard_pointer_for_current_thread()** itself is deceptively simple (#1): it has a **thread_local** variable of type **hp_owner** (#2) which stores the hazard pointer for the current thread. It then just returns the pointer from that object (#3). This works as follows: the first time *each thread* calls this function, a new instance of **hp_owner** is created. The constructor for this new instance (#4) then searches through the table of owner/pointer pairs looking for an entry without an owner. It uses **compare_exchange_strong()** to check for an entry without an owner and claim it in one go (#5). If the **compare_exchange_strong()** fails, then another thread owns that entry, so we move on to the next. If the exchange succeeds, we've successfully claimed the entry for the current thread, so we store it and stop the search (#6). If we get to the end of the list without finding a free entry (#7), then there are too many threads using hazard pointers, so we throw an exception.

When each thread exits, then if an instance of **hp_owner** was created for that thread then it is destroyed: the destructor then resets the actual pointer to **NULL**, before setting the owner ID to **std::thread::id()**, allowing another thread to reuse the entry later (#8).

With this implementation of **get_hazard_pointer_for_current_thread()**, the implementation of **outstanding_hazard_pointers_for()** is really simple: just scan through the hazard pointer table looking for entries.

```
bool outstanding_hazard_pointers_for(void* p)
{
    for(unsigned i=0;i<max_hazard_pointers;++i)
    {
        if(hazard_pointers[i].pointer.load()==p)
        {
            return true;
        }
    }
    return false;
}
```

It's not even worth checking whether or not each entry has an owner: unowned entries will have **NULL** pointers, so the comparison will return **false** anyway, and it simplifies the code.

reclaim_later() and **delete_nodes_with_no_hazards()** can then just work on a simple linked list: **reclaim_later()** just adds nodes to the list, and **delete_nodes_with_no_hazards()** scans through the list, deleting entries with no outstanding hazards. Listing 7.6 shows just such an implementation.

Listing 7.41: A simple implementation of reclaim_later() and delete_nodes_with_no_hazards()

```
    template<typename T>
    void do_delete(void* p)
    {
        delete static_cast<T*>(p);
    }

    struct data_to_reclaim
    {
        void* data;
        std::function<void(void*)> deleter;
        data_to_reclaim* next;

        template<typename T>
        data_to_reclaim(T* p):                                            #4
            data(p),
            deleter(&do_delete<T>),
            next(0)
        {}

        ~data_to_reclaim()
        {
            deleter(data);                                                #5
        }
    };

    std::atomic<data_to_reclaim*> nodes_to_reclaim;

    void add_to_reclaim_list(data_to_reclaim* node)                       #3
    {
        node->next=nodes_to_reclaim.load();
        while(!nodes_to_reclaim.compare_exchange_weak(node->next,node));
```

```
}

template<typename T>
void reclaim_later(T* data)                                             #1
{
    add_to_reclaim_list(new data_to_reclaim(data));                     #2
}

void delete_nodes_with_no_hazards()
{
    data_to_reclaim* current=nodes_to_reclaim.exchange(NULL);           #6
    while(current)
    {
        data_to_reclaim* const next=current->next;
        if(!outstanding_hazard_pointers_for(current->data))             #7
        {
            delete current;                                             #8
        }
        else
        {
            add_to_reclaim_list(current);                               #9
        }
        current=next;
    }
}
```

Cueballs in code and text

First off, I expect you've spotted that **reclaim_later()** is a function template rather than a plain function (#1). This is because hazard pointers are a general purpose utility, so we don't want to tie ourselves to stack nodes: we've been using **std::atomic_address** for storing the pointers already. We therefore need to handle any pointer type, but we can't use **void*** because we want to delete the data items when we can, and **delete** requires the real type of the pointer. The constructor of **data_to_reclaim** handles that nicely as we'll see in a minute: **reclaim_later()** just creates a new instance of **data_to_reclaim** for our pointer, and adds it to the reclaim list (#2). **add_to_reclaim_list()** itself (#3) is just a simple **compare_exchange_weak()** loop on the list head like we've seen before.

So, back to the constructor of **data_to_reclaim** (#4): the constructor is also a template. It stores the data to be deleted as a **void*** in the **data** member, and then stores a pointer to the appropriate instantiation of **do_delete()** — a simple function that casts the supplied **void*** to the chosen pointer type and then deletes the pointed-to object. **std::function<>** wraps this function pointer safely, so that the destructor of **data_to_reclaim** can then delete the data just by invoking the stored function (#5).

Of course, the destructor of **data_to_reclaim** isn't called when we're adding nodes to the list: it's called when there are no more hazard pointers to that node. This is the responsibility of **delete_nodes_with_no_hazards()**.

delete_nodes_with_no_hazards() firstly claims the entire list of nodes to be reclaimed for itself with a simple **exchange()** (#6) . This simple, but crucial, step ensures that this is the only thread trying to reclaim this particular set of nodes. Other threads are now free to add further nodes to the list or even try and reclaim them without impacting the operation of this thread.

Then, as long as there's still nodes left in the list, we check each node in turn to see if there's any outstanding hazard pointers (#7). If there aren't then we can safely delete the entry (and thus clean up the stored data) (#8), otherwise we just add the item back on the list for reclaiming later (#9).

Though this simple implementation does indeed safely reclaim the deleted nodes, it adds quite a bit overhead to the process. Scanning the hazard pointer array requires checking **max_hazard_pointers** atomic variables, and this is done for every **pop()** call. Atomic operations are inherently slow — typically 100 times slower than an equivalent non-atomic operation on desktop CPUs — so this makes **pop()** a *very* expensive operation. In fact, not only do we scan the hazard pointer list for the node we are about to remove, but we also scan it for each node in the waiting list. Clearly this is a very bad idea — there may well be **max_hazard_pointers** nodes in the list, and we're checking all of them against **max_hazard_pointers** stored hazard pointers. Ouch. There has to be a better way.

Better reclamation strategies using Hazard Pointers

Of course, there *is* a better way — what I've shown here is a very simple and naïve implementation of hazard pointers to help explain the technique. The first thing we can do is trade memory for performance — rather than checking every node on the reclamation list *every* time we call **pop()**, don't try and reclaim any nodes at all unless there's more than **max_hazard_pointers** nodes on the list. That way we're guaranteed to be able to reclaim at least one node. If we just wait until there's **max_hazard_pointers+1** nodes on the list then we're not much better off though: once we've got to **max_hazard_pointers** nodes, we'll be trying to reclaim nodes for most calls to **pop()**, so we're not doing much better. However, if we leave it until there are **2*max_hazard_pointers** nodes on the list, we're guaranteed to be able to reclaim at least **max_hazard_pointers** nodes, and it will then be at least **max_hazard_pointers** calls to **pop()** before we try and reclaim any nodes again. This is *much* better: rather than checking around **max_hazard_pointers** nodes every call to **push()** (and not necessarily reclaiming any), we're checking **2*max_hazard_pointers** nodes every **max_hazard_pointers** calls to **pop()**, and reclaiming at least

max_hazard_pointers nodes. That's effectively two nodes checked for every **pop()**, one of which is reclaimed.

However, even this has a downside (beyond the increased memory usage): we've now got to count the nodes on the reclamation list, which means using an atomic count, and we've still got multiple threads competing to access the reclamation list itself. If we've got memory to spare, we can trade increased memory usage for an even better reclamation scheme: each thread keeps its own reclamation list in a thread-local variable. There is thus no need for atomic variables for the count, or the list access — instead we've got **max_hazard_pointers*max_hazard_pointers** nodes allocated. If a thread exits before all its nodes have been reclaimed, they can be stored in the global list as before, and added to the local list of the next thread doing a reclamation process.

Of course, the biggest downside of Hazard pointers is that they're patented by IBM. If you write software for use in a country where the patents are valid, then you need to make sure you've got a suitable licensing arrangement in place. This is something common to many of the lock-free memory reclamation techniques: this is an active research area, so companies such as IBM and Sun are taking out patents where they can. You may well be asking why I've devoted so many pages to a technique that many people will be unable to use, and that's a fair question. Firstly, the patents in question don't apply in every country. Secondly, some of the licenses permit the techniques to be used in free software licensed under the GPL. Thirdly, and most importantly, the explanation of the techniques shows some of the things that it is important to think about when writing lock-free code.

So, are there any non-patented memory reclamation techniques that can be used with lock-free code? Luckily, there are. One such mechanism is reference counting.

7.2.4 Detecting Nodes in Use with Reference Counting

Back in 7.2.2, we saw that the problem with deleting nodes is detecting which nodes are still being accessed by reader threads. If we could safely identify precisely which nodes were being referenced, and when no threads were accessing these nodes then we could delete them. Hazard pointers tackle the problem by storing a list of the nodes in use. Reference counting tackles the problem by storing a count of the number of threads accessing each node.

This may seem nice and straightforward, but it's quite hard to manage in practice. At first, you might think that something like **std::shared_ptr<>** would be up to the task — after all, it's a reference-counted pointer. Unfortunately, though some operations on **std::shared_ptr<>** are atomic, they are not guaranteed to be lock free. Though by itself this is no different to any of the operations on the atomic types, **std::shared_ptr<>** is intended for use in many contexts, and making the atomic operations lock-free would likely impose an overhead on all uses of the class. If your platform supplies an implementation for which **std::atomic_is_lock_free(&some_shared_ptr)** returns **true** then the whole memory reclamation issue goes away: just use **std::shared_ptr<node>** for the list, as in listing 7.7.

Listing 7.42: A lock-free stack using a lock-free std::shared_ptr<> implementation

```
template<typename T>
class stack
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::shared_ptr<node> next;

        node(T const& data_):
            data(new T(data_))
        {}
    };

    std::shared_ptr<node> head;
public:
    void push(T const& data)
    {
        std::shared_ptr<node> const new_node=new node(data);
        new_node->next=head.load();
        while(!std::atomic_compare_exchange_weak(&head,
                  &new_node->next,new_node));
    }
    std::shared_ptr<T> pop()
    {
        std::shared_ptr<node> old_head=std::atomic_load(&head);
        while(old_head && !std::atomic_compare_exchange_weak(&head,
                  &old_head,old_head->next));
        return old_head ? old_head->data : std::shared_ptr<T>();
    }
};
```

Of course, in the probable case that your **std::shared_ptr<>** implementation is not lock free, then you need to manage the reference counting manually.

One possible technique involves the use of not one but *two* reference counts for each node: an internal count and an external count. The sum of these values is the total number of references to the node. The external count is kept alongside the pointer to the node, and is increased every time the pointer is read. When the reader is done with the node, it decreases the *internal* count. A simple operation that reads the pointer will thus leave the external count increased by one, and the internal count decreased by one when it is finished.

When the external count/pointer pairing is no longer required (I.e. the node is no longer accessible from a location accessible to multiple threads), then the internal count is increased by the value of the external count minus one. Once the internal count is equal to zero that means that there are no outstanding references to the node and it can be safely deleted. Of course it's still important to use atomic operations for updates of shared data. Listing 7.8 shows an implementation of a lock-free stack that uses this technique to ensure that the nodes are only reclaimed when it is safe.

Listing 7.43: A lock-free stack using split reference counts

```
template<typename T>
class stack
{
private:
    struct node;

    struct counted_node_ptr                                               #1
    {
        int external_count;
        node* ptr;
    };

    struct node
    {
        std::shared_ptr<T> data;
        std::atomic_int internal_count;                                   #3
        counted_node_ptr next;                                            #2

        node(T const& data_):
            data(new T(data_)),
            internal_count(0)
        {}
    };

    std::atomic<counted_node_ptr> head;                                   #4

    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter)); #7
```

```
            old_counter.external_count=new_counter.external_count;
        }

    public:
        ~stack()
        {
            while(pop());
        }

        void push(T const& data)                                          #5
        {
            counted_node_ptr new_node;
            new_node.ptr=new node(data);
            new_node.external_count=1;
            new_node.ptr->next=head.load();
            while(!head.compare_exchange_weak(new_node.ptr->next,new_node));
        }

        std::shared_ptr<T> pop()                                          #6
        {
            counted_node_ptr old_head=head.load();
            for(;;)
            {
                increase_head_count(old_head);
                node* const ptr=old_head.ptr;                             #8
                if(!ptr)
                {
                    return std::shared_ptr<T>();
                }
                if(head.compare_exchange_strong(old_head,ptr->next))      #9
                {
                    std::shared_ptr<T> res;
                    res.swap(ptr->data);                                  #10

                    int const count_increase=old_head.external_count-2;   #12

                    if(ptr->internal_count.fetch_add(count_increase)==    #11
                       -count_increase)
                    {
                        delete ptr;
                    }

                    return res;                                           #13
                }
                else if(ptr->internal_count.fetch_add(-1)==1)
                {
                    delete ptr;                                           #14
                }
            }
        }
```

```
};
```

Cueballs in code and text

First off, the external count is wrapped together with the node pointer in the **counted_node_ptr** structure (#1). This can then be used for the **next** pointer in the **node** structure (#2) alongside the internal count (#3). Because **counted_node_ptr** is just a simple **struct**, we can use it with the **std::atomic<>** template for the **head** of the list (#4). On many platforms this structure will be small enough for **std::atomic<counted_node_ptr>** to be lock-free: if it isn't on your platform you might be better off using the **std::shared_ptr<>** version from listing 7.7.

push() is relatively simple (#5). We construct a **counted_node_ptr** that refers to a freshly-allocated **node** with associated data, and set the **next** value of the **node** to the current value of **head**. We can then use **compare_exchange_weak()** to set the value of **head**, just as in the previous listings. The counts are set up so the **internal_count** is zero, and the **external_count** is one: there is currently one external reference to the node (the **head** pointer itself).

As ever, all the complication is in **pop()** (#6). This time, once we've loaded the value of **head** we must first increase the count of external references to the **head** node to indicate that we're referencing it, and ensure it is safe to dereference it. We do this with a **compare_exchange_strong()** loop (#7). Once the count has been increased, we can safely dereference the **ptr** field of the value loaded from **head** in order to access the pointed-to node (#8). If the pointer is **NULL** then we're at the end of the list: no more entries. If the pointer is not **NULL** then we can try and remove the node by a **compare_exchange_strong()** call on **head** (#9).

If the **compare_exchange_strong()** succeeds, then we've taken ownership of the node, and can swap the **data** out in preparation for returning it (#10). This ensure that the data isn't kept alive just because other threads accessing the stack happen to still have pointers to its node. Then we can add the external count to the internal count on the node with an atomic **fetch_add** (#11). If the reference count is now zero then the *previous* value (which is what **fetch_add** returns) was the negative of what we just added, in which case we can delete the node. It is important to note that the value we add is actually *two less* than the external count (#12) — we've removed the node from the list, so we drop one off the count for that, and we're no longer accessing the node from this thread so we drop another off the count for that. Whether or not we deleted the node, we're done, so we can return the data (#13).

If the compare/exchange (#9) *fails* then another thread removed our node before we did, or another thread added a new node to the stack. Either way, we need to start again with the fresh value of **head** returned by the compare/exchange call. However, first we must decrease the reference count on the node we were trying to remove — this thread isn't going to access it any more. If we're the last thread to hold a reference (because another thread removed it from the stack) then the internal reference count will be 1, so adding -1 will set the count to zero. In this case, we can delete the node here before we loop (#14).

So far, we've been using the default **std::memory_order_seq_cst** memory ordering for all our atomic operations. On most systems these are more expensive in terms of execution time and synchronization overhead than the other memory orderings, and on some systems considerably so. Now we've got the logic of our data structure right, we can think about relaxing some of these memory ordering requirements: we don't want to impose any unnecessary overhead on the users of our stack. So, before we leave our stack behind and move onto the design of a lock-free queue, let's examine the stack operations and ask ourselves: can we use more relaxed memory orderings for some operations and still get the same level of safety?

7.2.5 Applying the memory model to our lock-free stack

Before we go about changing the memory orderings, we need to examine the operations, and identify the required relationships between them. We can then go back and find the minimum memory orderings that provide these required relationships. In order to do this, we'll have to look at the situation from the point of view of threads in several different scenarios. The simplest possible scenario has to be where one thread pushes a data item onto the stack, and another thread then pops that data item off the stack some time later, so we'll start from there.

In this simple case, there are three important pieces of data that are involved. Firstly, there is the **counted_node_ptr** used for transferring the data: **head**. Secondly, there is the **node** structure that **head** refers to, and finally there is the data item pointed to by that node.

The thread doing the **push()** first constructs the data item and the **node**, and then sets **head**. The thread doing the **pop()** first loads the value of **head**, then does a compare/exchange loop on **head** to increase the reference count, and then reads the **node** structure to obtain the **next** value. Right here we can see a required relationship: the **next** value is a plain non-atomic object, so in order to read this safely there must be a *happens-before* relationship between the store (by the pushing thread) and the load (by the popping thread). Since the only atomic operation in the **push()** is the **compare_exchange_weak()**, and you need a *release* operation to get a *happens-before* relationship between threads, the **compare_exchange_weak()** must be **std::memory_order_release** or stronger. If the **compare_exchange_weak()** call fails then nothing has changed, and we keep looping, so we only need **std::memory_order_relaxed** in that case.

```
void push(T const& data)
{
    counted_node_ptr new_node;
```

```
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load();
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
            std::memory_order_release,std::memory_order_relaxed));
    }
```

What about the **pop()** code? In order to get the *happens-before* relationship we need, then we must have an operation that is **std::memory_order_acquire** or stronger *before* the access to **next**. The pointer we dereference to access the **next** field is the old value read by the **compare_exchange_strong()** in **increase_head_count()**, so we need the ordering on that if it succeeds. As with the call in **push()**, if the exchange fails we just loop again, so we can use relaxed ordering on failure:

```
    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter,
            std::memory_order_acquire,std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }
```

If the **compare_exchange_strong()** call succeeds, then we know that the value read had the **ptr** field to what is now stored in **old_counter**. Since the store in **push()** was a release operation, and this **compare_exchange_strong()** is an acquire operation, the store *synchronizes- with* the load and we have a *happens-before* relationship. Consequently the store to the **ptr** field in the **push()** *happens-before* the **ptr->next** access in **pop()**, and we're safe.

Note that the memory ordering on the initial **head.load()** didn't matter to this analysis, so we can safely use **std::memory_order_relaxed** for that.

Next up, the **compare_exchange_strong()** to set **head** to **old_head.ptr->next**. Do we need anything from this operation to guarantee the data integrity of this thread? If the exchange succeeds, we access **ptr->data**, so we need to ensure that the store to **ptr->data** in the **push()** thread *happens-before* the load. However, we already have that guarantee: the acquire operation in **increase_head_count()** ensures that there's a *synchronizes-with* relationship between the store in the **push()** thread and that compare/exchange. Since the store to **data** in the **push()** thread is *sequenced-before* the store to **head** and the call to **increase_head_count()** is *sequenced-before* the load of **ptr->data** there is a *happens-before* relationship and all is well even if this compare/exchange in

pop() uses **std::memory_order_relaxed**. The only other place that **ptr->data** is changed is the very call to **swap()** that we're looking at, and no other thread can be operating on the same node: that's the whole point of the compare/exchange.

If the **compare_exchange_strong()** fails, the new value of **old_head** isn't touched until next time round the loop, and we already decided that the **std::memory_order_acquire** in **increase_head_count()** was enough, so **std::memory_order_relaxed** is enough there also.

What about other threads? Do we need anything stronger here to ensure other threads are still safe? The answer is *no*, because **head** is only ever modified by compare/exchange operations. Since these are *read-modify-write* operations they form part of the *release sequence* headed by the compare/exchange in **push()**. Therefore, the **compare_exchange_weak()** in **push()** *synchronizes-with* a call to **compare_exchange_strong()** in **increase_head_count()** that reads the value stored, *even if many other threads modify head in the mean time*.

So, we're nearly done: the only remaining things to deal with are the **fetch_add()** operations for modifying the reference count. If we're the thread that got to return the data from this node then no other thread can have modified the node data. However, if we're not the thread that got to use this node, another thread *did* modify the node data: we use **swap()** to extract the referenced data item. Therefore we need to ensure that the **swap()** *happens-before* the **delete** in order to avoid a data race. The easy way to do this is to make the **fetch_add()** in the successful-return branch use **std::memory_order_release**, and the **fetch_add()** in the loop-again branch use **std::memory_order_acquire**. However, this is still overkill: only one thread does the **delete** (the one that sets the count to zero), so only that thread needs to do an *acquire* operation. Thankfully, since **fetch_add()** is a *read-modify-write* operation we can do that with an additional **load()**: if the loop-again branch finds decreases the reference count to zero then it can reload the reference count with **std::memory_order_acquire** in order to ensure the required *synchronizes-with* relationship, and the **fetch_add()** itself can use **std::memory_order_relaxed**. The final stack implementation with the new version of **pop()** is shown in listing 7.9.

Listing 7.44: A lock-free stack with reference counting and relaxed atomic operations

```
template<typename T>
class stack
{
private:
    struct node;

    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };

    struct node
    {
        std::shared_ptr<T> data;
        std::atomic_int internal_count;
        counted_node_ptr next;

        node(T const& data_):
            data(new T(data_)),
            internal_count(0)
        {}
    };

    std::atomic<counted_node_ptr> head;

    void increase_head_count(counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!head.compare_exchange_strong(old_counter,new_counter,
                                            std::memory_order_acquire,
                                            std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }

public:
    ~stack()
    {
        while(pop());
    }

    void push(T const& data)
    {
        counted_node_ptr new_node;
        new_node.ptr=new node(data);
        new_node.external_count=1;
        new_node.ptr->next=head.load();
        while(!head.compare_exchange_weak(new_node.ptr->next,new_node,
                                          std::memory_order_release,
                                          std::memory_order_relaxed));
    }
    std::shared_ptr<T> pop()
```

```
        {
            counted_node_ptr old_head=
                head.load(std::memory_order_relaxed);
            for(;;)
            {
                increase_head_count(old_head);
                node* const ptr=old_head.ptr;
                if(!ptr)
                {
                    return std::shared_ptr<T>();
                }
                if(head.compare_exchange_strong(old_head,ptr->next,
                                                std::memory_order_relaxed))
                {
                    std::shared_ptr<T> res;
                    res.swap(ptr->data);

                    int const count_increase=old_head.external_count-2;

                    if(ptr->internal_count.fetch_add(count_increase,
                           std::memory_order_release)==-count_increase)
                    {
                        delete ptr;
                    }

                    return res;
                }
                else if(ptr->internal_count.fetch_add(-1,
                            std::memory_order_relaxed)==1)
                {
                    ptr->internal_count.load(std::memory_order_acquire);
                    delete ptr;
                }
            }
        }
    };
```

That was quite a plough, but we got there in the end, and the stack is better for it: by using more relaxed operations in a carefully thought-through manner, the performance is improved without impacting the correctness. As you can see, the implementation of **pop()** is now 37 lines rather than the 8 lines of the equivalent **pop()** in the lock-based stack of listing 6.1, and the 7 lines of the basic lock-free stack without memory management in listing 7.1. As we move on to look at writing a lock-free queue, we will see a similar pattern: lots of the complexity in lock-free code comes from managing memory.

7.2.6 Writing a thread-safe queue without locks

A queue offers a slightly different challenge to a stack, since the **push()** and **pop()** operations access different parts of the data structure in a queue, whereas they both access the same head node for a stack. Consequently, the synchronization needs are different: we need to ensure that changes made to one end are correctly visible to accesses at the other. However, the structure of **try_pop()** for the queue in listing 6.6 is not actually that far off that of **pop()** for the simple lock-free stack in listing 7.1, so we can reasonably assume that the lock-free code won't be that dissimilar. Let's see how we go.

If we take listing 6.6 as a basis, then we need two **node** pointers: one for the **head** of the list, and one for the **tail**. We're going to be accessing these from multiple threads, so they'd better be atomic in order to allow us to do away with the corresponding mutexes. Let's just start by making that small change and see where it gets us. Listing 7.10 shows the result.

Listing 7.45: A first attempt at a lock-free queue

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        node* next;

        node():
            next(NULL)
        {}
    };

    std::atomic<node*> head;
    std::atomic<node*> tail;

    node* pop_head()
    {
        node* const old_head=head.load();
        if(old_head==tail.load())                                    #2
        {
            return NULL;
        }
        head.store(old_head->next);
        return old_head;
    }
public:
    queue():
        head(new node),tail(head.load())
    {}

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    ~queue()
    {
        while(node* const old_head=head.load())
        {
            head.store(old_head->next);
            delete old_head;
        }
    }
    std::shared_ptr<T> pop()
    {
        node* old_head=pop_head();
        if(!old_head)
        {
            return std::shared_ptr<T>();
        }

        std::shared_ptr<T> const res(old_head->data);                #4
        delete old_head;
        return res;
    }

    void push(T new_value)
    {
        std::shared_ptr<T> new_data(new T(new_value));
        node* p=new node;                                            #5
        node* const old_tail=tail.load()                             #6
        old_tail->data.swap(new_data);                               #3
        old_tail->next=p;                                            #7
        tail.store(p);                                               #1
    }
};
```

Cueballs in code and text

At first glance, this doesn't seem too bad, and if there's only one thread calling **push()** at a time, and only one thread calling **pop()** then this is actually perfectly fine: the important thing in that case is the *happens-before* relationship between the **push()** and the **pop()** to ensure that it is safe to retrieve the **data**. The store to **tail** (#1) *synchronizes-with* the load from **tail** (#2), the store to the preceding node's **data** pointer (#3) is *sequenced-before* the store to **tail**, and the load from **tail** is *sequenced-before* the load from the **data** pointer (#4), so the store to **data** *happens-before* the load, and everything is OK. This is therefore a perfectly serviceable *single-producer, single-consumer queue*.

The problems come when multiple threads call **push()** concurrently, or multiple threads call **pop()** concurrently. Let's look at **push()** first. If we have two threads calling **push()** concurrently, then they will both allocate new nodes to be the new dummy node (#5), both read the *same* value for **tail** (#6), and consequently both update the data members of the same node when setting the **data** and **next** pointers (#3, #7). This is a data race!

There are similar problems in **pop_head()**: if two threads call concurrently, then they will both read the same value of **head**, and both then overwrite the old value with the same **next** pointer. Both threads will now think they've retrieved the same node: a recipe for disaster. Not only do we have to ensure that only one thread **pop()**s a given item, but we need to ensure that other threads can safely access the **next** member of the node they read from **head**. Of course, this is exactly the problem we saw with **pop()** for our lock-free stack, so any of the solutions for that could be used.

So, if **pop()** is a “solved problem”, what about **push()**? The problem here is that in order to get the required *happens-before* relationship between **push()** and **pop()** we need to set the data items on the dummy node before we update **tail**, but this then means that concurrent calls to **push()** are racing over those very same data items, since they've read the same **tail** pointer.

Handling multiple threads in ***push()***

One option is to add a dummy node between each real node. This way, the only part of the current **tail** node that needs updating is the **next** pointer, which could therefore be made atomic. If a thread managed to successfully change the **next** pointer from **NULL** to its new node then it had successfully added the pointer, otherwise it would have to start again and re-read the **tail**. This would then require a minor change to **pop()** in order to discard nodes with a **NULL** data pointer and loop again. The downside here is that every **pop()** call will typically have to remove *two* nodes, and there's twice as many memory allocations.

A second option is to make the **data** pointer atomic, and set that with a call to compare/exchange. If the call succeeds, then this is our tail node, and we can safely set the **next** pointer to our new node and then update **tail**. If the compare/exchange fails because another thread has stored the data then we loop around, re-read **tail** and start again. If the atomic operations on **std::shared_ptr<>** are lock-free we're home dry. If not, then we need an alternative. One possibility is to have **pop()** return a **std::unique_ptr<>** (after all, it's the only reference to the object) and store the data as a plain pointer in the queue. This would allow us to store it as a **std::atomic<T*>**, which would then support the necessary **compare_exchange_strong()** call. **push()** now looks like listing 7.11:

Listing 7.46: A first attempt at revising push()

```
void push(T new_value)
{
    std::unique_ptr<T> new_data(new T(new_value));
    counted_node_ptr new_next;
    new_next.ptr=new node;
    new_next.external_count=1;
    for(;;)
    {
        node* const old_tail=tail.load();                         #1
        T* old_data=NULL;
        if(old_tail->data.compare_exchange_strong(
            old_data,new_data.get()))                             #2
        {
            old_tail->next=new_next;
            tail.store(new_next.ptr);                             #3
            new_data.release();
            break;
        }
    }
}
```

Cueballs in code and text

This avoids this particular race, but it's not the only one. If we look at the revised version of **push()** in listing 7.11 then we see a pattern we saw in the stack: load an atomic pointer (#1), and dereference that pointer (#2). In the mean time, another thread could update the pointer (#3), eventually leading to the node being deallocated (in **pop()**). If the node is deallocated before we dereference the pointer then we've got undefined behaviour. Ouch.

It's tempting to use the same reference counting scheme again, but now there's an additional twist: this node can potentially be accessed from *two* places: via the **tail** pointer loaded here, and via the **next** pointer in the previous node in the queue. This makes the reference counting more complicated, as we've now got two separate external counts: one in the **next** pointer and one in the **tail** pointer. We can address this by also counting the number of external counters inside the **node** structure, and decreasing this when each external counter is destroyed (as well as adding the corresponding external count to the internal count). If the internal count is zero and there are no external counters then we know the node can safely be deleted. This is a technique I first encountered through Joe Seigh's atomic-ptr-plus project [atomic_ptr_plus]. The revised queue source code is shown in listing 7.12.

Listing 7.47: A lock-free queue with a reference-counted tail

```
template<typename T>
class queue
{
private:
    struct node;

    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };

    struct node_counter
    {
        int internal_count:30;
        unsigned external_counters:2;                                #3
    };

    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;                             #2
        counted_node_ptr next;

        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
            new_count.external_counters=2;                           #4
            count.store(new_count);

            next.ptr=NULL;
            next.external_count=0;
        }

        void release_ref()                                           #8
        {
            node_counter old_counter=
                count.load(std::memory_order_relaxed);
            node_counter new_counter;
            do
            {
                new_counter=old_counter;
                --new_counter.internal_count;
            }
            while(!count.compare_exchange_strong(
                  old_counter,new_counter,
                  std::memory_order_acquire,std::memory_order_relaxed));

            if(!new_counter.internal_count &&
               !new_counter.external_counters)
            {
                delete this;
            }
        }

    };

    std::atomic<counted_node_ptr> head;
    std::atomic<counted_node_ptr> tail;                                #1

    static void increase_external_count(                               #5
        std::atomic<counted_node_ptr>& counter,
        counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!counter.compare_exchange_strong(
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }

    static void free_external_counter(counted_node_ptr &old_node_ptr)  #6
    {
        node* const ptr=old_node_ptr.ptr;
        int const count_increase=old_node_ptr.external_count-2;

        node_counter old_counter=
            ptr->count.load(std::memory_order_relaxed);
        node_counter new_counter;
        do
        {
            new_counter=old_counter;
            --new_counter.external_counters;
            new_counter.internal_count+=count_increase;
        }
        while(!ptr->count.compare_exchange_strong(                     #7
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));

        if(!new_counter.internal_count &&
           !new_counter.external_counters)
        {
            delete ptr;
        }
    }

public:
    queue()
    {
        counted_node_ptr new_node;
        new_node.external_count=1;
        new_node.ptr=new node;

        head.store(new_node);
        tail.store(new_node);
    }

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    ~queue()
    {
        while(pop());
        delete head.load().ptr;
    }
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_external_count(head,old_head);
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                return std::unique_ptr<T>();
            }
            if(head.compare_exchange_strong(old_head,ptr->next))
            {
                T* const res=ptr->data.exchange(NULL);
                free_external_counter(old_head);
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();
        }
    }

    void push(T new_value)
    {
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();

        for(;;)
        {
            increase_external_count(tail,old_tail);

            T* old_data=NULL;
            if(old_tail.ptr->data.compare_exchange_strong(            #9
               old_data,new_data.get()))
            {
                old_tail.ptr->next=new_next;
                old_tail=tail.exchange(new_next);
                free_external_counter(old_tail);
                new_data.release();
                break;
            }
            old_tail.ptr->release_ref();
        }
    }
};
```

Cueballs in code and text

In listing 7.12, **tail** is now an **atomic<counted_node_ptr>** the same as **head** (#1), and the **node** structure has a **count** member to replace the **internal_count** from before (#2). This **count** is a structure containing the **internal_count** and an additional **external_counters** member (#3). It is important to update these counts together as a single entity in order to avoid race conditions, as we shall see shortly.

The **node** is initialized with the **internal_count** set to zero and the **external_counters** set to 2 (#4) since every new node starts out referenced from **tail** and from the **next** pointer of the previous node. **increase_head_count()** has been renamed to **increase_external_count()** (#5) and made into a **static** member function that takes the external counter to update as the first parameter. The counterpart **free_external_counter()** has also been extracted (#6) as it is now used by both **push()** and **pop()**. This is slightly different to the code from the old version of **pop()** since it now has to handle the **external_counters** count. It updates the two counts using a single **compare_exchange_strong()** on the whole **count** structure (#7). The **internal_count** value is updated as before, and the **external_counters** value is decreased. If *both* the values are now zero then there are no more references so the node can be deleted. This has to be done as a single action (which therefore requires the compare/exchange loop) to avoid a race condition: if they are updated separately then two threads may both think they are the last one and thus both delete the node, resulting in undefined behaviour. Also, we know that the **external_counters** value will never be more than 2, so we can use a bit-field here

— this reduces the size of the structure which may make the atomic operations more efficient.

The freshly-extracted **release_ref()** function (#8) has a similar compare/exchange loop, but this one just decreases the **internal_count**. If the **external_counters** value is already zero and the **internal_count** is set to zero then the node is deleted here.

Though this now works, and is race-free, there's still a performance issue: once one thread has started a **push()** operation by successfully completing the **compare_exchange_strong()** on **old_tail.ptr->data** (#9) then no other thread can perform a **push()** operation. Any thread that tries will see the new value, which will cause the the **compare_exchange_strong()** call to fail and make that thread loop again. This is a busy-wait, and as such is a bad idea since it consumes CPU cycles without achieving anything. This calls for the next trick from the lock-free bag of tricks: the waiting thread can *help* the thread that is doing the **push()**.

Helping out another thread

In this case, we know exactly what needs to be done: the **next** pointer on the tail node needs to be set to a new dummy node, and then the **tail** pointer itself must be updated. The thing about dummy nodes is that they're all equivalent, so it doesn't matter if we use the dummy node created by the thread that successfully pushed the data or the dummy node from one of the threads that is waiting to push. If we make the **next** pointer in a node atomic we can then use **compare_exchange_strong()** to set the pointer. Once the **next** pointer is set we can then use a **compare_exchange_weak()** loop to set the **tail** whilst ensuring that it's still referencing the same original node: if it isn't then someone else has updated it and we can stop trying and loop again. This requires a minor change to **pop()** as well in order to load the **next** pointer. The complete code is shown in listing 7.13.

Listing 7.48: A lock-free queue with helping

```
template<typename T>
class queue
{
private:
    struct node;

    struct counted_node_ptr
    {
        int external_count;
        node* ptr;
    };

    struct node_counter
    {
        int internal_count:30;
        unsigned external_counters:2;
    };

    struct node
    {
        std::atomic<T*> data;
        std::atomic<node_counter> count;
        std::atomic<counted_node_ptr> next;                          #1

        node()
        {
            node_counter new_count;
            new_count.internal_count=0;
            new_count.external_counters=2;
            count.store(new_count);

            counted_node_ptr next_node={0};
            next.store(next_node);
        }

        void release_ref()
        {
            node_counter old_counter=
                count.load(std::memory_order_relaxed);
            node_counter new_counter;
            do
            {
                new_counter=old_counter;
                --new_counter.internal_count;
            }
            while(!count.compare_exchange_strong(
                  old_counter,new_counter,
                  std::memory_order_acquire,std::memory_order_relaxed));

            if(!new_counter.internal_count &&
               !new_counter.external_counters)
            {
                delete this;
            }
        }

    };

    std::atomic<counted_node_ptr> head;
    std::atomic<counted_node_ptr> tail;

    static void increase_external_count(
        std::atomic<counted_node_ptr>& counter,
        counted_node_ptr& old_counter)
    {
        counted_node_ptr new_counter;

        do
        {
            new_counter=old_counter;
            ++new_counter.external_count;
        }
        while(!counter.compare_exchange_strong(
              old_counter,new_counter,
              std::memory_order_acquire,
              std::memory_order_relaxed));

        old_counter.external_count=new_counter.external_count;
    }

    void set_new_tail(counted_node_ptr &old_tail,                      #6
                      counted_node_ptr const &new_tail)
    {
        node* const current_tail_ptr=old_tail.ptr;
        while(!tail.compare_exchange_weak(old_tail,new_tail) &&        #7
              old_tail.ptr==current_tail_ptr);
        if(old_tail.ptr==current_tail_ptr)                             #8
        {
            free_external_counter(old_tail);                           #9
        }
        else
        {
            current_tail_ptr->release_ref();                           #10
        }
    }

    static void free_external_counter(counted_node_ptr &old_node_ptr)
    {
        node* const ptr=old_node_ptr.ptr;
        int const count_increase=old_node_ptr.external_count-2;

        node_counter old_counter=
            ptr->count.load(std::memory_order_relaxed);
        node_counter new_counter;
        do
        {
            new_counter=old_counter;
            --new_counter.external_counters;
            new_counter.internal_count+=count_increase;
        }
        while(!ptr->count.compare_exchange_strong(
              old_counter,new_counter,
              std::memory_order_acquire,std::memory_order_relaxed));
        if(!new_counter.internal_count &&
           !new_counter.external_counters)
        {
            delete ptr;
        }
    }

public:
    queue()
    {
        counted_node_ptr new_node;
        new_node.external_count=1;
        new_node.ptr=new node;

        head.store(new_node);
        tail.store(new_node);
    }

    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;

    ~queue()
    {
        while(pop());
        delete head.load().ptr;
    }
    std::unique_ptr<T> pop()
    {
        counted_node_ptr old_head=head.load(std::memory_order_relaxed);
        for(;;)
        {
            increase_external_count(head,old_head);
            node* const ptr=old_head.ptr;
            if(ptr==tail.load().ptr)
            {
                return std::unique_ptr<T>();
            }
            counted_node_ptr next=ptr->next.load();                    #2
            if(head.compare_exchange_strong(old_head,next))
            {
                T* const res=ptr->data.exchange(NULL);
                free_external_counter(old_head);
                return std::unique_ptr<T>(res);
            }
            ptr->release_ref();
        }
    }

    void push(T new_value)
    {
```

```
        std::unique_ptr<T> new_data(new T(new_value));
        counted_node_ptr new_next;
        new_next.ptr=new node;
        new_next.external_count=1;
        counted_node_ptr old_tail=tail.load();

        for(;;)
        {
            increase_external_count(tail,old_tail);

            T* old_data=NULL;
            if(old_tail.ptr->data.compare_exchange_strong(
                   old_data,new_data.get()))
            {
                counted_node_ptr old_next={0};
                if(!old_tail.ptr->next.compare_exchange_strong(       #3
                       old_next,new_next))
                {
                    delete new_next.ptr;                              #4
                    new_next=old_next;                                #5
                }
                set_new_tail(old_tail, new_next);
                new_data.release();
                break;
            }
            else
            {
                counted_node_ptr old_next={0};
                if(old_tail.ptr->next.compare_exchange_strong(        #11
                       old_next,new_next))
                {
                    old_next=new_next;                                #12
                    new_next.ptr=new node;                            #13
                }
                set_new_tail(old_tail, old_next);                     #14
            }
        }
    }
};
```

Cueballs in code and text

#1 `next` pointer is now atomic
#2 We must explicitly `load()` the `next` pointer

Having set the **data** pointer in the node, this new version of **push()** updates the **next** pointer using **compare_exchange_strong()** (#3). If the exchange *fails* then another thread has already set the **next** pointer, so we don't need the new node we allocated at the beginning, so we can delete it (#4). We also want to use the **next** value that the other thread set for updating **tail** (#5).

The actual update of the **tail** pointer has been extracted into **set_new_tail()** (#6). This uses a **compare_exchange_weak()** loop (#7) to update the **tail**, since if other threads are trying to **push()** a new node then the **external_count** part may have changed, and we don't want to lose it. However, we also need to take care that we don't replace the value if another thread has successfully changed it already, otherwise we may end up with loops in the queue, which would be a rather bad idea. Consequently we need to ensure that the **ptr** part of the loaded value is the same if the compare/exchange fails. If the **ptr** is the same once the loop has exited (#8) then we must have successfully set the **tail**, so we need to free the old external counter (#9). If the **ptr** value is different then another thread will have freed the counter, so we just need to release the single reference held by this thread (#10).

If the thread calling **push()** failed to set the **data** pointer this time round the loop, then it can help the successful thread to complete the update. First off, we try and update the **next** pointer to the new node allocated on this thread (#11). If this succeeds then we want to use the node we allocated as the new **tail** node (#12) and we need to allocate another new node in anticipation of actually managing to push an item on the queue (#13). We can then try and set the **tail** node by calling **set_new_tail** before looping round again (#14).

You may have noticed that there's rather a lot of **new** and **delete** calls for such a small piece of code, as new nodes are allocated on **push()** and destroyed in **pop()**. The efficiency of the memory allocator therefore has a considerable impact on the performance of this code: a poor allocator can completely destroy the scalability properties of a lock-free container such as this. The selection and implementation of such allocators is beyond the scope of this book, but it is important to bear in mind that the only way to know an allocator is better is to try it and measure the performance of the code before and after.

That's enough examples for now; instead, let's look at extracting some guidelines for writing lock-free data structures from the examples.

7.3 Guidelines For Writing Lock-free Data Structures

If you've followed through all the examples in this chapter then you'll appreciate the complexities involved in getting lock-free code right. If you're going to design your own data structures then it helps to have some guidelines to focus on. The general guidelines regarding concurrent data structures from the beginning of chapter 6 still apply, but we need more than that. I've pulled a few useful guidelines out from the examples which you can then refer to when designing your own lock-free data structures.

7.3.1 Guideline: Use `std::memory_order_seq_cst` for prototyping

std::memory_order_seq_cst is *much* easier to reason about than any other memory ordering because all such operations form a total order. In all the examples in this chapter, we've started with **std::memory_order_seq_cst** and only relaxed the memory ordering constraints once the basic operations are working. In this sense, using other memory orderings is an *optimization*, and as such we need to avoid doing it prematurely. In general, you can only determine which operations can be relaxed when you can see the full set of code that can operate on the guts of the data structure. Attempting to do otherwise just makes your life harder. This is complicated by the fact that the code may work when tested whilst not being guaranteed: unless you have an algorithm checker that can systematically test all possible combinations of thread visibilities that are consistent with the specified ordering guarantees (and such things do exist), just running the code is not enough.

7.3.2 Guideline: Use a Lock-free memory reclamation scheme

One of the biggest difficulties with lock-free code is managing memory. It is essential to avoid deleting objects when other threads might still have references to them, but we still want to delete the object as soon as possible in order to avoid excessive memory consumption. In this chapter we've seen three techniques for ensuring that memory can safely be reclaimed:

- Waiting until no threads are accessing the data structure and deleting all objects that are pending deletion then;
- Using hazard pointers to identify that a thread is accessing a particular object; and
- Reference counting the objects so that they aren't deleted until there are no outstanding references.

In all cases the key idea is to use some method to keep track of how many threads are accessing a particular object, and only delete each object when it is no longer referenced from anywhere.

7.3.3 Guideline: Identify busy-wait loops and “help” the other thread

In the final queue example we saw how a thread performing a push operation had to wait for another thread also performing push to complete its operation before it could proceed. Left alone, this would have been a busy-wait loop with the waiting thread wasting CPU time failing to proceed. If you end up with a busy-wait loop then you effectively have a blocking operation, and might as well use mutexes and locks. By modifying the algorithm so that the waiting thread performs the incomplete steps if it is scheduled to run before the original thread completes the operation we can remove the busy-wait and the operation is no longer blocking. In the queue example this required changing a data member to be an atomic variable rather than a non-atomic variable and using compare/exchange operations to set it, but in more complex data structures it might require more extensive changes.

7.4 Summary

Following on from the lock-based data structures of chapter 6, this chapter has described simple implementations of various lock-free data structures, starting with a stack and a queue, as before. We saw how we must take care with the memory ordering on our atomic operations to ensure that there are no data races, and that each thread sees a coherent view of the data structure. We also saw how memory management becomes much harder for lock-free data structures than lock-based ones, and examined a couple of mechanisms for handling it. We also saw how we can avoid creating wait loops by helping the thread we are waiting for to complete its operation.

Designing lock-free data structures is a difficult task, and it is easy to make mistakes, but such data structures have scalability properties which are important in some situations. Hopefully, by following through the examples in this chapter and reading the guidelines you'll be better equipped to design your own lock-free data structure, implement one from a research paper or find the bug in the one your former colleague wrote just before he left the company.

Wherever data is shared between threads you need to think about the data structures used and how the data is synchronized between threads. By designing data structures for concurrency we can encapsulate that responsibility in the data structure itself, so the rest of the code can focus on the task it's trying to perform *with* the data rather than the data synchronization. We shall see this in action in chapter 8 as we move on from concurrent data structures to concurrent code in general: parallel algorithms use multiple threads to improve their performance, and the choice of concurrent data structure is crucial where the algorithms need their worker threads to share data.

8

Designing Concurrent Code

Most of the preceding chapters have focused on the tools we have in our new C++0x toolbox for writing concurrent code. In chapters 6 and 7 we looked at how to use those tools to design basic data structures that were safe for concurrent access by multiple threads. Much as a carpenter needs to know more than just how to build a hinge or a joint in order to make a cupboard or a table, we now need to look at how to use these tools in a wider context, so we can build bigger structures that perform useful work. We'll be using multithreaded implementations of some of the C++ Standard Library algorithms as examples, but the same principles apply at all scales of an application.

Just as with any programming project, it is vital to think carefully about the design of concurrent code. However, with multi-threaded code there are even more factors to consider than with sequential code — not only must we think about the usual factors such as encapsulation, coupling and cohesion (which are amply described in the many books on software design), but we also need to consider which data to share, how to synchronize accesses to that data, which threads need to wait for which other threads to complete certain operations and so forth.

In this chapter we'll be focusing on these issues — from the high level (but fundamental) considerations of how many threads to use, which code to execute on which thread and how this can affect the clarity of the code, to the low level details of how to structure the shared data for optimal performance.

Let's start by looking at techniques for dividing work between threads.

8.1 Techniques for Dividing Work Between Threads

Imagine for a moment that you've been tasked with building a house. In order to complete the job, you'll need to dig foundations, build walls, put in plumbing, wire up the electrics and so forth. Theoretically, you could do it all yourself with sufficient training, but it would probably take a long time, and you would be continually switching tasks as necessary. Alternatively, you could hire a few other people to help out. You now have to choose how many people to hire, and what skills they need. You could, for example, hire a couple of people with general skills, and have everybody chip in with everything. You still all switch tasks as necessary, but now things can be done a bit quicker because there's more of you.

Alternatively, you could hire a team of specialists: a bricklayer, a carpenter, an electrician and a plumber, for example. Your specialists just do whatever their speciality is, so if there's no plumbing needed then your plumber sits around drinking tea. Things still get done quicker than before, because there's more of you, and the plumber can put the toilet in whilst the electrician does the wiring in the kitchen, but there's more waiting around when there is no work for a particular specialist to do. Even with the waiting around, you might find that the work is done quicker with specialists than with a team of general handy men though — your specialists don't need to keep changing tools, and they can probably do each their tasks quicker than the generalists can. Whether or not this is the case depends on the particular circumstances — you'd have to try it and see.

Even if you hire specialists, you can still choose to hire different numbers of each — it might make sense to have more bricklayers than electricians for example. Also, the make up of your team and the overall efficiency might change if you had to build more than one house — though your plumber might not have lots of work to do on any given house, you might have enough work to keep him busy all the time if you are building many houses at once. Also, if you don't have to pay your specialists when there's no work for them to do, then you might be able to afford a larger overall team even if only the same number of people are working at any one time.

OK, enough about building — what's all this got to do with threads? Well, with threads the same issues apply — we need to decide how many threads to use, and what tasks they should be doing. We need to decide whether to have “generalist” threads which do whatever work is necessary at any point in time, or “specialist” threads that do one thing well, or some combination. You need to make these choices whatever the driving reason for using concurrency, and quite how you do this will have a crucial effect on the performance and clarity of the code. It is therefore vital to understand the options so we can make an appropriately informed decision when designing the structure of our application. In this section, we're going to look at several techniques for dividing the tasks, starting with dividing data between threads before we do any other work.

8.1.1 Dividing Data Between Threads Before Processing Begins

The easiest algorithms to parallelize are simple algorithms such as **`std::for_each`** which perform an operation on each element in a data set. In order to parallelize such an algorithm we can simply assign each element to one of the processing threads. How the elements are best divided for optimal performance depends very much on the details of the data structure, as we shall see later in this chapter when we look at performance issues.

The simplest means of dividing the data is just to allocate the first *N* elements to one thread, the next *N* elements to another thread, and so on, as in figure 8.1, but other patterns could be used too. However the data is divided, each thread then processes just the elements it has been assigned without any communication with the other threads until it has completed its processing.

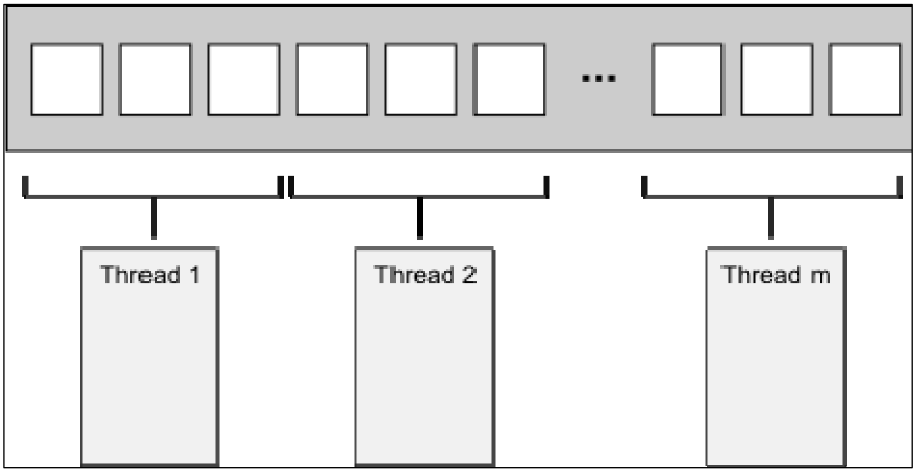


Figure 8.13 Distributing consecutive chunks of data between threads.

This structure will be familiar to anyone who has programmed using the MPI [MPI] or OpenMP [OpenMP] frameworks: a task is split into a set of parallel tasks, the worker threads run these tasks independently, and the results are combined in a final *reduction* step. It is the approach used by the **accumulate** example from section 2.4: in this case both the parallel tasks and the final reduction step are accumulations. For a simple **for_each**, the final step is a no-op as there are no results to reduce.

Identifying this final step as a reduction is important: a naïve implementation such as listing 2.8 will perform this reduction as a final serial step. However, this step can often be parallelized as well: **accumulate** actually *is* a reduction operation itself, so listing 2.8 could be modified to call itself recursively where the number of threads is larger than the minimum number of items to process on a thread, for example. Alternatively, the worker threads could be made to perform some of the reduction steps as each one completes its task, rather than spawning new threads each time.

Though this technique is powerful, it can't be applied to everything: sometimes the data cannot be divided neatly up-front because the necessary divisions only become apparent as the data is processed. This is particularly apparent with recursive algorithms such as quick-sort; they therefore need a different approach.

8.1.2 Dividing Data Recursively

The quick-sort algorithm has two basic steps: partition the data into items that come before or after one of the elements (the pivot) in the final sort order, and recursively sort those two “halves”. You cannot parallelize this by simply dividing the data up front, since it is only be processing the items that you know which “half” they go in. If we're going to parallelize such an algorithm we need to make use of the recursive nature: with each level of recursion there are *more* calls to the quick-sort function, because we have to sort both the elements that belong before the pivot, *and* those that belong after it. These recursive calls are entirely independent, as they access separate sets of elements, and so are prime candidates for concurrent execution. Figure 8.2 shows such recursive division.

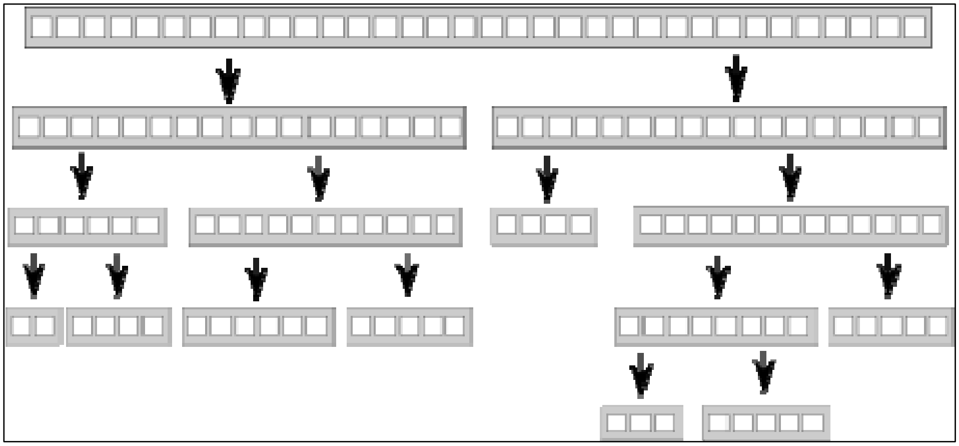


Figure 8.14 Recursively dividing data.

In chapter 4, we saw an such an implementation: rather than just performing two recursive calls for the higher and lower chunks, we wrote a **`spawn_task`** function to launch one of the recursive calls on a separate thread at each stage.

Unfortunately, if you are sorting a large set of data, this will quickly result in a lot of threads. As we shall see when we look at performance, if we have too many threads we might actually *slow down* the application. There's also a possibility of running out of threads if the data set is very large. The idea of dividing the overall task in a recursive fashion like this is a good one; we just need to keep a tighter rein on the number of threads. One alternative is to use the **std::thread::hardware_concurrency()** function to choose the number of threads, as we did with the parallel version of **accumulate()** from listing 2.8. Then, rather than starting a new thread for the recursive calls we can just push the chunk to be sorted onto a thread-safe stack such as one of those described in chapters 6 and 7. If a thread has nothing else to do, either because it has finished processing all its chunks, or because it is waiting for a chunk to be sorted, then it can take a chunk from the stack and sort that.

Listing 8.1 shows a sample implementation that uses this technique.

Listing 8.49: Parallel Quick Sort using a stack of pending chunks to sort

```
template<typename T>
struct sorter                                                          #2
{
    struct chunk_to_sort
    {
        std::list<T> data;
        bool end_of_data;
        std::promise<std::list<T> > promise;

        chunk_to_sort(bool done=false):
            end_of_data(done)
        {}

    };

    thread_safe_stack<chunk_to_sort> chunks;                           #3
    std::vector<std::thread> threads;                                  #4
    unsigned const max_thread_count;

    sorter():
        max_thread_count(std::thread::hardware_concurrency()-1)
    {}

    ~sorter()                                                          #16
    {
        for(unsigned i=0;i<threads.size();++i)
        {
            chunks.push(std::move(chunk_to_sort(true)));               #18
        }

        for(unsigned i=0;i<threads.size();++i)
        {
            threads[i].join();                                         #19
        }
    }

    bool try_sort_chunk()
    {
        boost::shared_ptr<chunk_to_sort > chunk=chunks.pop();          #11
        if(chunk)
        {
            if(chunk->end_of_data)                                     #20
            {
                return false;
            }
            sort_chunk(chunk);                                         #12
        }
        return true;
    }

    std::list<T> do_sort(std::list<T>& chunk_data)                     #5
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }

        std::list<T> result;
        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();

        typename std::list<T>::iterator divide_point=                  #6
            std::partition(chunk_data.begin(),chunk_data.end(),
                           less_than<T>(partition_val));

        chunk_to_sort new_lower_chunk;
        new_lower_chunk.data.splice(new_lower_chunk.data.end(),
                                    chunk_data,chunk_data.begin(),
                                    divide_point);

        std::unique_future<std::list<T> > new_lower=
            new_lower_chunk.promise.get_future();
        chunks.push(std::move(new_lower_chunk));                       #7
        if(threads.size()<max_thread_count)                            #8
        {
            threads.push_back(std::thread(&sorter<T>::sort_thread,this));
        }

        std::list<T> new_higher(do_sort(chunk_data));

        result.splice(result.end(),new_higher);
        while(!new_lower.is_ready())                                   #9
        {
            try_sort_chunk();                                           #10
        }

        result.splice(result.begin(),new_lower.get());
        return result;
    }

    void sort_chunk(boost::shared_ptr<chunk_to_sort > const& chunk)
    {
        chunk->promise.set_value(do_sort(chunk->data));                 #13
    }

    void sort_thread()
    {
        while(try_sort_chunk())                                         #21
        {
            std::this_thread::yield();
        }
    }
};


template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)                    #1
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;

    return s.do_sort(input);                                            #17
}
```

Cueballs in code and text

Here, the **parallel_quick_sort** function (#1) delegates most of the functionality to the **sorter** class (#2), which provides an easy way of grouping the stack of unsorted chunks (#3) and the set of threads (#4). The main work is done in the **do_sort** member function (#5), which does the usual partitioning of the data (#6). This time, rather than spawning a new thread for one chunk, it pushes it onto the stack (#7), and spawns a new thread whilst we still have processors to spare (#8). Because the lower chunk might be handled by another thread, we then have to wait for it to be ready (#9). In order to help things along (in case we're the only thread, or all the others are already busy), we try and process chunks from the stack on this thread whilst we're waiting (#10). **try_sort_chunk** essentially just pops a chunk off the stack (#11) and sorts it (#12), storing the result in the **promise** ready to be picked up by the thread that posted the chunk on the stack (#13).

Our freshly spawned threads just sit in a loop trying to sort chunks off the stack (#14). In between checking, they yield to other threads (#15) to give them a chance to put some more work on the stack. **try_sort_chunk** returns **true** even if there wasn't a chunk to sort in order that the threads don't terminate early — just because there aren't any chunks to sort *right now* doesn't mean that there won't be in half a nanosecond. Instead, we rely on the destructor of our **sorter** class (#16) to tidy up: when all the data has been sorted, **do_sort** will return (even though the worker threads are still running), so our main thread will return from **parallel_quick_sort** (#17), and thus destroy our **sorter** object. This posts special values to the stack (#18) and waits for the threads to finish (#19). The special chunk values cause **try_sort_chunk** to return **false** (#20), and thus terminates the loop in the thread function (#21).

With this approach we no longer have the problem of unbounded threads that we have with a **spawn_task** that launches a new thread. Instead we limit the number of threads to the value of **std::thread::hardware_concurrency()** in order to avoid excessive task switching. We do, however, have another potential problem: the management of these threads, and the communication between them adds quite a lot of complexity to the code. Also, though the threads are processing separate data elements, they all access the stack to add new chunks, and to remove chunks for processing. This heavy contention can reduce performance, even if you use a lock-free (and hence non-blocking) stack, for reasons that we shall see shortly.

This approach is actually a very specialized version of a *thread pool* — there is a set of threads that take work to do from a list of pending work, do the work, and then go back to the list for more. Some of the potential problems with thread pools (including the contention on the work list) and ways of addressing them are covered in chapter 9.

Both dividing the data before processing begins, and dividing it recursively presume that the data itself is fixed beforehand, and you're just looking at ways of dividing it. This is not always the case: if the data is dynamically generated, or is coming from external input then this approach doesn't work. In this case, it might make more sense to divide the work by task type, rather than dividing based on the data.

8.1.3 Dividing Work By Task Type

Dividing work between threads by allocating different chunks of data to each thread (whether up front or recursively during processing) still assumes that the threads are going to be doing essentially the same work to each chunk of data. An alternative to dividing the work is to make the threads specialists, where they each perform a distinct task, just as plumbers and electricians perform distinct tasks when building a house. Threads may or may not work on the same data, but if they do then it is for different purposes.

This is the sort of division of work that results from separating concerns with concurrency: each thread has a different task, which it carries out independently of other threads. Occasionally other threads may give it data or trigger events which it needs to handle, but generally each thread focuses on doing one thing well. In itself, this is just general good design: each piece of code should have a single responsibility.

Dividing Work By Task Type to Separate Concerns

A single-threaded application has to handle conflicts with the single responsibility principle where there are multiple tasks that need to be run continuously over a period of time, or where the application needs to be able to handle incoming events (such as user key presses or incoming network data) in a timely fashion, even while other tasks are ongoing. In the single-threaded world we end up by manually writing code that performs a bit of task A, performs a bit of task B, checks for key presses, checks for incoming network packets, and then loops back to perform another bit of task A. This means that the code for task A ends up being complicated by the need to save its state and return control to the main loop periodically. If you add too many tasks to the loop, things might slow down too far, and the user may find it takes too long to respond to the key press. I'm sure we've all seen the extreme form of this in action with some application or other: you set it doing some task and the interface freezes until it has completed the task.

This is where threads come in. If we run each of the tasks in a separate thread then the operating system handles this for us. In the code for task A we can just focus on performing the task, and not worrying about saving state and returning to the main loop, or how long we spend between doing so. The operating system will automatically save the state and switch to task B or C when appropriate, and if the target system has multiple cores or processors then tasks A and B may well be able to run truly concurrently. The code for handling the key press or network packet will now be run in a timely fashion, and everybody wins: the user gets timely responses, and you as developer have simpler code because each thread can focus on doing operations related directly to its responsibilities, rather than getting mixed up with control flow and user interaction.

That sounds like a nice rosy vision. Can it really be like that? As with everything, it depends on the details. If everything really is independent, and the threads have no need to communicate with each other, then it really is this easy. Unfortunately, the world is rarely like that. These nice background tasks are often doing something that the user requested, and need to let the user know when they're done by updating the user interface in some manner. Alternatively, the user might want to cancel the task, which therefore requires the user interface to somehow send a message to the background task telling it to stop. Both these cases require careful thought and design, and suitable synchronization, but the concerns are still separate. The user interface thread still just does the user interface, but it might have to update it when asked to by others threads. Likewise, the thread running the background task still just focuses on the operations required for that task, it just happens that one of them is "allow task to be stopped by another thread". In neither case do the threads care where the request came from, only that it was intended for them, and relates directly to their responsibilities.

There are two big dangers with separating concerns with multiple threads. The first is that you'll end up separating the *wrong* concerns. The symptoms to check for are that there is a lot of data shared between the threads, or the different threads end up waiting for each other; both cases boil down to too much communication between threads. If this happens, it is worth looking at the reasons for the communication: if all the communication relates to the same issue, maybe that should be the key responsibility of a single thread, and extracted from all the threads that refer to it. Alternatively, if two threads are communicating a lot with each other, but much less with other threads then maybe they should be combined into a single thread.

When dividing work across threads by task type, you don't have to limit yourself to completely isolated cases: if multiple sets of input data require the same *sequence* of operations to be applied, we can divide the work so each thread performs one stage from the overall sequence.

Dividing a Sequence of Tasks Between Threads

If your task consists of applying the same sequence of operations to many independent data items, then you can use a *pipeline* to exploit the available concurrency of your system. This is by analogy to a physical pipeline: data flows in at one end, through a series of operations (pipes) and out at the other.

To divide the work this way, you create a separate thread for each stage in the pipeline: one thread for each of the operations in the sequence. When the each operation is completed the data element is put on a queue to be picked up by the next thread. This allows the thread performing the first operation in the sequence to start on the next data element whilst the second thread in the pipeline is working on the first element.

This is an alternative to just dividing the data between threads as described in section 8.1.1, and is appropriate in circumstances where the input data itself is not all known when the operation is started. For example, the data might be coming in over a network, or the first operation in the sequence might be to scan a file system in order to identify files to process.

Pipelines are also good where each operation in the sequence is time consuming: by dividing the tasks between threads rather than the data, then it changes the performance profile. Suppose you have 20 data items to process, on 4 cores, and each data item requires 4 steps which take 3 seconds each. If you divide the data between 4 threads, then each thread has 5 items to process. Assuming there's no other processing that might affect the timings, after 12 seconds you'll have 4 items processed, after 24 seconds 8 items processed, and so forth. All 20 items will be done after 1 minute. With a pipeline, things work differently. Your four steps can be assigned one to each processing core. Now, the first item has to be processed by each core, so it still takes the full 12 seconds. Indeed, after 12 seconds you only have one item processed, which is not as good as with the division by data. However, once the pipeline is *primed*, things proceed a bit differently: after the first core has processed the first item it moves onto the second, so once the final core has processed the first item it can perform its step on the second. We now get one item processed every 3 seconds rather than having the items processed in batches of four every 12 seconds.

The overall time to process the entire batch takes longer since we have to wait 9 seconds before the final core starts processing the first item, but smoother, more regular processing can be beneficial in some circumstances. Consider, for example a system for watching high-definition digital videos. In order for the video to be watchable you typically need at least twenty five frames per second, and ideally more. It's no good your software being able to decode 100 frames per second if they come in one big block once a second: the viewer needs these to be evenly spaced to give the impression of continuous movement. On the other hand, viewers are probably happy to accept a couple of second delay when they *start* watching a video. In this case, parallelizing using a pipeline that outputs frames at a nice steady rate is probably preferable.

Having looked at various techniques for dividing the work between threads, let's take a look at the factors affecting the performance of a multi-threaded system, and how that can impact our choice of techniques.

8.2 Factors Affecting the Performance of Concurrent Code

If you're using concurrency in order to improve the performance of your code on systems with multiple processors, you really need to ensure you know what factors are going to affect the performance. Even if you're just using multiple threads to separate concerns you need to ensure that this doesn't adversely affect the performance — customers will not thank you if your application runs *slower* on their shiny new 16-core machine than it did on their old single-core one.

As we shall see shortly, there are many factors that affect the performance of multi-threaded code — even something as simple as changing *which* data elements are processed by each thread (whilst keeping everything else identical) can have a dramatic effect on performance. So, without further ado let's look at some of these factors, starting with the obvious one — how many processors does our target system have?

8.2.1 How many processors?

The number (and structure) of processors is the first big factor that affects the performance of a multi-threaded application, and it's quite a crucial one. In some cases you do know exactly what the target hardware is, and can thus design with this in mind, and take real measurements on the target system or an exact duplicate. If so, you're one of the lucky ones: in general we don't have that luxury. We might be developing on a *similar* system, but the differences can be crucial. For example, you might be developing on a dual- or quad-core system, but your customers' systems may have one multi-core processor (with any number of cores), or multiple single-core processors, or even multiple multi-core processors. The behaviour and performance characteristics of a concurrent program can vary considerably under such different circumstances, so we need to think carefully about what the impact may be, and test things where possible.

To a first approximation, a single 16-core processor is the same as 4 quad-core processors or 16 single-core processors: in each case the system can run 16 threads concurrently. If you want to take advantage of this, your application must have at least 16 threads: if it has fewer than 16, then you're leaving processor power on the table (unless the system is running other applications too, but we'll ignore that possibility for now). On the other hand, if you have more than 16 threads actually ready to run (and not blocked, waiting for something) then your application will waste processor time switching between the threads, as discussed in chapter 1. When this happens, the situation is called *over-subscription*.

To allow applications to scale the number of threads in line with the number of threads the hardware can run concurrently, the C++0x Standard Thread Library provides **`std::thread::hardware_concurrency()`**. We've already seen how that can be used to scale the number of threads to the hardware.

One additional twist to this is that the ideal algorithm for a problem can depend on the size of the problem compared to the number of processing units. If you have a *massively parallel* system with many processing units, then an algorithm that performs more operations overall may finish quicker than one that performs fewer operations, since each processor only performs a few operations.

As the number of processors increases, so does the likelihood and performance impact of another problem: that of multiple processors trying to access the same data.

8.2.2 Data Contention and Cache Ping Pong

If two threads are executing concurrently on different processors and they are both *reading* the same data then this is will not usually cause a problem: the data will be copied into their respective caches, and both processors can proceed. However, if one of the threads *modifies* the data, this change then has to propagate to the cache on the other core, which takes time. Depending on the nature of the operations on the two threads, and the memory orderings used for the operations, such a modification may cause the second processor to stop in its tracks and wait for the change to propagate through the memory hardware. In terms of CPU instructions, this can be a *phenomenally* slow operation, equivalent to many hundreds of individual instructions, though the exact timing depends strongly on the physical structure of the hardware.

Consider the following simple piece of code:

```
std::atomic<unsigned long> counter(0);
void processing_loop()
{
    while(counter.fetch_add(1,std::memory_order_relaxed)<100000000)
    {
        do_something();
    }
}
```

The **counter** is global, so any threads that call **processing_loop()** are modifying the same variable. Therefore, for each increment the processor must ensure it has an up-to-date copy of **counter** in its cache, modify the value, and publish it to other processors. Even though we're using **std::memory_order_relaxed**, so the compiler doesn't have to synchronize any other data, **fetch_add** is a read-modify-write operation, and therefore needs to retrieve the most recent value of the variable. If another thread on another processor is running the same code, the data for **counter** must therefore be passed back and forth between the two processors and their corresponding caches so that each processor has the latest value for **counter** when it does the increment. If **do_something()** is short enough, or there are too many processors running this code then the processors might actually find themselves *waiting* for each other: one processor is ready to update the value, but another processor is currently doing that, so it has to wait until the second processor has completed its update and the change has propagated. This situation is called *high contention*. If the processors rarely have to wait for each other then you have *low contention*.

In a loop like this one, the data for **counter** will be passed back and forth between the caches many times. This is called *cache ping pong*, and can seriously impact the performance of the application. If a processor stalls due to waiting for a cache transfer, it cannot do *any* work in the mean time, even if there are other threads waiting that could do useful work, so this is bad news for the whole application.

You might think that this won't happen to you: after all, you don't have any loops like that. Are you sure? What about mutex locks? If you acquire a mutex in a loop, your code is very similar to that above from the point of view of data accesses: in order to lock the mutex, another thread must transfer the data that makes up the mutex over to its processor and modify it. When it is done, it modifies the mutex again to unlock it, and the mutex data has to be transferred over to the next thread to acquire the mutex. This transfer time is *in addition* to any time that the second thread has to wait for the first to release the mutex.

```
std::mutex m;
my_data data;
void processing_loop_with_mutex()
{
    while(true)
    {
        std::lock_guard<std::mutex> lk(m);
        if(done_processing(data)) break;
    }
}
```

Now, here's the worst part: if the data and mutex really are accessed by more than one thread, then as you add more cores and processors to the system, the chances of high contention, and one processor having to wait for another *goes up*.

There can be two reasons for this. Firstly, if you are using multiple threads to process the same data quicker, then the threads are competing for the data, and thus compete for the same mutex. The more of them there are, the more likely they will try and acquire the mutex at the same time or access the atomic variable at the same time, and so forth.

The effects of contention with mutexes are usually different to the effects of contention with atomic operations for the simple reason that the use of a mutex naturally serializes threads at the operating system level, rather than at the processor level. If you have enough threads ready to run, the operating system can schedule another thread to run whilst one thread is waiting for the mutex, whereas a processor stall prevents any threads from running on that processor. However, it will still impact the performance of those threads that *are* competing for the mutex — they can only run one at a time, after all.

If this cache ping-pong is bad, how can we avoid it? As we shall see later in the chapter, the answer ties in nicely with general guidelines for improving the potential for concurrency: do what you can to reduce the potential for two threads to compete for the same memory location.

It's not quite that simple, though; things never are. Even if a particular memory location is only ever accessed by one thread, you can *still* get cache ping-pong due to an effect known as *false sharing*.

8.2.3 False Sharing

Processor caches don't generally deal in individual memory locations; instead they deal in blocks of memory called *cache lines*. These blocks of memory are typically 32 or 64 bytes in size, but the exact details depend on the particular processor model being used. Because the cache hardware only deals in cache-line-sized blocks of memory, small data items in adjacent memory locations will be in the same cache line. Sometimes this is good: if a set of data accessed by a thread is in the same cache line this is better for the performance of the application than if the same set of data was spread over multiple cache lines. However, if the data items in a cache line are unrelated and need to be accessed by different threads, then this can be a major cause of performance problems.

Suppose you have an array of **int** values, and a set of threads that each access their own entry in the array, but do so repeatedly, including updates. Since an **int** is typically much smaller than a cache line, quite a few of those array entries will be in the same cache line. Consequently, even though each thread only accesses its own array entry, the cache hardware *still* has to play cache ping pong: every time the thread accessing entry 0 needs to update the value ownership of the cache line needs to be transferred to the processor running that thread, only to be transferred to the cache for the processor running the thread for entry 1 when that thread needs to update its data item. The cache line is shared, even though none of the data is: hence *false sharing*. The solution here is to structure the data so that data items to be accessed by the same thread are close together in memory (and thus more likely to be in the same cache line), whilst those that are to be accessed by separate threads are far apart in memory, and thus more likely to be in separate cache lines. We'll see how this affects the design of the code and data later in this chapter.

If having multiple threads access data from the same cache line is bad, how does the memory layout of data accessed by a single thread affect things?

8.2.4 How Close is My Data?

Whereas false sharing is caused by having data accessed by one thread too close to data accessed by another thread, another pitfall associated with data layout directly impacts the performance of a single thread on its own. The issue is data proximity: if the data accessed by a single thread is spread out in memory then it is likely that it lies on separate cache lines. On the flip side, if the data accessed by a single thread is close together in memory then it is more likely to lie on the same cache line. Consequently, if data is spread out then more cache lines must be loaded from memory onto the processor cache, which can increase memory access latency, and reduce performance compared to data that is located close together.

Also, if the data is spread out there is an increased chance that a given cache line containing data for the current thread also contains data that is *not* for the current thread. At the extreme there will be more data in the cache that we don't care about than data that we do. This wastes precious cache space, and thus increases the chance that the processor will have to fetch a data item from main memory even if it once held it in the cache, as it had to remove the item from the cache to make room for another.

Now, this is important with single threaded code, so why am I bringing it up here? The reason is *task switching*. If there are more threads than cores in the system, then each core is going to be running multiple threads. This increases the pressure on the cache, as we try and ensure that different threads are accessing different cache lines in order to avoid false sharing. Consequently, when the processor switches threads it is more likely to have to reload the cache lines if the each thread uses data spread across multiple cache lines, than if each thread's data is close together in the same cache line.

If there are more threads than cores or processors then the operating system might also choose to schedule a thread on one core for one time slice and then on another core for the next time slice. This will therefore require transferring the cache lines for that thread's data from the cache for the first core to the cache for the second — the more cache lines that need transferring, the more time consuming this will be. Though operating systems typically will avoid this when they can, it does happen, and does impact performance when it happens.

Task switching problems are particularly prevalent when there are lots of threads *ready to run* as opposed to *waiting*. This is an issue we've already touched on — over-subscription.

8.2.5 Over-subscription and Excessive Task Switching

In multi-threaded systems, it is typical to have more threads than processors, unless you're running on *massively parallel* hardware. However, threads often spend time waiting for external I/O to complete, or blocked on mutexes or waiting for condition variables and so forth, so this is not a problem — having the extra threads enables the application to perform useful work rather than having processors sitting idle whilst the threads wait.

However, this is not always a good thing: if you have *too many* additional threads then there will be more threads *ready to run* than there are available processors, and the operating system will have to start task switching quite heavily in order to ensure they all get a fair time slice. As we saw in chapter 1, this can increase the overhead of the task switching, as well as compounding any cache problems due to lack of proximity. Over-subscription can arise when you have a task that repeatedly spawns new threads without limits, as the recursive quick sort from chapter 4 did, or where the natural number of threads when you separate by task type is more than the number of processors, and the work is naturally CPU-bound rather than I/O-bound.

If you are simply spawning too many threads due to data division then you can limit the number of worker threads, as we saw in section 8.1.2. If the over-subscription is due to the natural division of work, then there's not a lot that can be done to ameliorate the problem save for choosing a different division. In that case, choosing the appropriate division may require more knowledge of the target platform than you have available, and is only worth doing if performance is unacceptable and it can be demonstrated that changing the division of work does improve performance.

There are of course other factors that can affect the performance of multi-threaded code — the cost of cache ping-pong can vary quite considerably between two single-core processors, and a single dual-core processor, even if they are the same CPU type and clock speed, for example — but these are the major ones that will have a very visible impact. Let's now take a look at how that affects the design of the code and data structures.

8.3 Designing Data Structures for Multi-threaded Performance

In section 8.1 we looked at various ways of dividing work between threads, and in section 8.2 we looked at various factors that can affect the performance of our code. How can we use this information when designing data structures for multi-threaded performance? This is a different question to that addressed in chapters 6 and 7, which were about designing data structures that are safe for concurrent access — as we've just seen in section 8.2, the layout of the data used by a single thread can have an impact, even if that data is not shared with any other threads.

The key things to bear in mind when designing our data structures for multi-threaded performance are *contention*, *false-sharing* and *data proximity*. All three of these can have a big impact on performance, and you can often improve things just by altering the data layout, or changing which data elements are assigned to which thread. First off let's look at an easy win — dividing array elements between threads.

8.3.1 Dividing Array Elements for Complex Operations

Suppose you're doing some heavy duty maths, and you need to multiply two large square matrices together. To multiply matrices, you multiply each element in the first *row* of the first matrix with the corresponding element of the first *column* of the second matrix and add up the products to give the top left element of the result. You then repeat this with the second row and the first column to give the second element in the first column of the result, and with the first row and second column to give the first element in the second column of the result, and so forth. This is shown in figure 8.3, where the highlighting shows the second row of the first matrix is paired with the third column of the second matrix to give the entry in the second row of the third column of the result.

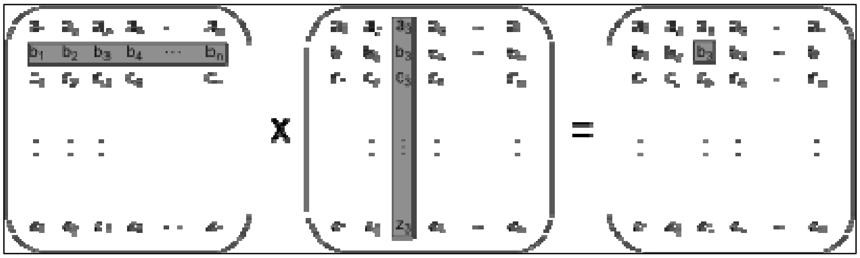


Figure 8.15 Matrix multiplication

Now let's assume that these are *large* matrices with several thousand rows and columns, in order to make it worthwhile using multiple threads to optimize the multiplication. Typically, a matrix is represented by a big array in memory, with all the elements of the first row followed by all the elements of the second row, and so forth. To multiply our matrices we've thus got three of these huge arrays. In order to get optimal performance, we need to pay careful attention to the data access patterns, particularly the writes to the third array.

There are two many ways we can divide the work between threads. Assuming we have more rows/columns than available processors, we could have each thread calculate the values for a number of columns in the result matrix, or each thread calculate the result for a number of rows, or even have each thread calculate the results for a rectangular subset of the matrix.

Back in 8.2.3 and 8.2.4, we've already seen that it is better to access contiguous elements from an array rather than values all over the place, because this reduces cache usage and the chance of false sharing. If we have each thread do a set of columns, then it needs to read every value from the first matrix, and the values from the corresponding columns in the second matrix, but we only have to write the column values. Since the matrices are stored with the rows contiguous, this means that we're accessing *N* elements from the first row, *N* elements from the second, and so forth (where *N* is the number of columns we're processing). Since other threads will be accessing the other elements of each row, it is clear that we ought to be accessing adjacent columns, so the *N* elements from each row are adjacent, and we minimize false sharing. Of course, if the space occupied by our *N* elements is an exact number of cache lines, then there will be no false sharing because threads will be working on separate cache lines.

On the other hand, if we have each thread do a set of *rows* then it needs to read every value from the *second* matrix, and the values from the corresponding *rows* of the *first* matrix, but it only has to write the row values. Since the matrices are stored with the rows contiguous, we are now accessing *all* elements from *N* rows. If we again choose adjacent rows, this means that the thread is now the *only* thread writing to those *N* rows: it has a contiguous block of memory which is not touched by any other thread. This is likely an improvement over having each thread do a set of columns, as the only possibility of false sharing is for the last few elements of one block with the first few of the next, but it is worth timing it on the target architecture to confirm.

What about our third option: dividing into rectangular blocks? This can be viewed as dividing into columns, and then dividing into rows. As such it has the same false-sharing potential as division by columns. If you can choose the number of columns in the block to avoid this possibility, then there is an advantage to rectangular division from the *read* side: you don't need to read the entirety of either source matrix. You only need to read the values corresponding to the rows and columns of the target rectangle. To look at this in concrete terms, consider multiplying two matrices that have 1000 rows and 1000 columns. That's 1 million elements. If you have 100 processors, they can do 10 rows each for a nice round 10000 elements. However, to calculate the results of those 10000 elements they need to access the entirety of the second matrix (1 million elements) plus the 10000 elements from the corresponding rows in the first matrix, for a grand total of 1010000 elements. On the other hand, if they each do a block of 100 elements by 100 elements (which is still 10000 elements total), then they need to access the values from 100 rows of the first matrix (100 x 1000 = 100000 elements) and 100 columns of the second matrix (another 100000). This is only 200000 elements, which is a five-fold reduction in the number of elements read. If you're reading fewer elements, there's less chance of a cache miss, and the potential for greater performance.

It may therefore be better to divide the result matrix up into small square or almost-square blocks rather than have each thread do the entirety of a small number of rows. Of course, you can adjust the size of each block at runtime, depending on the size of the matrices and the available number of processors. As ever, if performance is important then it is vital to profile various options on the target architecture.

Now, chances are you're not doing matrix multiplication, so how does this apply to you? The same principles apply to any situation where you have large blocks of data to divide between threads — look at all the aspects of the data access patterns carefully, and identify the potential causes of performance hits. There may be similar circumstances in your problem domain where changing the division of work can improve performance without requiring any change to the basic algorithm.

OK, so we've looked at how access patterns in arrays can affect performance. What about other types of data structure?

8.3.2 Data Access Patterns in Other Data Structures

Fundamentally, the same considerations apply when trying to optimize the data access patterns of other data structures as when optimizing the access to arrays:

- try to adjust the data distribution between threads so that data that is close together is worked on by the same thread,
- try to minimize the data required by any given thread, and
- try to ensure that data accessed by separate threads is sufficiently far apart to avoid false sharing.

Of course, that's not easy to apply to other data structures — for example, binary trees are inherently difficult to subdivide in any unit other than a sub-tree, which may or may not be useful, depending how balance the tree is, and how many sections you need to divide it into. Also, the nature of the trees means that the nodes are likely dynamically allocated, and thus end up in different places on the heap.

Now, having data end up in different places on the heap isn't a particular problem in itself, but it does mean that the processor has to keep more things in cache. This can actually be a beneficial — if multiple threads need to traverse the tree then they all need to access the tree nodes, but if the tree nodes only contain *pointers* to the real data held at the node, then the processor only has to load the data from memory if it is actually needed. If the data is being modified by the threads that need it this can avoid the performance hit of false sharing between the node data itself and the data that provides the tree structure.

There's actually a similar issue with data protected by a mutex. Suppose you have a simple class which contains a few data items and a mutex used to protect accesses from multiple threads. If the mutex and the data items are close together in memory, this is ideal for a thread that acquires the mutex: the data it needs may well already be in the processor cache, because it was just loaded in order to modify the mutex. However, there is also a downside: if other threads try and lock the mutex whilst it is held by the first thread, they will need access to that memory. Mutex locks are typically implemented as a read-modify-write atomic operation on a memory location within the mutex to try and acquire the mutex, followed by a call to the operating system kernel if the mutex is already locked. This read-modify-write operation may well cause the data held in the cache by the thread that owns the mutex to be invalidated. As far as the mutex goes, this isn't a problem — that thread isn't going to touch the mutex until it unlocks it. However, if the mutex shares a cache line with the data being used by the thread then the thread that owns the mutex can take a performance hit *because another thread tried to lock the mutex*!

One way to test whether this kind of false sharing is a problem is to add huge blocks of padding between the data elements that can be concurrently accessed by different threads. e.g.

```
    struct protected_data
    {
        std::mutex m;
        char padding[65536];
        my_data data_to_protect;
    };
```

to test the mutex contention issue, or

```
    struct my_data
    {
        data_item1 d1;
        data_item2 d2;
        char padding[65536];
    };
    my_data some_array[256];
```

to test for false sharing of array data. If this improves the performance then you know that false sharing was a problem, and can either leave the padding in or work to eliminate the false sharing in another way by rearranging the data accesses.

Of course, there's more than just the data access patterns to consider when designing for concurrency, so let's take a look at some of these additional considerations.

8.4 Additional Considerations when Designing Code for Concurrency

So far in this chapter we've looked at ways of dividing work between threads, factors affecting performance, and how these factors affect our choice of data access patterns and data structures. There's more to designing code for concurrency than just that, though: you also need to consider things such as exception safety, and scalability. Code is said to be *scalable* if the performance (whether in terms of reduced speed of execution or increased throughput) increases as more processing cores are added to the system. Ideally the performance increase is linear so a system with 100 processors performs 100 times better than a system with 1 processor.

Whilst code can work even if it is not scalable — a single-threaded application is certainly not scalable, for example — exception safety is a matter of correctness. If your code is not exception safe then you can end up with broken invariants or race conditions, or your application might terminate unexpectedly because an operation threw an exception. With this in mind, we'll look at exception safety first.

8.4.1 Exception Safety in Parallel Algorithms

Exception safety is an essential aspect of good C++ code, and code that uses concurrency is no exception. In fact, parallel algorithms often need more care taken with regards to concurrency than normal sequential algorithms. If an operation in a sequential algorithm throws an exception, then the algorithm only has to worry about ensuring that it tidies up after itself to avoid resource leaks and broken invariants; it can merrily allow the exception to propagate to the caller for them to handle. By contrast, in a parallel algorithm many of the operations will be running on separate threads. In this case, the exception cannot be allowed to propagate, as it is on the wrong call stack — if a function spawned on a new thread exits with an exception then the application is terminated.

As a concrete example, let's revisit the **parallel_accumulate** function from listing 2.8, which is reproduced below as listing 8.2.

Listing 8.50: A 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);                   #10
    }
};

template<typename Iterator,typename T>
T parallel_accumulate(Iterator first,Iterator last,T init)
{
    unsigned long const length=std::distance(first,last);            #1

    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);                               #2
    std::vector<std::thread>  threads(num_threads-1);                #3

    Iterator block_start=first;                                      #4
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        Iterator block_end=block_start;                              #5
        std::advance(block_end,block_size);
        threads[i]=std::thread(                                      #6
            accumulate_block<Iterator,T>(),
            block_start,block_end,std::ref(results[i]));
        block_start=block_end;                                       #7
    }
    accumulate_block()(block_start,last,results[num_threads-1]);     #8

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

    return std::accumulate(results.begin(),results.end(),init);      #9
}
```

Cueballs in code and text

Now, let's go through and identify the places where an exception can be thrown: basically anywhere where we call a function we know can throw, or we perform an operation on a user-defined type which may throw.

First up, we've got the call to **distance** (#1), which performs operations on the user-supplied iterator type. Since we haven't yet done any work, and this is on the calling thread, this is fine. Next up, we've got the allocation of the **results** vector (#2), and the **threads** vector (#3). Again, these are on the calling thread, and we haven't done any work or spawned any threads, so this is fine. Of course, if the construction of **threads** throws then the memory allocated for **results** will have to be cleaned up, but the destructor will take care of that for us.

Skipping over the initialization of **block_start** (#4) as that's similarly safe, we come to the operations in the thread-spawning loop (#5, #6 & #7). Once we've been through the creation of the first thread at #6 we're in trouble if we throw any exceptions: our new threads will become detached, and continue running with references to the (now destroyed) elements of **results**, as well as copies of the supplied iterators. This is not a good place to be.

The call to **accumulate_block** (#8) can throw with similar problems: our new threads will become detached and continue running. On the other hand, the final call to **std::accumulate** (#9) can throw without causing any hardship, as all the threads have been joined by this point.

That's it for the main thread, but there's more: the calls to **accumulate_block** on the new threads might throw at #10. We've got no **catch** blocks, so this exception will be left unhandled, and cause the library to call **std::terminate()** to abort the application.

In case it's not glaringly obvious: **this code is not exception safe**.

Adding Exception Safety

OK, so we've identified all the possible **throw** points and the nasty consequences of exceptions. What can we do about it? Let's start by addressing the issue of the exceptions thrown on our new threads.

We encountered the tool for this job in chapter 4. If we look carefully at what we're trying to achieve from our new threads, it's apparent that we're trying to calculate a result to return, whilst allowing for the possibility that the code might throw an exception. This is *precisely* what the combination of **std::packaged_task** and **std::unique_future** are designed for. If we rearrange our code to use **std::packaged_task** then we end up with listing 8.3.

Listing 8.51: A parallel version of std::accumulate using std::packaged_task

```
template<typename Iterator,typename T>
struct accumulate_block
{
    T operator()(Iterator first,Iterator last)                         #1
    {
        return std::accumulate(first,last,T());                        #2
    }
};

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<std::unique_future<T> > futures(num_threads-1);        #3
    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);
        std::packaged_task<T(Iterator,Iterator)> task(                 #4
            accumulate_block<Iterator,T>());
        futures[i]=std::move(task.get_future());                       #5
        threads[i]=std::thread(std::move(task),block_start,block_end); #6
        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);                #7

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

    T result=init;                                                     #8
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();                                      #9
    }
    result += last_result;                                            #10
    return result;
}
```

Cueballs in Code and Text

The first change is that the function call operator of **accumulate_block** now returns the result directly, rather than taking a reference to somewhere to store it (#1) — we're using **std::packaged_task** and **std::unique_future** for the exception safety, so we can use it to transfer the result too. This does require that we explicitly pass in a default-constructed **T** in the call to **std::accumulate** (#2), rather than reusing the supplied **result** value, but that's a minor change.

The next change is that rather than having a vector of results, we have a vector of **futures** (#3) to store a **std::unique_future<T>** for each spawned thread. In the thread-spawning loop, we first create a task for **accumulate_block** (#4) — **std::packaged_task<T(Iterator,Iterator)>** declares a task that takes two **Iterator**s and returns a **T**, which is what our function does. We then get the future for that task (#5), and run that task on a new thread, passing in the start and end of the block to process (#6).

When the task runs, then the result will be captured in the future, as will any exception thrown.

Since we've been using futures, we don't have a result array so we must store the result from the final block in a variable (#7) rather than in a slot in the array. We also can't use **std::accumulate** to total the results, as we've got to get the values out of the futures. Instead, we can just code a simple loop, starting with the supplied initial value (#8), and adding in the result from each future (#9). If the corresponding task threw an exception this will have been captured in the future, and will now be thrown again by the call to **get()**. Finally, we add the result from the last block (#10) before returning the overall result to the caller.

So, that's removed one of the potential problems: exceptions thrown *in* the worker threads are re-thrown in the main thread. Of course, if more than one of the worker threads throws an exception, only one will be propagated, but that's not too big a deal. If it really matters you can use something like **std::nested_exception** to capture all the exceptions, and throw that instead.

The remaining problem is the leaking threads if an exception is thrown between when we spawn the first thread, and when we've joined with them all. The simplest solution is just to catch any exceptions, join with the threads that are still **joinable()** and rethrow the exception:

```
    try
    {
        for(unsigned long i=0;i<(num_threads-1);++i)
        {
            // ... as before
        }
        T last_result=accumulate_block()(block_start,last);

        std::for_each(threads.begin(),threads.end(),
            std::mem_fn(&std::thread::join));
    }
    catch(...)
    {
        for(unsigned long i=0;i<(num_thread-1);++i)
        {
            if(threads[i].joinable())
                thread[i].join();
        }
        throw;
    }
```

Now, this works: all the threads will be joined, however the code leaves the block. However, **try**-**catch** blocks are ugly, and we've got duplicate code: we're joining the threads both in the “normal” control flow, **and** in the **catch** block. Duplicate code is rarely a good thing, as it means more places to change. Instead, let's extract this out into the destructor of an object: it is after all the idiomatic way of cleaning up resources in C++. Here's our class:

```
class join_threads
{
    std::vector<std::thread>& threads;
public:
    explicit join_threads(std::vector<std::thread>& threads_):
        threads(threads_)
    {}
    ~join_threads()
    {
        for(unsigned long i=0;i<threads.size();++i)
        {
            if(threads[i].joinable())
                threads[i].join();
        }
    }
};
```

We can then simplify our code to listing 8.4.

Listing 8.52: An exception-safe parallel version of `std::accumulate`

```
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<std::unique_future<T> > futures(num_threads-1);
    std::vector<std::thread> threads(num_threads-1);
    join_threads joiner(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);
        std::packaged_task<T(Iterator,Iterator)> task(
            accumulate_block<Iterator,T>());
        futures[i]=std::move(task.get_future());
        threads[i]=std::thread(std::move(task),block_start,block_end);
        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        result+=futures[i].get();                                      #2
    }
    result += last_result;
    return result;
}
```

Cueballs in Code and Text

Once we've created our container of threads, we create an instance of our new class (#1) to join with all the threads on exit. We can then remove our explicit join loop, safe in the knowledge that the threads will be joined however the function exits. Note that the calls to **`futures[i].get()`** (#2) will block until the results are ready, so we don't need to have explicitly joined with the threads at this point. This is unlike the original from listing 8.2 where we needed to have joined with the threads to ensure that the **`results`** vector was correctly populated. Not only do we get exception safe code, but our function is actually shorter because we've extracted the join code into our new (reusable) class.

What other considerations to we need to take into account when designing concurrent code? Let's take a look at *scalability* — how much does the performance improve if you move your code to a system with more processors?

8.4.2 Scalability and Amdahl's Law

Scalability is all about ensuring that your application can take advantage of additional processors in the system it is running on. At one extreme you've got a single-threaded application which is completely unscaleable — even if you add 100 processors to your system then the performance will remain unchanged. At the other extreme you have something like the SETI@Home [SETI@Home] project which is designed to take advantage of thousands of additional processors (in the form of individual computers added to the network by users) as and when they become available.

For any given multi-threaded program, the number of threads that are performing useful work will vary as the program runs. Even if every thread is doing useful work for the entirety of its existence, the application may initially only have one thread, which will then have the task of spawning all the others. However, even that is a highly unlikely scenario — threads often spend time waiting for each other, or waiting for I/O operations to complete.

Every time one thread has to wait for something (whatever that something is), unless there is another thread ready to take its place on the processor, you have a processor sitting idle that could be doing useful work.

A simplified way of looking at this is to divide the program into “serial” sections where only one thread is doing any useful work, and “parallel” sections where all the available processors are doing useful work. If run your application on a system with more processors then the “parallel” sections will theoretically be able to complete quicker, as the work can be divided between more processors, whilst the “serial” sections will remain serial. Under such a simplified set of assumptions, you can therefore estimate the potential performance gain to be achieved by increasing the number of processors: if the “serial” sections constitute a fraction f_s of the program, then the performance gain *P* from using *N* processors can be estimated as:

$$P=\frac{1}{f_s \quad \frac{1-f_s}{N}}$$

This is *Amdahl's Law*, which is often cited when talking about the performance of concurrent code. However, this paints a very naïve picture, since tasks are rarely infinitely divisible in the way that would be required for the equation to hold, and it is also rare for everything to be CPU-bound in the way that is assumed — as we've just seen, threads may wait for many things whilst executing.

One thing that is clear from Amdahl's law is that when you're using concurrency for performance, it is worth looking at the overall design of the application to maximize the potential for concurrency and ensure that there is always useful work for the processors to be doing. If you can reduce the size of the “serial” sections, or reduce the potential for threads to wait, then you can improve the potential for performance gains on systems with more processors. Alternatively, if you can provide more data for the system to process, and thus keep the parallel sections primed with work then you can reduce the serial fraction, and increase the performance gain *P*.

Essentially, scalability is about two things: **reducing the time it takes to perform an action** or **increasing the amount of data that can be processed in a given time** as more processors are added. Sometimes these are equivalent (you can process more data if each element is processed faster), but not always. Before choosing the techniques to use for dividing work between threads, it is important to identify which of these aspects of scalability are important to you.

I mentioned at the beginning of this section that threads don't always have useful work to do — sometimes they have to wait either for other threads, or for I/O to complete, or something else. If we give the system something useful to do during this wait, we can effectively “hide” the waiting.

8.4.3 Hiding Latency with Multiple Threads

For lots of the discussions of the performance of multi-threaded code, we've been assuming that the threads are running “flat out”, and always have useful work to do when they are actually running on a processor. This is of course not true: in application code threads frequently block waiting for something. For example, they may be waiting for some I/O to complete, waiting to acquire a mutex, waiting for another thread to complete some operation and notify a condition variable or populate a future, or even just sleeping for a period of time.

Whatever the reason for the waits, if we only have as many threads as there are physical processing units in the system then blocked threads means we are wasting CPU time: the processor that would otherwise be running a blocked thread is instead doing nothing. Consequently, if we know that one of our threads is likely to spend a considerable portion of its time waiting around then we can make use of that spare CPU time by running one or more additional threads.

Consider a virus scanner application which divides the work across threads using a pipeline. The first thread searches the file system for files to check, and puts them on a queue. Meanwhile another thread take file names from the queue, load the files and scan them for viruses. We know that the thread searching the file system for files to scan is definitely going to be I/O bound, so we make use of the “spare” CPU time by running an additional scanning thread. We would then have one file-searching thread, and then as many scanning threads as there are physical cores or processors in the system. Since the scanning thread may also have to read significant portions of the files off the disk in order to scan them, it might make sense to have even more scanning threads, but at some point there will be too many threads, and the system will slow down again as it spends more and more time task switching, as described in section 8.2.5.

As ever, this is an optimization, so it is important to measure performance before and after any change in the number of threads: the optimal number of threads will by highly dependent on the nature of the work being done, and the percentage of time the thread spends waiting.

Depending on the application, it might be possible to use up this spare CPU time without running additional threads. For example, if a thread is blocked waiting for an I/O operation to complete, it might make sense to use asynchronous I/O if that is available, and then the thread can perform other useful work whilst the I/O is performed in the background. In other cases, if a thread is waiting for another thread to perform an operation then rather than blocking, the waiting thread might be able to perform that operation itself instead, as we saw with the lock-free queue in chapter 7. In an extreme case, if a thread is waiting for a task to be completed and that task has not yet been started by any thread, then the waiting thread might perform the task in entirety itself or another task that is incomplete. We saw an example of this in listing 8.1, where the sort function repeatedly tries to sort outstanding chunks so long as the chunks it needs are not yet sorted.

Rather than adding threads to ensure that all available processors are being used, sometimes it pays to add threads to ensure that external events are handled in a timely manner — to increase the *responsiveness* of the system.

8.4.4 Improving Responsiveness with Concurrency

Most modern graphical user interface frameworks are *event driven* — the user performs actions on the user interface by pressing keys or moving the mouse, which generate a series of events or messages which the application then handles. The system may also generate messages or events on its own. In order to ensure that all events and messages are correctly handled, the application therefore typically has an event loop that looks like this:

```
while(true)
{
    event_data event=get_event();
    if(event.type==quit)
        break;
    process(event);
}
```

Obviously, the details of the API will vary, but the structure is generally the same: wait for an event, do whatever processing is necessary to handle it, and then wait for the next one. If you have a single-threaded application, this can make long-running tasks hard to write, as described in section 8.1.3. In order to ensure that user input is handled in a timely manner, **get_event()** and **process()** must be called reasonable frequently, whatever the application is doing. This means that either the task must periodically suspend itself and return control to the event loop, or the **get_event()/process()** code must be called from within the code at convenient points. Either option complicates the implementation of the task.

By separating the concerns with concurrency, we can put the lengthy task on a whole new thread, and leave a dedicated GUI thread to process the events. The threads can then communicate through simple mechanisms rather than having to somehow mix the event handling code into the task code. Listing 8.5 shows a simple outline for such a separation.

Listing 8.53: Separating GUI thread from task thread

```
std::thread task_thread;
std::atomic<bool> task_cancelled(false);

void gui_thread()
{
    while(true)
    {
        event_data event=get_event();
        if(event.type==quit)
            break;
        process(event);
    }
}

void task()
{
    while(!task_complete() && !task_cancelled)
    {
        do_next_operation();
    }
    if(task_cancelled)
    {
        perform_cleanup();
    }
    else
    {
        post_gui_event(task_complete);
    }
}

void process(event_data const& event)
{
    switch(event.type)
    {
    case start_task:
        task_cancelled=false;
        task_thread=std::thread(task);
        break;
    case stop_task:
        task_cancelled=true;
        task_thread.join();
        break;
    case task_complete:
        task_thread.join();
        display_results();
        break;
    default:
        //...
    }
}
```

By separating the concerns in this way, the user thread is always able to respond to the events in a timely fashion, even if the task takes a long time. This *responsiveness* is often key to the user experience when using an application: applications that completely "lock up" whenever a particular operation is being performed (whatever that may be) are really inconvenient to use. By providing a dedicated event handling thread the GUI can handle GUI-specific messages (such as resizing or repainting the window) without interrupting the execution of the time-consuming processing, whilst still passing on the relevant messages where they *do* affect the long-running task.

So far in this chapter we've had a thorough look at the issues that need to be considered when designing concurrent code. Taken as a whole these can be quite overwhelming, but as you get used to working with your "multithreaded programming hat" on most of them will become second nature. If these considerations are new to you, then hopefully they will become clearer as we look at how they impact some concrete examples of multithreaded code.

8.5 Designing Concurrent Code in Practice

When designing concurrent code for a particular task then the extent to which you will need to consider each of the issues described above will depend on the task. To demonstrate how they apply, we're going to look at the implementation of parallel versions of three functions from the C++ Standard Library. This will give us a familiar basis on which to build, whilst providing a platform for looking at the issues. As a bonus, we'll also have usable implementations of the functions, which could be used to help with parallelizing a larger task.

I've primarily selected these implementations to demonstrate particular techniques rather than to be state of the art implementations — more advanced implementations that make better use of the available hardware concurrency may be found in the academic literature on parallel algorithms, or in specialist multithreading libraries such as Intel's Threading Building Blocks [TBB].

The simplest parallel algorithm conceptually is a parallel version of **std::for_each**, so we'll start with that.

8.5.1 A parallel implementation of std::for_each

std::for_each is really simple in concept — it calls a user-supplied function on every element in a range in turn. The big difference between a parallel implementation and the sequential **std::for_each** is the order of the function calls — **std::for_each** calls the function with the first element in the range, then the second, and so on, whereas with a parallel implementation there is no guarantee which order the elements will be processed, and they may (indeed we hope they *will*) be processed concurrently.

To implement a parallel version of this, we just need to divide the range into sets of elements to process on each thread. We know the number of elements in advance, so we can divide the data before processing begins (section 8.1.1). We also know that the elements can be processed entirely independently, so we can use contiguous blocks to avoid false sharing (section 8.2.3).

This algorithm is similar in concept to the parallel version of **std::accumulate** described in section 8.4.1, but rather than computing the sum of each element we merely have to apply the specified function. Though you might imagine this would greatly simplify the code, as there is no result to return, if we wish to pass on exceptions to the caller then we still need to use the **std::packaged_task** and **std::unique_future** mechanisms to transfer the exception between threads. A sample implementation is shown in listing 8.6.

Listing 8.54: A parallel version of std::for_each

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first,Iterator last,Func f)
{
    unsigned long const length=std::distance(first,last);

    if(!length)
        return;

    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<std::unique_future<void> > futures(num_threads-1);    #1
    std::vector<std::thread> threads(num_threads-1);
    join_threads joiner(threads);

    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);
        std::packaged_task<void(void)> task(                           #2
            [=]()
            {
                std::for_each(block_start,block_end,f);
            });
        futures[i]=std::move(task.get_future());
        threads[i]=std::thread(std::move(task));                    #3
        block_start=block_end;
    }
    std::for_each(block_start,last,f);
    for(unsigned long i=0;i<(num_threads-1);++i)
    {
        futures[i].get();                                           #4
    }
}
```

Cueballs in Code and Text

The basic structure of the code is identical to that of listing 8.4, which is unsurprising. The key difference is that the **futures** vector stores **std::unique_future<void>** (#1) because the worker threads do not return a value, and a simple lambda function which invokes the function **f** on the range from **block_start** to **block_end** is used for the task (#2). This avoids having to pass the range into the thread constructor (#3). Since the worker threads don't return a value, the calls to **futures[i].get()** (#4) just provide a means of retrieving any exceptions thrown on the worker threads — if we didn't wish to pass on the exceptions, this could be omitted.

Let's move on from algorithms that must perform the same operation on each element (of which there are several — **std::count** and **std::replace** spring to mind for starters) to a slightly more complicated examples in the shape of **std::find**.

8.5.2 A parallel implementation of std::find

std::find is a useful algorithm to consider next, because it is one of several algorithms that can complete without every element having been processed. For example, if the first element in the range matches the search criterion, then there is no need to examine any other elements. As we shall see shortly, this is an important property for performance, and has very direct consequences for the design of the parallel implementation. It is a particular example of how data access patterns can affect the design of our code (section 8.3.2). Other algorithms in this category include **std::equal**, and **std::any_of**.

If you were searching for an old photograph through the boxes of keepsakes in your loft with your wife or partner, you wouldn't let them continue searching if you found the photograph. Instead, you'd let them know you'd found the photograph (perhaps by shouting “found it!”) so that they could stop searching and move on to something else. The nature of many algorithms requires that they process every element, so they have no equivalent to shouting “found it!”. For algorithms such as **std::find** the ability to complete “early” is an important property, and not something to squander. We therefore need to design our code to make use of it — interrupt the other tasks in some way when the answer is known, so that the code doesn't have to wait for the other worker threads to process the remaining elements.

If we do not interrupt the other threads then the serial version may well out-perform our parallel implementation, because the serial algorithm can just stop searching and return once a match is found. If, for example, the system can support four concurrent threads, then each thread will have to examine one quarter of the elements in the range and our naïve parallel implementation would thus take approximately one quarter of the time as a single thread would take to check every element. If the matching element lies in the first quarter of the range then the sequential algorithm will return first, because it does not need to check the remainder of the elements.

One way in which we can interrupt the other threads is by making use of an atomic variable as a flag and checking the flag after processing every element — if the flag is set then one of the other threads has found a match so we can cease processing and return. By interrupting the threads in this way we preserve the property that we don't have to process every value, and thus improve the performance compared to the serial version in more circumstances.

Now, we have two choices on how to return the values, and how to propagate any exceptions. We can either use an array of futures and use **std::packaged_task** for transferring the values and exceptions, and then process the results back in the main thread, or we can use **std::promise** to set the final result directly from the worker threads. It all depends on how we wish to handle exceptions from the worker threads. If we want to stop on the first exception (even if we haven't processed all elements) then we can use **std::promise** to set both the value or the exception. On the other hand, if we want to allow the other workers to keep searching then we can use **std::packaged_task**, store all the exceptions and then rethrow one of them if a match is not found.

In this case I've opted to use **std::promise** as the behaviour matches that of **std::find** more closely. One thing to watch out for here is the case that the element being searched for is not in the supplied range. We therefore need to wait for all the threads to finish *before* getting the result from the future — if we just block on the future, we'll be waiting forever if the value is not there. The result is listing 8.7.

Listing 8.55: An implementation of a parallel find algorithm

```
template<typename Iterator,typename MatchType>
Iterator parallel_find(Iterator first,Iterator last,MatchType match)
{
    struct find_element                                             #1
    {
        void operator()(Iterator begin,Iterator end,
                        MatchType match,
                        std::promise<Iterator>* result,
                        std::atomic<bool>* done_flag)
        {
            try
            {
                for(;(begin!=end) && !done_flag->load();++begin)       #2
                {
                    if(*begin==match)
                    {
                        result->set_value(begin);                      #3
                        done_flag->store(true);                        #4
                        return;
                    }
                }
            }
            catch(...)                                                 #5
            {
                try
                {
                    result->set_exception(std::current_exception());   #6
                    done_flag->store(true);
                }
                catch(...)                                             #7
                {}
            }
        }
    };

    unsigned long const length=std::distance(first,last);

    if(!length)
        return last;

    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::promise<Iterator> result;                                     #8
    std::atomic<bool> done_flag(false);                                #9
    std::vector<std::thread> threads(num_threads-1);
```

```
        {                                                                      #12
            join_threads joiner(threads);

            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(find_element(),                         #10
                                       block_start,block_end,match,
                                       &result,&done_flag);
                block_start=block_end;
            }
            find_element()(block_start,last,match,&result,&done_flag);     #11
        }
        if(!done_flag.load())                                                  #13
        {
            return last;
        }
        return result.get_future().get();                                      #14
    }
```

Cueballs in code and text

The main body of listing 8.7 is very similar to the previous examples. This time, the work is done in the function call operator of the local **find_element** class (#1). This loops through the elements in the block it's been given, checking the flag at each step (#2). If a match is found then it sets the final result value in the promise (#3), and then sets the **done_flag** (#4) before returning.

If an exception is thrown, then this is caught by the catch-all handler (#5), and we try and store the exception in the promise (#6) before setting the **done_flag**. Setting the value on the promise might throw if the promise is already set, so we catch and discard any exceptions that happen here (#7).

This means that if a thread calling **find_element** either finds a match or throws an exception then all other threads will see **done_flag** set, and stop. If multiple threads find a match or throw at the same time, then they will race to set the result in the promise, but this is a benign race condition: whichever succeeds is therefore nominally “first”, and is therefore an acceptable result.

Back in the main **parallel_find** function itself, we've got the promise (#8) and flag (#9) used to stop the search, which are passed in to the new threads along with the range to search (#10). The main thread also uses **find_element** to search the remaining elements (#11). As already mentioned, we need to wait for all threads to finish before we check the result, because there might not be *any* matching elements. We do this by enclosing the thread launching-and-joining code in a block (#12), so all threads are joined when we check the flag to see if a match was found (#13). If a match was found, then we can get the result or throw the stored exception by calling **`get()`** on the **`std::unique_future<Iterator>`** we can get from the promise.

A key feature that this algorithm shares with the other parallel algorithms we've seen is that there is no longer the guarantee that items are processed in sequence that you get from **`std::find`**. This is essential if we're going to parallelize the algorithm — you can't process elements concurrently if the order matters. If the elements are independent, then it doesn't matter for things like **`parallel_for_each`**, but it means that our **`parallel_find`** might return an element towards the end of the range even when there is a match towards the beginning, which might be surprising if we're not expecting it.

OK, so we've managed to parallelize **`std::find`**. As I stated at the beginning of this section, there are other similar algorithms that can complete without processing every data element, and the same techniques can be used for those. We will also look further at the issue of interrupting threads in chapter 9.

To complete our trio of examples, we'll go in a different direction and take a look at **`std::partial_sum`**. This algorithm doesn't get a lot of press, but it's an interesting algorithm to parallelize and highlights some additional design choices.

8.5.3 A parallel implementation of `std::partial_sum`

`std::partial_sum` calculates the running totals in a range, so each element is replaced by the sum of that element and all the elements prior to it in the original sequence. Thus the sequence 1, 2, 3, 4, 5 becomes 1, (1+2)=3, (1+2+3)=6, (1+2+3+4)=10, (1+2+3+4+5)=15. This is interesting to parallelize because you can't just divide the range into chunks and do each chunk independently — the initial value of the first element needs to be added to every other element, for example.

One approach to determining the partial sum of a range is to do the partial sum of individual chunks, and then add the resulting value of the last element in the first chunk onto the elements in the next chunk, and so forth. If we've got the elements 1, 2, 3, 4, 5, 6, 7, 8, 9 and we're splitting into 3 chunks we get {1, 3, 6}, {4, 9, 15}, {7, 15, 24} in the first instance. If we then add 6 (the sum for the last element in the first chunk) onto the elements in the second chunk we get {1, 3, 6}, {10, 15, 21}, {7, 15, 24}. Then, we add the last element of the second chunk (21) onto the elements in the third and final chunk to give us the final result: {1, 3, 6}, {10, 15, 21}, {28, 36, 55}.

As well as the original division into chunks, the addition of the partial sum from the previous block can also be parallelized — if the last element of each block is updated first, then the remaining elements in a block can be updated by one thread whilst a second thread updates the next block, and so forth. This works well when there are many more elements in the list than processing cores, as each core has a reasonable number of elements to process at each stage.

If you've got a lot of processing cores (as many, or more than the number of elements), this doesn't work so well. If you divide the work amongst the processors, you end up working in pairs of elements at the first step. Under these conditions, this forward propagation of results means that a lot of processors are left waiting, so we need to find some work for them to do. We can then take a different approach to the problem — rather than doing the full forward propagation of the sums from one chunk to the next, we do partial propagation: first sum adjacent elements as before, but then add those sums to those 2 elements away, then add the next set of results to the results from 4 elements away, and so forth. If we start with the same initial 9 elements, we get 1, 3, 5, 7, 9, 11, 13, 15, 17 after the first round, which gives us the final results for the first two elements. After the second we then have 1, 3, 6, 10, 14, 18, 22, 26, 30, which is correct for the first four elements. After round 3 we have 1, 3, 6, 10, 15, 21, 28, 36, 44 which is correct for the first 8 elements, and finally after round 4 we have 1, 3, 6, 10, 15, 21, 28, 36, 45 which is the final answer. Though there's more total steps than in the first approach, there is greater scope for parallelism if there are a lot of processors: each processor can update one entry with each step.

Overall, the second approach takes $\log_2(N)$ steps of around *N* operations (one per processor), where *N* is the number of elements in the list. This compares to the first algorithm where each thread has to perform N/k operations for the initial partial sum of the chunk allocated to it, and then a further N/k operations to do the forward propagation, where *k* is the number of threads. Thus the first approach is O(N), whereas the second is O(N log(N)) in terms of total number of operations. However, if we have as many processors as list elements, then the second approach only requires log(N) operations per processor, whereas the first essentially serializes the operations when *k* gets large, due to the forward propagation. For small numbers of processing units, the first approach will therefore finish faster, whereas for massively-parallel systems the second will finish faster. This is an extreme example of the issues discussed in section 8.2.1.

Anyway, efficiency issues aside, let's look at some code. Listing 8.8 shows the first approach.

Listing 8.56: Calculating partial sums in parallel by dividing the problem

```
template<typename Iterator>
void parallel_partial_sum(Iterator first,Iterator last)
{
    typedef typename Iterator::value_type value_type;

    struct process_chunk                                              #12
```

```
        {
            void operator()(Iterator begin,Iterator last,
                            std::unique_future<value_type>*
    previous_end_value,
                            std::promise<value_type>* end_value)
            {
                try
                {
                    Iterator end=last;
                    ++end;
                    std::partial_sum(begin,end);                         #13
                    if(previous_end_value)                               #14
                    {
                        value_type& addend=previous_end_value->get();    #15
                        *last+=addend;                                   #16
                        if(end_value)
                        {
                            end_value->set_value(*last);                 #17
                        }
                        std::for_each(begin,last,[addend](value_type& item)#18
                                      {
                                          item+=addend;
                                      });
                    }
                    else if(end_value)
                    {
                        end_value->set_value(*last);                     #19
                    }
                }
                catch(...)                                               #20
                {
                    if(end_value)
                    {
                        end_value->set_exception(std::current_exception());#21
                    }
                    else
                    {
                        throw;                                           #22
                    }
                }
            }
        };

        unsigned long const length=std::distance(first,last);

        if(!length)
            return last;

        unsigned long const min_per_thread=25;                            #1
        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;

        typedef typename Iterator::value_type value_type;

        std::vector<std::thread> threads(num_threads-1);                  #2
        std::vector<std::promise<value_type> > end_values(num_threads-1);  #3
        std::vector<std::unique_future<value_type> > previous_end_values;  #4
        previous_end_values.reserve(num_threads-1);                        #5
        join_threads joiner(threads);

        Iterator block_start=first;
        for(unsigned long i=0;i<(num_threads-1);++i)
        {
            Iterator block_last=block_start;
            std::advance(block_last,block_size-1);                         #6
            threads[i]=std::thread(process_chunk(),                        #7
                                   block_start,block_last,
                                   (i!=0)?&previous_end_values[i-1]:0,
                                   &end_values[i]);
            block_start=block_last;
            ++block_start;                                                 #8
            previous_end_values.push_back(end_values[i].get_future());     #9
        }
        Iterator final_element=block_start;
        std::advance(final_element,std::distance(block_start,last)-1);     #10
        process_chunk()(block_start,final_element,                         #11
                        (num_threads>1)?&previous_end_values.back():0,
                        0);
    }
```

Cueballs in code and text

In this instance, the general structure is the same as with the previous algorithms, dividing the problem into chunks, with a minimum chunk size per thread (#1). In this case, as well as the vector of threads (#2), we've got a vector of promises (#3) which is used to store the value of the last element in the chunk, and a vector of futures (#4), which is used to retrieve the last value from the previous chunk. **`std::unique_future`** is not default-constructible, so we have to reserve the space (#5) rather than pre-allocate all the elements.

The main loop is the same as before, except this time we actually want the iterator that *points to* the last element in each block, rather than being the usual one-past-the-end (#6), so that we can do the forward-propagation of the last element in each range. The actual processing is done in the **process_chunk** function object which we'll look at shortly; the start and end iterators for this chunk are passed in as arguments alongside the future for the end value of the previous range (if any), and the promise to hold the end value of this range (#7).

After we've spawned the thread, we can update the block start, remembering to advance it past that last element (#8), and store the future for the last value in the current chunk into the vector of futures so it will be picked up next time round the loop (#9).

Before we process the final chunk, we need to get an iterator for the last element (#10), which we can pass in to **process_chunk** (#11). **std::partial_sum** doesn't return a value so we don't need to do anything once the final chunk has been processed — the operation is complete once all the threads have finished.

OK, now it's time to look at the **process_chunk** function object that actually does all the work (#12). We start by calling **std::partial_sum** for the entire chunk, including the final element (#13), but then we need to know if we're the first chunk or not (#14). If we are *not* the first chunk then there was a **previous_end_value** from the previous chunk, so we need to wait for that (#15). In order to maximize the parallelism of the algorithm we then update the last element first (#16), so we can pass the value on to the next chunk (if there is one) (#17). Once we've done that, we can just use **std::for_each** and a simple lambda function (#18) to update all the remaining elements in the range.

If there was *not* a **previous_end_value**, then we're the first chunk, so we can just update the **end_value** for the next chunk (again, if there is one — we might be the one and only chunk) (#19).

Finally, if any of the operations threw an exception, we catch it (#20), and store it in the promise (#21) so it will propagate to the next chunk when it tries to get the previous end value (#15). This will propagate all exceptions into the final chunk, which then just rethrows (#22), because we know we're running on the main thread.

With the block-based, forward-propagation approach out the way, let's take a look at the second approach to computing the partial sums of a range.

Implementing the incremental pairwise algorithm for partial sums

This second approach to calculating the partial sums by adding elements increasingly further away works best where your processors can execute the additions in lock-step — in this case, no further synchronization is necessary because all the intermediate results can be propagated directly to the next processor that needs them. However, in practice we rarely have such systems to work with except for those cases where a single processor can execute the same instruction across a small number of data elements simultaneously with so-called *Single-Instruction-Multiple-Data* (SIMD) instructions. Therefore, we must design our code for the general case, and explicitly synchronize the threads at each step.

One way to do this is to use a *barrier* — a synchronization mechanism that causes threads to wait until the required number of threads has reached the barrier. Once all the threads have reached the barrier, they are all unblocked, and may proceed. The C++0x thread library doesn't offer such a facility directly, so we have to design one ourselves.

Imagine a roller-coaster at the fairground. If there's a reasonable number of people queuing, the fairground staff will ensure that every seat is filled before the roller coaster leaves the on-off platform. A barrier works the same way: you specify up front the number of “seats”, and threads have to wait until all the “seats” are filled. Once there's enough waiting threads then they can all proceed, the barrier is reset and starts waiting for the next batch of threads. Often, such a construct is used in a loop, where it is the same threads that come around and wait next time. The idea is to keep the threads in lock-step, so one thread doesn't run away in front of the others, and get out of step — for an algorithm such as this one that would be disastrous, because the runaway thread would potentially modify data that was still being used by other threads, or use data that hadn't been correctly updated yet.

Anyway, listing 8.9 shows a *really simple* implementation of a barrier.

Listing 8.57: A really simple barrier class.

```
class barrier
{
    unsigned const count;
    std::atomic<unsigned> spaces;
    std::atomic<unsigned> generation;
public:
    explicit barrier(unsigned count_):                              #1
        count(count_),spaces(count),generation(0)
    {}
    void wait()
    {
        unsigned const my_generation=generation;                    #6
        if(!--spaces)                                               #2
        {
            spaces=count;                                           #3
            ++generation;                                           #4
        }
        else
        {
            while(generation==my_generation)                        #5
                std::this_thread::yield();                          #7
        }
        }
    };
```

Cueballs in code and text

With this implementation, you construct a **barrier** with the number of “seats” (#1), which is stored in the **count** variable. Initially, the number of **spaces** at the barrier is equal to this count. As each thread waits, the number of **spaces** is decremented (#2). When it reaches zero, then the number of spaces is reset back to **count**, and the **generation** is increased to signal to the other threads that they can continue (#4). If the number of free **spaces** did not reach zero, then we have to wait. This implementation uses a simple spin lock (#5) checking the **generation** against the value we retrieved at the beginning of **wait()** (#6). Since the **generation** is only updated when all the threads have reached the barrier (#4), we **yield()** whilst waiting (#7), so the waiting thread doesn't hog the CPU in a busy wait.

When I said this implementation was simple, I meant it: it uses a spin wait, so it's not ideal for cases where threads are likely to be waiting a long time, and it doesn't work if there's more than **count** threads that can potentially call **wait()** at any one time. If you need to handle either of those scenarios then a more robust (but more complex) implementation must be used instead. I've also stuck to sequentially consistent operations on the atomic variables, as that makes everything easier to reason about, but you could potentially relax some of the ordering constraints.

Anyway, this is just what we need here — we have a fixed number of threads, that need to run in a lock-step loop. Well, it's *almost* a fixed number of threads — as you may remember, the items at the beginning of the list acquire their final values after a couple of steps. This means that either we have to keep those threads looping until the entire range has been processed, or we need to allow our barrier to handle threads dropping out, and thus decreasing **count**. I've opted to go for the latter option, as it avoids having threads doing unnecessary work just looping until the final step is done.

This means we've got to change **count** to be an atomic variable, so we can update it from multiple threads without external synchronization.

```
    std::atomic<unsigned> count;
```

The initialization remains the same, but now we've got to explicitly **load()** from **count** when we reset the number of **spaces**:

```
    spaces=count.load();
```

That's all the changes that we need on the **wait()** front, now we need a new member function to decrement **count**. Let's call it **done_waiting()**, as a thread is declaring that it is done with waiting:

```
    void done_waiting()
    {
        --count;                                                          #1
        if(!--spaces)                                                #2
        {
            spaces=count.load();                                     #3
            ++generation;
        }
    }
```

Cueballs in code and text

The first thing we do is decrement the **count** (#1), so that the next time **spaces** is reset it reflects the new lower number of waiting threads. Then we need to decrease the number of free **spaces** (#2). If we don't do this, then the other threads will be waiting forever, since **spaces** was initialized to the old, larger, value. Of course, if we are the last thread through on this batch, then we need to reset the counter, and increase the **generation** (#3), just as we do in **wait()**. The key difference here is that if we're not the last thread in the batch, then we don't have to wait — we are done with waiting after all!

We're now ready to write our second implementation of partial sum — at each step, every thread calls **wait()** on the barrier to ensure the threads step through together, and once each thread is done it calls **done_waiting()** on the barrier to decrement the count. If we use a second buffer alongside the original range the barrier provides all the synchronization we need — at each step the threads read from either the original range or the buffer, and write the new value to the corresponding element of the other. If the threads read from the original range on one step, they read from the buffer on the next, and vice versa. This ensures there are no race condition between the reads and writes by separate threads. Of course, once a thread has finished looping, it must ensure that the correct final value has been written to the original range. Listing 8.10 puts this all together.

Listing 8.58: A parallel implementation of partial_sum by pairwise updates

```
    struct barrier
    {
        std::atomic<unsigned> count;
        std::atomic<unsigned> spaces;
        std::atomic<unsigned> generation;

        barrier(unsigned count_):
            count(count_),spaces(count_),generation(0)
        {}

        void wait()
        {
            unsigned const gen=generation.load();
            if(!--spaces)
            {
                spaces=count.load();
                ++generation;
            }
            else
            {
                while(generation.load()==gen)
                {
                    std::this_thread::yield();
                }
            }
        }

        void done_waiting()
        {
            --count;
            if(!--spaces)
            {
                spaces=count.load();
                ++generation;
            }
        }
    };

    template<typename Iterator>
    void parallel_partial_sum(Iterator first,Iterator last)
    {
        typedef typename Iterator::value_type value_type;

        struct process_element                                          #1
        {
            void operator()(Iterator first,Iterator last,
                            std::vector<value_type>& buffer,
                            unsigned i,barrier& b)
            {
                value_type& ith_element=*(first+i);
                bool update_source=false;

                for(unsigned step=0,stride=1;stride<=i;++step,stride*=2)
                {
                    value_type const& source=(step%2)?                  #5
                        buffer[i]:ith_element;
                    value_type& dest=(step%2)?
                        ith_element:buffer[i];
                    value_type const& addend=(step%2)?                  #6
                        buffer[i-stride]:*(first+i-stride);

                    dest=source+addend;                                 #7
                    update_source=!(step%2);
                    b.wait();                                           #8
                }
                if(update_source)                                          #9
                {
                    ith_element=buffer[i];
                }
                b.done_waiting();                                          #10
            }
        };

        unsigned long const length=std::distance(first,last);

        if(length<=1)
            return;

        std::vector<value_type> buffer(length);
        barrier b(length);

        std::vector<std::thread> threads(length-1);                 #3
        join_threads joiner(threads);

        Iterator block_start=first;
        for(unsigned long i=0;i<(length-1);++i)
        {
            threads[i]=std::thread(process_element(),first,last,    #2
                                   std::ref(buffer),i,std::ref(b));
        }
        process_element()(first,last,buffer,length-1,b);           #4
    }
```

Cueballs in code and text

The overall structure of this code is probably getting familiar by now — we've got a class with a function call operator (**process_element**) for doing the work (#1), which we run on a bunch of threads (#2) stored in a vector (#3), and which we also call from the main thread (#4). The key difference this time is that the number of threads is dependent on the number of items in the list rather than on **std::thread::hardware_concurrency**. As I said already, unless you're on a massively parallel machine where threads are cheap, this is probably a bad idea, but it shows the overall structure. It would be possible to have fewer threads, with each thread handling several values from the source range, but there will come a point where there are sufficiently few threads that this is less efficient than the forward-propagation algorithm.

Anyway, the key work is done in the function call operator of **process_element** — each step we either take the *i*-th element from the original range or the *i*-th element from the buffer (#5), and add it to the value **stride** elements prior (#6), storing it in the buffer if we started in the original range, or back in the original range if we started in the buffer (#7). We then wait on the barrier (#8) before starting the next step. We're done when the **stride** takes us off the start of the range, in which case we need to update the element in the original range if our final result was stored in the buffer (#9). Finally, we tell the barrier that we're **`done_waiting()`** (#10).

Note that this solution is not exception-safe — if an exception is thrown in **`process_element`** on one of the worker threads then it will terminate the application. This could be dealt with by using a **`std::promise`** to store the exception, as we did for the **`parallel_find`** implementation from listing 8.7, or even just using a **`std::exception_ptr`** protected by a mutex.

That concludes our three examples. Hopefully, they have helped to crystallize some of the design considerations highlighted in sections 8.1, 8.2, 8.3 and 8.4, and demonstrate how these techniques can be brought to bear in real code.

8.6 Summary

We've covered quite a lot of ground in this chapter. We started with various techniques for dividing work between threads such as dividing the data beforehand, or using a number of threads to form a pipeline. We then looked at the issues surrounding the performance of multi-threaded code from a low-level perspective, with a look at false sharing and data contention before moving on to how the patterns of data access can affect the performance of a bit of code. We then looked at additional considerations in the design of concurrent code, such as exception safety and scalability. Finally, we've ended with a number of examples of parallel algorithm implementations, each of which have highlighted particular issues that can occur when designing multi-threaded code.

One item that has cropped up a couple of times in this chapter is the idea of a “thread pool” — a preconfigured group of threads which run tasks assigned to the pool. There is quite a lot of thought that goes into the design of a good thread pool, so we'll look at some of the issues in the next chapter, along with other aspects of advanced thread management.

9

Advanced Thread Management

In earlier chapters, we've been explicitly managing our threads by creating **`std::thread`** objects for each and every thread. In a couple of places we've seen how this can be undesirable, as we then have to manage the lifetime of the thread objects, determine the number of threads appropriate to the problem and to the current hardware, and so forth. The ideal scenario would be that we could just divide the code up into the smallest pieces that can be executed concurrently, pass them over to the compiler and library and say "parallelize this for optimal performance".

Another recurring theme in several of the examples is that you might use several threads to solve a problem, but require that they finish early if some condition is met. This might be because the result has already been determined, or because an error has occurred, or even because the user has explicitly requested that the operation be aborted. Whatever the reason, the threads need to be sent a "please stop" request so that they can give up on the task they were given, tidy up, and finish as soon as possible.

In this chapter, we're going to look at mechanisms for both of these, starting with the automatic management of the number of threads, and the division of tasks between them.

9.1 Thread Pools

In many companies, employees that would normally spend their time in the office are occasionally required to visit clients or suppliers, or attend a trade show or conference. Though these trips might be necessary, and on any given day there might be several people making such a trip, it may well be months or even years between such trips for any particular employee. While it would therefore be rather expensive and impractical for each employee to have a company car, companies often offer a *car pool* instead: they have a limited number of cars which are available to all employees. When an employee needs to make an off-site trip they then book one of the pool cars for the appropriate time, and return it for others to use when they are back in the office. If there are no pool cars free on a given day then the employee will have to reschedule their trip for a subsequent date.

A *thread pool* is a similar idea, except that it is *threads* that are being shared rather than cars. On most systems, it is impractical to have a separate thread for every task that can potentially be done in parallel with other tasks, but we'd still like to take advantage of the available concurrency where possible. A thread pool allows us to accomplish this: tasks that can be executed concurrently are submitted to the pool, which puts them on a queue of pending work. Each task is then taken from the queue by one of the *worker threads*, which executes the task before looping back to take another from the queue.

There are several key design issues when building a thread pool, such as how many threads to use, the most efficient way to allocate tasks to threads, and whether or not you can wait for a task to complete. In this section we'll look at some thread pool implementations that address these design issues, starting with the simplest possible thread pool.

9.1.1 The Simplest Possible Thread Pool

At its simplest, a thread pool is just a fixed number of *worker threads* (typically the same number as the value returned by **`std::thread::hardware_concurrency()`**) which process work. When you have work to do, you call a function to put it on the queue of pending work. Each worker thread takes work off the queue, runs the specified task and then goes back to the queue for more work. In the simplest case there is no way to wait for the task to complete — if you need to do this, then you have to manage the synchronization yourself.

Listing 9.1 shows a sample implementation of such a thread pool.

Listing 9.59: Simple thread pool

```
class thread_pool
{
    std::atomic_bool done;
    thread_safe_queue<std::function<void()> > work_queue;             #2
    std::vector<std::thread> threads;                                 #1
    join_threads joiner;                                              #7

    void worker_thread()
    {
        while(!done)                                                  #9
        {
            std::function<void()> task;
            if(work_queue.try_pop(task))                              #10
            {
                task();                                               #11
            }
            else
            {
                std::this_thread::yield();                            #12
                }
            }
        }

    public:
        thread_pool():
            joiner(threads),done(false)
        {
            unsigned const thread_count=std::thread::hardware_concurrency();#4

            try
            {
                for(unsigned i=0;i<thread_count;++i)
                {
                    threads.push_back(
                        std::thread(&thread_pool::worker_thread,this));    #5
                }
            }
            catch(...)
            {
                done=true;                                                #6
                throw;
            }
        }

        ~thread_pool()
        {
            done=true;
        }

        template<typename FunctionType>
        void submit(FunctionType f)
        {
            work_queue.push(std::function<void()>(f));                    #3
        }
    };
```

Cueballs in code and text

This implementation has a vector of worker threads (#1), and uses one of the thread-safe queues from chapter 6 (#2) to manage the queue of work. In this case, users can't wait for the tasks, and they can't return any values, so we can use **std::function<void()>** to encapsulate our tasks. The **submit()** function then just wraps whatever function or callable object is supplied inside a **std::function<void()>** instance and pushes it on the queue (#3).

The threads are started in the constructor: we use **std::thread::hardware_concurrency()** to tell us how many concurrent threads the hardware can support (#4), and create that many threads running our **worker_thread()** member function (#5).

Now, starting a thread can fail by throwing an exception, so we need to ensure that any threads we'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 (#6), alongside an instance of the **join_threads** class from chapter 8 (#7) to join all the threads. This also works with the destructor: we can just set the **done** flag (#8), 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 really quite simple: it sits in a loop waiting until the **done** flag is set (#9), pulling tasks off the queue (#10) and executing them (#11) in the mean time. If there are no tasks on the queue, then the function calls **std::this_thread::yield()** to take a small break (#12), and give another thread a chance to put some work on the queue before it tries to take some off again the next time round.

For many purposes such a simple thread pool will suffice, especially if the tasks are entirely independent and do not return any values or perform any blocking operations. However, there are also many circumstances where such a simple thread pool may not adequately address our needs, and yet others where it can cause problems such as deadlock. Through this chapter, we will 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.

9.1.2 Waiting for Tasks Submitted to a Thread Pool

In the examples in chapter 8, after dividing the work between threads, the master thread always waited for the newly-spawned threads to finish, to ensure that the overall task is complete before returning to the caller. With thread pools, we would need to wait for the *tasks* submitted to the thread pool to complete, rather than the worker threads themselves. With the simple thread pool from listing 9.1, we would have to do this manually using the techniques from chapter 4 — condition variables and futures. This adds complexity to the code; it would be better if we could just wait for the tasks directly.

By moving that complexity into the thread pool itself, we *can* wait for the tasks directly — we can have the **submit()** function return a task handle of some description that we can then use to wait for the task to complete. This task handle would wrap the use of condition variables or futures, thus simplifying the code that uses the thread pool.

A special case of having to wait for the spawned task to finish is when the main thread actually needs a result computed by the task. We've seen this in examples throughout the book, such as the **parallel_accumulate()** function from chapter 2. In this case, we can combine the waiting with the result transfer through the use of futures. Listing 9.2 shows a the changes required to our simple thread pool that allows you to wait for tasks to complete, and pass return values from the task to the waiting thread.

Listing 9.60: A thread pool with waitable tasks

```
class thread_pool
{
public:
    template<typename ResultType>
    using task_handle=std::unique_future<ResultType>;                #1

    template<typename FunctionType>
    task_handle<std::result_of<FunctionType()>::type>                #2
        submit(FunctionType f)
    {
        typedef std::result_of<FunctionType()>::type result_type;     #3

        std::packaged_task<result_type()> task(f);                   #4
        task_handle<result_type> res(task.get_future());             #5
        work_queue.push(std::move(task));                            #6
        return res;                                                  #7
    }
    // rest as before
};
```

Cueballs in Code and Text

Firstly, we've defined a **task_handle<>** template alias as another name for **std::unique_future<>** (#1). This is a new C++0x feature similar to the use of **typedef**, except that it is a template: **task_handle<int>** is an alias for **std::unique_future<int>**, and **task_handle<std::string>** is an alias for **std::unique_future<std::string>**. We could just use **std::unique_future<>** directly, but the use of the new name **task_handle** means we can make it a proper class in the future and add new features without having to alter our code too much.

Our new **task_handle<>** template is used for the return type for the modified **submit()** function (#2). This requires that we know the return type of the supplied function **f**, which is where **std::result_of<>** comes in: **std::result_of<FunctionType()>::type** is the type of the result of invoking an instance of type **FunctionType** (such as **f**) with no arguments. We use the same **std::result_of<>** expression for the **result_type typedef** (#3) inside the function.

We then wrap our function **f** in a **std::packaged_task<result_type()>** (#4), since **f** is a function or callable object that takes no parameters and returns an instance of type **result_type**, as we just deduced. We can now get our future, and build our **task_handle<result_type>** from it (#5), before pushing the task onto the queue (#6), and returning the task handle (#7). Note that we have to use **std::move()** when pushing the task onto the queue, since **std::packaged_task<>** is not copyable. Our queue still stores **std::function<void()>**, but that's OK: the function call operator on **std::packaged_task<result_type()>** fits this profile.

This pool thus allows us to wait for our tasks, and have them return results. Listing 9.3 shows what our **parallel_accumulate** function looks like with such a thread pool.

Listing 9.61: parallel_accumulate using a thread pool with waitable tasks

```
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 block_size=25;
    unsigned long const num_blocks=(length+block_size-1)/block_size;   #1

    std::vector<thread_pool::task_handle<T> > futures(num_blocks-1);
    thread_pool pool;

    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);
        futures[i]=pool.submit(accumulate_block<Iterator,T>());        #2
        block_start=block_end;
    }
    T last_result=accumulate_block()(block_start,last);
    T result=init;
    for(unsigned long i=0;i<(num_blocks-1);++i)
    {
        result+=futures[i].get();
    }
    result += last_result;
    return result;
}
```

Cueballs in code and text

When we compare this against listing 8.4, there are a couple of things to notice. Firstly, we're working in terms of the number of blocks to use (**num_blocks** (#1)) rather than the number of threads — in order to make the most use of the scalability of our thread pool, we need to divide the work into the smallest blocks that it's worth working with concurrently. When there are only a few threads in the pool, each thread will process many blocks, but as the number of threads grows with the hardware, the number of blocks processed in parallel will also grow.

Secondly, we don't have to worry about packaging the tasks, obtaining the futures or storing the **std::thread** objects so we can join with the threads later — the thread pool takes care of that. All we need to do is call **submit()** with our task (#2).

The thread pool takes care of the exception safety too — any exception thrown by the task gets propagated through the future returned from **submit()**, and if the function exits with an exception then the thread pool destructor abandons any not-yet-completed tasks and waits for the pool threads to finish.

This works really well for simple cases like this, where the tasks are independent. However, it's not so good for situations where the tasks depend on other tasks also submitted to the thread pool.

9.1.3 Tasks that Wait for Other Tasks

The quick sort algorithm is an example that we've used throughout this book. It is very simple in concept: the data to be sorted is partitioned into those items that go before a pivot item, and those that go after it in the sorted sequence. These two sets of items are then recursively sorted, and then stitched back together to form a fully sorted set. When parallelizing this algorithm, we need to ensure that these recursive calls make use of the available concurrency.

Back in chapter 4, when we first introduced this example, we used a **spawn_task** function to run one of the recursive calls at each stage on a new thread. Unfortunately, this has the down side that the number of threads can become very large if our data set is very large. When we revisited the implementation in chapter 8, we sought to restrict the number of threads to less than the available hardware concurrency. To do this, we used a stack of pending chunks that needed sorting. As each thread partitioned the data it was sorting, it added a new chunk to the stack for one of the sets of data, and then sorted the other one directly. At this point, a straightforward wait for the sorting of the other chunk to complete would potentially deadlock, since we would be consuming one of our limited number of threads waiting — it would be very easy to end up in a situation where all of the threads were waiting for chunks to be sorted, and no threads actually doing any sorting. We addressed this issue by having the threads pull chunks off the stack and sort them whilst the particular chunk they were waiting for was unsorted.

We would get the same problem if we changed our implementation from chapter 4 to use a simple thread pool like the ones we've seen so far in this chapter — there are now only a limited number of threads, and they might end up all waiting for tasks that have not been scheduled because there are no free threads. We therefore need to use a solution similar to the one we used in chapter 8 — process outstanding chunks whilst we're waiting for our chunk to complete. If we're using the thread pool to manage the list of tasks and their association with threads — which is, after all, the whole point of using a thread pool — then we don't have access to the task list to do this. What we need is to modify the thread pool to do this automatically.

The simplest way to do this is to add a new function on **thread_pool** to run a task from the queue, and manage the loop ourselves, so we'll go with that. Advanced thread pool implementations might add logic into the wait function or additional wait functions to handle this case, possibly prioritizing the task being waited for. Listing 9.4 shows the new **run_pending_task()** function, and a modified quick sort to make use of it is shown in listing 9.5.

Listing 9.62: An implementation of run_pending_task()

```
void thread_pool::run_pending_task()
{
    std::function<void()> task;
    if(work_queue.try_pop(task))
    {
        task();
    }
    else
    {
        std::this_thread::yield();
    }
}
```

This implementation of **run_pending_task()** is lifted straight out of the main loop of our **worker_thread()** function, which can now be modified to call the extracted **run_pending_task()**. This just tries to take a task of the queue, and run it if there is one, otherwise it yields to allow the OS to reschedule the thread. The quick sort implementation in listing 9.5 is a lot simpler than the corresponding version from listing 8.1, as all the thread management logic has been moved to the thread pool.

Listing 9.63: A thread-pool based implementation of quick sort

```
template<typename T>
```

```
struct sorter                                                      #1
{
    thread_pool pool;                                              #2

    std::list<T> do_sort(std::list<T>& chunk_data)
    {
        if(chunk_data.empty())
        {
            return chunk_data;
        }

        std::list<T> result;
        result.splice(result.begin(),chunk_data,chunk_data.begin());
        T const& partition_val=*result.begin();

        typename std::list<T>::iterator divide_point=
            std::partition(chunk_data.begin(),chunk_data.end(),
                           less_than<T>(partition_val));

        std::list<T> new_lower_chunk;
        new_lower_chunk.splice(new_lower_chunk.end(),
                               chunk_data,chunk_data.begin(),
                               divide_point);

        thread_pool::task_handle<std::list<T> > new_lower=            #3
            pool.submit(std::bind(&sorter::do_sort,this,
                                  std::move(new_lower_chunk)));

        std::list<T> new_higher(do_sort(chunk_data));

        result.splice(result.end(),new_higher);
        while(!new_lower.is_ready())
        {
            pool.run_pending_task();                                  #4
        }

        result.splice(result.begin(),new_lower.get());
        return result;
    }
};


template<typename T>
std::list<T> parallel_quick_sort(std::list<T> input)
{
    if(input.empty())
    {
        return input;
    }
    sorter<T> s;
```

```
        return s.do_sort(input);
    }
```

Cueballs in code and text

Just as in listing 8.1, we've delegated the real work to the **do_sort()** member function of the **sorter** class template (#1), though in this case the class is only there to wrap the **thread_pool** instance (#2).

Our thread and task management is now reduced to submitting a task to the pool (#3), and running pending tasks whilst waiting (#4). This is much simpler than in listing 8.1, where we had to explicitly manage the threads and the stack of chunks to sort. When submitting the task to the pool, we use **std::bind()** to bind the **this** pointer to **do_sort()**, and to supply the chunk to sort — in this case, we call **std::move()** on the **new_lower_chunk** as we pass it in, to ensure that the data is moved rather than copied.

Though this has now addressed the crucial deadlock-causing problem, this thread pool is still far from ideal. For starters, every call to **submit()** and every call to **run_pending_task()** is accessing the same queue. We saw in chapter 8 how having a single set of data modified by multiple threads can have a detrimental effect on performance, so we need to somehow address this problem.

9.1.4 Avoiding Contention on the Work Queue

Every time a thread calls **submit()** on a particular instance of our thread pool it has to push a new item onto the single shared work queue. Likewise, the worker threads are continually popping items off the queue in order to run the tasks. This means that as the number of processors increases, there is increasing contention on the queue. This can be a real performance drain — even if you use a lock-free queue, so there is no explicit waiting, the cache ping-pong can be a substantial time sink.

One way to handle avoid the cache ping-pong is to use a separate work queue per thread. Each thread then posts new items to it's own queue, and only takes work from the global work queue if there is no work on its own individual queue. Listing 9.6 shows an implementation that makes use of a **thread_local** variable to ensure that each thread has its own work queue, as well as the global one.

Listing 9.64: A thread-pool with thread-local work queues.

```
    class thread_pool
    {
        thread_safe_queue<std::function<void()> > pool_work_queue;
        typedef std::queue<std::function<void()> > local_queue_type;       #6
        static thread_local std::unique_ptr<local_queue_type>
            local_work_queue;                                              #1

        void worker_thread()
        {
            local_work_queue.reset(new local_queue_type);                  #2

            while(!done)
            {
                run_pending_task();
            }
        }

    public:
        template<typename FunctionType>
        task_handle<std::result_of<FunctionType()>::type>
            submit(FunctionType f)
        {
            typedef std::result_of<FunctionType()>::type result_type;

            std::packaged_task<result_type()> task(f);
            task_handle<result_type> res(task.get_future());
            if(local_work_queue)                                           #3
            {
                local_work_queue->push(std::move(task));
            }
            else
            {
                pool_work_queue.push(std::move(task));                     #4
            }
            return res;
        }

        void run_pending_task()
        {
            std::function<void()> task;
            if(local_work_queue && !local_work_queue->empty())             #5
            {
                task=std::move(local_work_queue->front());
                local_work_queue->pop();
                task();
            }
            else if(pool_work_queue.try_pop(task))                         #7
            {
                task();
            }
            else
            {
                std::this_thread::yield();
            }
        }
        // rest as before
    };
```

Cueballs in code and text

I've used a **std::unique_ptr<>** to hold the thread-local work queue (#1) since we don't want non-pool threads to have one; this is initialized in the **worker_thread()** function before the processing loop (#2). The destructor of **std::unique_ptr<>** will ensure that the work queue is destroyed when the thread exits.

submit() then checks to see if the current thread has a work queue (#3). If it does, then it's a pool thread, and we can put the task on the local queue; otherwise we need to put the task on the pool queue as before (#4).

There's a similar check in **run_pending_task()** (#5), except this time we also need to check to see if there are any items on our local queue. If there are, then we can take the front one, and process it — notice that the local queue can be a plain **std::queue<>** (#6) since it is only ever accessed by the one thread. Of course, if there are no tasks on the local queue then we try the pool queue as before (#7).

This works fine for reducing contention, but when the distribution of work is uneven it can easily result in one thread having a lot of work on its queue, whilst the others have no work do to. For example, with our quick sort example, only the top-most chunk would make it to the pool queue, since the remaining chunks would end up on the local queue of the worker thread that processed that one. This defeats the purpose of using a thread pool.

Thankfully, there is a solution to this — allow the threads to *steal* work from each other's queues if there is no work in their queue, and no work in the global queue.

9.1.5 Work-Stealing

In order to allow a thread with no work to do to take work from another thread with a full queue, the queue must be accessible to the thread doing the stealing from **run_pending_tasks()**. This requires that each thread register it's queue with the thread pool, or be given one by the thread pool. Also, we must ensure that the data in the work queue is suitably synchronized and protected, so that our invariants are protected.

It is possible to write a lock-free queue that allows the owner thread to push and pop at one end, whilst other threads can steal entries from the other, but the implementation of such a queue is beyond the scope of this book. In order to demonstrate the idea, we will stick to using a mutex to protect the queue's data. We hope work stealing to be a rare event, so there should be little contention on the mutex, and such a simple queue should therefore have minimal overhead. A simple lock-based implementation is shown in listing 9.7.

Listing 9.65: Lock-based queue for work-stealing

```
class work_stealing_queue
{
private:
    typedef std::function<void()> data_type;
    std::deque<data_type> the_queue;                                 #1
    mutable std::mutex the_mutex;

public:
    work_stealing_queue()
    {}

    work_stealing_queue(const work_stealing_queue& other)=delete;
    work_stealing_queue& operator=(
        const work_stealing_queue& other)=delete;

    void push(data_type const& data)                                  #2
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        the_queue.push_front(data);
    }

    bool empty() const
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        return the_queue.empty();
    }

    bool try_pop(data_type& res)                                      #3
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }

        res=std::move(the_queue.front());
        the_queue.pop_front();
        return true;
    }

    bool try_steal(data_type& res)                                    #4
    {
        std::lock_guard<std::mutex> lock(the_mutex);
        if(the_queue.empty())
        {
            return false;
        }
```

```
            res=std::move(the_queue.back());
            the_queue.pop_back();
            return true;
        }
    };
```

Cueballs in code and text

This queue is a very simple wrapper around a **std::deque<std::function<void()>>** (#1) that protects all accesses with a mutex lock. Both **push()** (#2) and **try_pop()** (#3) work on the front of the queue, whilst **try_steal()** (#4) works on the back.

This actually means that this “queue” is a last-in-first-out stack for its own thread — the task most recently pushed on is the first one off again. This can help improve performance from a cache perspective, since the data related to that task is more likely to still be in the cache than the data related to a task pushed on the queue previously. Also, it maps nicely to algorithms such as quick sort: in our implementation above, each call to **do_sort()** pushes one item on the stack and then waits for it. By processing the most recent item first we ensure that the chunk needed for the current call to complete is processed before the chunks needed for the other branches, thus reducing the number of active tasks, and the total stack usage. **try_steal()** takes items from the opposite end of the queue to **try_pop()** in order to minimize contention — we could potentially use the techniques discussed in chapters 6 and 7 to enable concurrent calls to **try_pop()** and **try_steal()**.

OK, so we have our nice sparkly work queue that permits stealing; how do we use it in our thread pool? Listing 9.8 shows one potential implementation.

Listing 9.66: A thread pool that uses work stealing

```
    class thread_pool
    {
        typedef std::function<void()> task_type;

        std::atomic_bool done;
        thread_safe_queue<task_type> pool_work_queue;
        std::vector<std::unique_ptr<work_stealing_queue> > queues;         #3
        std::vector<std::thread> threads;
        join_threads joiner;

        static thread_local work_stealing_queue* local_work_queue;          #1
        static thread_local unsigned my_index;

        void worker_thread(unsigned my_index_)
        {
            my_index=my_index_;
            local_work_queue=queues[my_index].get();                        #4
            while(!done)
            {
                run_pending_task();
            }
        }

        bool pop_task_from_local_queue(task_type& task)
        {
            return local_work_queue && local_work_queue->try_pop(task);
        }

        bool pop_task_from_pool_queue(task_type& task)
        {
            return pool_work_queue.try_pop(task);
        }

        bool pop_task_from_other_thread_queue(task_type& task)             #8
        {
            for(unsigned i=0;i<queues.size();++i)
            {
                unsigned const index=(my_index+i+1)%queues.size();         #9
                if(queues[index]->try_steal(task))
                {
                    return true;
                }
            }

            return false;
        }

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

            try
            {
                for(unsigned i=0;i<thread_count;++i)
                {
                    queues.push_back(std::unique_ptr<work_stealing_queue>( #2
                                         new work_stealing_queue));
                    threads.push_back(
                        std::thread(&thread_pool::worker_thread,this,i));
                }
            }
            catch(...)
            {
                done=true;
                throw;
            }
        }

        ~thread_pool()
        {
            done=true;
        }

        template<typename ResultType>
        using task_handle=std::unique_future<ResultType>;

        template<typename FunctionType>
        task_handle<std::result_of<FunctionType()>::type> submit(
            FunctionType f)
        {
            typedef std::result_of<FunctionType()>::type result_type;

            std::packaged_task<result_type()> task(f);
            task_handle<result_type> res(task.get_future());
            if(local_work_queue)
            {
                local_work_queue->push(std::move(task));
            }
            else
            {
                pool_work_queue.push(std::move(task));
            }
            return res;
        }

        void run_pending_task()
        {
            task_type task;
            if(pop_task_from_local_queue(task) ||                              #5
               pop_task_from_pool_queue(task) ||                               #6
               pop_task_from_other_thread_queue(task))                         #7
            {
                task();
            }
            else
            {
                std::this_thread::yield();
            }
        }
    };
```

Cueballs in Code and Text

This code is actually very similar to listing 9.6. The first difference is that each thread has a **work_stealing_queue** rather than a plain **std::queue<>** (#1). When each thread is created, rather than allocating it's own work queue, the pool constructor allocates one (#2), which is then stored in the list of work queues for this pool (#3). The index of the queue in the list is then passed in to the thread function and used to retrieve the pointer to the queue (#4). This means that the thread pool can access the queue when trying to steal a task for a thread that has no work to do — **run_pending_task()** will now try to take a task from its thread's own queue (#5), take a task from the pool queue (#6), or take a task from the queue of another thread (#7).

pop_task_from_other_thread_queue() (#8) just iterates through the queues belonging to all the threads in the pool, trying to steal a task from each in turn. In order to avoid every thread trying to steal from the first thread in the list, each thread starts at the next thread in the list, by offsetting the index of the queue to check by its own index (#9).

Now we have a working thread pool, that's good for many potential uses. Of course, there are still a myriad of ways to improve it for any particular usage, but that's left as an exercise for the reader. One aspect in particular that hasn't been explored is the idea of dynamically resizing the thread pool to ensure that there is optimal CPU usage even when threads are blocked waiting for something such as I/O, or a mutex lock.

Next on the list of “advanced” thread management techniques is interrupting threads.

9.2 Interrupting Threads

In many situations it is desirable to signal to a long-running thread that it is time to stop. This might be because it is a worker thread for a thread-pool, and the pool is now being destroyed, or it might be because the work being done by the thread has been explicitly cancelled by the user, or a myriad of other reasons. Whatever the reason, the idea is the same — you need to signal from one thread that another should stop before it reaches the natural end of its processing, and you need to do this in a way that allows that thread to terminate nicely rather than abruptly pulling the rug from under it.

You could potentially design a separate mechanism for each and every case where you need to do this, but that would be overkill — not only does a common mechanism make it easier to write the code on subsequent occasions, but it can allow you to write code that can be interrupted, without having to worry about where that code is being used. The C++0x standard doesn't provide such a mechanism, but it is relatively straightforward to build one. Let's take a look at how we can do that, starting from the point of view of the launching and interrupting interface rather than the thread being interrupted.

9.2.1 Launching and Interrupting Another Thread

To start with, let's look at the external interface — what do we need from an interruptible thread? At the basic level, all we need is the same interface as we have for **std::thread**, with an additional **interrupt()** function:

```
class interruptible_thread
{
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f);
    void join();
    void detach();
    bool joinable() const;
    void interrupt();
};
```

Internally, we can just use **std::thread** to manage the thread itself, and some custom data structure to handle the interruption. Now, what about from the point of view of the thread itself? At the most basic level we want to be able to say "I can be interrupted here" — we want an *interruption point*. For this to be usable without having to pass down additional data it needs to be a simple function that can be called without any parameters — **interruption_point()**. This essentially implies that the interruption-specific data structure needs to be accessible through a **thread_local** variable which is set when the thread is started, so that when a thread calls our **interruption_point()** function it checks the data structure for the currently executing thread. We'll look at the implementation of **interruption_point()** later.

This **thread_local** flag is the primary reason we can't just use plain **std::thread** to manage the thread — it needs to be allocated in a way that the **interruptible_thread** instance can access as well as the newly-started thread. We can do this by wrapping the supplied function before we pass it to **std::thread** to actually launch the thread in the constructor, as shown in listing 9.9.

Listing 9.67: Basic implementation of interruptible_thread

```
class interrupt_flag
{
public:
    void set();
    bool is_set() const;
};
thread_local interrupt_flag this_thread_interrupt_flag;

class interruptible_thread
{
    std::thread internal_thread;
    interrupt_flag* flag;
public:
    template<typename FunctionType>
    interruptible_thread(FunctionType f)
    {
        struct wrapper
        {
            FunctionType m_f;

            wrapper(FunctionType& f_):
                m_f(f_)
            {}

            void operator()(interrupt_flag** flag_ptr,
                            std::mutex* m,std::condition_variable* c)
            {
                {
                    std::lock_guard<std::mutex> lk(m);
                    *flag_ptr=&this_thread_interrupt_flag;
                }
                c.notify_one();

                m_f();
            }
        };

        std::mutex flag_mutex;
        std::condition_variable flag_condition;
        flag=0;
        internal_thread=std::thread(wrapper(f),&flag,
                                    &flag_mutex,&flag_condition);
        std::unique_lock ul(flag_mutex);
        while(!flag)
        {
            flag_condition.wait(ul);
        }
    }

    void interrupt()
    {
        if(flag)
        {
            flag->set();
        }
    }
};
```

The wrapper takes a copy of the supplied function object, and stores it internally. When the wrapper is invoked on the new thread, it receives pointers to a pointer-to-**interrupt_flag**, and a mutex and condition variable. It then takes the address of the interrupt flag for the current thread, stores it in the supplied pointer (with the mutex locked) and notifies the condition variable before invoking the wrapped function. After the **interruptible_thread** constructor has started the new thread through the wrapper, it then waits on the condition variable until the flag pointer has been set. The **interrupt()** function is then relatively straight-forward: if we have a valid pointer to an interrupt flag then we have a thread to interrupt, so we can just set the flag. It's then up to the interrupted thread what it does with the interruption. Let's take a look at that next.

9.2.2 Detecting that a Thread has been Interrupted

We can now set the interruption flag, but that doesn't to us any good if the thread doesn't actually check whether it's being interrupted. In the simplest case we can do this with an **interruption_point()** function: you can call this function at a point where it is safe to be interrupted, and it throws a **thread_interrupted** exception if the flag is set:

```
void interruption_point()
{
    if(this_thread_interrupt_flag.is_set())
    {
        throw thread_interrupted();
    }
}
```

You can use such a function by calling it at convenient points within your code:

```
void foo()
{
    while(!done)
    {
        interruption_point();
        process_next_item();
    }
}
```

Whilst this works, it's not ideal — some of the best places for interrupting a thread are where it is blocked waiting for something, which means that the thread is not running in order to call **interruption_point()**! What we need here is a means for waiting for something in an interruptible fashion.

9.2.3 Interrupting a Condition Variable Wait

OK, so we can detect interruptions at carefully chosen places in our code, with explicit calls to **interruption_point()**, but that doesn't help when we want to do a blocking wait, such as waiting for a condition variable to be notified. We need a new function — **interruptible_wait()** — which we can then overload for the various things we might want to wait for, and which we can work out how to interrupt the waiting. We've already mentioned that one thing we might be waiting for is a condition variable, so let's start there:

what do we need to do in order to be able to interrupt a wait on a condition variable? The simplest thing that would work is to notify the condition variable once we'd set the interrupt flag, and put an interruption point immediately after the wait. However, for this to work we would have to notify all threads waiting on the condition variable in order to ensure that our thread of interest wakes up. Waiters have to handle spurious wake-ups anyway, so other threads would just handle this the same as a spurious wake up — they wouldn't be able to tell the difference. Our **interrupt_flag** structure would need to be able to store a pointer to a condition variable so that it can be notified in a call to **set()**. One possible implementation of **interruptible_wait()** for condition variables might look like listing 9.10.

Listing 9.68: A broken implementation of interruptible_wait for std::condition_variable

```
void interruptible_wait(std::condition_variable& cv,
                        std::unique_lock<std::mutex>& lk)
{
    interruption_point();
    this_thread_interrupt_flag.set_condition_variable(cv);          #1
    cv.wait(lk);                                                    #2
    this_thread_interrupt_flag.clear_condition_variable();          #3
    interruption_point();
}
```

Cueballs in code and text

Assuming the presence of some functions for setting and clearing an association of a condition variable with an interrupt flag, this code is nice and simple. It checks for interruption, associates the condition variable with the **interrupt_flag** for the current thread (#1), waits on the condition variable (#2), clears the association with the condition variable (#3) and checks for interruption again. If the thread is interrupted during the wait on the condition variable, then the interrupting thread will broadcast the condition variable, and wake us from the wait, so we can check for interruption. Unfortunately this code is **broken**: there are two problems with this code. The first problem is relatively obvious if you've got your exception safety hat on: **std::condition_variable::wait()** can throw an exception, so we might exit the function without removing the association of the interrupt flag with the condition variable. This is easily fixed with a structure that removes the association in its destructor.

The second, less obvious problem is that there is a race condition. If the thread is interrupted after the initial call to **interruption_point()**, but before the call to **wait()** then it doesn't matter whether or not the condition variable has been associated with the interrupt flag, because *the thread is not waiting, and so cannot be woken by a notify on the condition variable*. We actually need to ensure that the thread cannot be notified between the last check for interruption and the call to **wait()**. Without delving into the internals of **std::condition_variable** we only have one way of doing that: use the mutex held by **lk** to protect this too, which requires passing it in on the call to **set_condition_variable()**. Unfortunately, this creates its own problems: we would be passing a reference to a mutex whose lifetime we don't know to another thread (the thread doing the interrupting) for that thread to lock (in the call to **interrupt()**), without knowing whether or not that thread has locked the mutex already when it makes the call. This has the potential for deadlock *and* the potential to access a mutex after it has already been destroyed, so is a non-starter. It would be rather too restrictive if we couldn't *reliably* interrupt a condition variable wait — we can do almost as well without a special **interruptible_wait()** — so what other options do we have? One option is to put a timeout on the wait — use **wait_for()** rather than **wait()** with a small timeout value (such as 1ms). This then puts an upper limit on how long the thread will have to wait before it sees the interruption. Of course, if we do this then the waiting thread will see rather more “spurious” wakes due to the timeout, but it can't easily be helped. Such an implementation is shown in listing 9.11, along with the corresponding implementation of **interrupt_flag**.

Listing 9.69: An implementation of interruptible_wait for std::condition_variable using a timeout

```
class interrupt_flag
{
    std::atomic<bool> flag;
    std::condition_variable* thread_cond;
    std::mutex set_clear_mutex;

public:
    interrupt_flag():
        thread_cond(0)
    {}

    void set()
    {
        flag.store(true,std::memory_order_relaxed);
        std::lock_guard<std::mutex> lk(set_clear_mutex);
        if(thread_cond)
        {
            thread_cond->notify_all();
        }
    }

    bool is_set() const
```

```
        {
            return flag.load(std::memory_order_relaxed);
        }

        void set_condition_variable(std::condition_variable& cv)
        {
            std::lock_guard<std::mutex> lk(set_clear_mutex);
            thread_cond=&cv;
        }

        void clear_condition_variable()
        {
            std::lock_guard<std::mutex> lk(set_clear_mutex);
            thread_cond=0;
        }

        struct clear_cv_on_destruct
        {
            ~clear_cv_on_destruct()
            {
                this_thread_interrupt_flag.clear_condition_variable();
            }
        };

    };

    void interruptible_wait(std::condition_variable& cv,
                            std::unique_lock<std::mutex>& lk)
    {
        interruption_point();
        this_thread_interrupt_flag.set_condition_variable(cv);
        interrupt_flag::clear_cv_on_destruct guard;
        interruption_point();
        cv.wait_for(lk,std::chrono::milliseconds(1));
        interruption_point();
    }
```

Of course, if we have the predicate that's being waited for then the 1ms timeout can be completely hidden inside the predicate loop:

```
    template<typename Predicate>
    void interruptible_wait(std::condition_variable& cv,
                            std::unique_lock<std::mutex>& lk,
                            Predicate pred)
    {
        interruption_point();
        this_thread_interrupt_flag.set_condition_variable(cv);
        interrupt_flag::clear_cv_on_destruct guard;
        while(!this_thread_interrupt_flag.is_set() && !pred())
        {
            cv.wait_for(lk,std::chrono::milliseconds(1));
        }
        interruption_point();
    }
```

This will result in the predicate being checked more often than it might otherwise be, but is easily used in place of a plain call to **wait()**. The variants with timeouts are easily implemented: wait either for the time specified, or 1ms, whichever is shortest. OK, so that's **std::condition_variable** waits taken care of; what about **std::condition_variable_any**? Is this just the same, or can we do better?

9.2.4 Interrupting a Wait on *std::condition_variable_any*

std::condition_variable_any differs from **std::condition_variable** in that it works with **any** lock type rather than just **std::unique_lock<std::mutex>**. It turns out that this makes things much easier, and we *can* do better with **std::condition_variable_any** than we could with **std::condition_variable** — because it works with **any** lock type, we can build our own lock type that locks/unlocks both the internal **set_clear_mutex** in our **interrupt_flag** *and* the lock supplied to the wait call, as in listing 9.12.

Listing 9.70: An implementation of interruptible_wait for std::condition_variable_any using a custom lock type

```
    class interrupt_flag
    {
        std::atomic<bool> flag;
        std::condition_variable* thread_cond;
        std::condition_variable_any* thread_cond_any;
        std::mutex set_clear_mutex;

    public:
        interrupt_flag():
            thread_cond(0),thread_cond_any(0)
        {}

        void set()
        {
            flag.store(true,std::memory_order_relaxed);
            std::lock_guard<std::mutex> lk(set_clear_mutex);
            if(thread_cond)
            {
                thread_cond->notify_all();
            }
            else if(thread_cond_any)
            {
                thread_cond_any->notify_all();
            }
        }
        template<typename Lockable>
        void wait(std::condition_variable_any& cv,Lockable& lk)
        {
            struct custom_lock
            {
                interrupt_flag* self;
                Lockable& lk;

                custom_lock(interrupt_flag* self_,
                            std::condition_variable_any& cond,
                            Lockable& lk_):
                    self(self_),lk(lk_)
                {
                    self->set_clear_mutex.lock();
                    self->thread_cond_any=&cond;
                }

                void unlock()
                {
                    lk.unlock();
                    self->set_clear_mutex.unlock();
                }

                void lock()
                {
                    self->set_clear_mutex.lock();
                    lk.lock();
                }

                ~custom_lock()
                {
                    self->thread_cond_any=0;
                    self->set_clear_mutex.unlock();
                }
            };
            custom_lock cl(this,cv,lk);
            interruption_point();
            cv.wait(cl);
            interruption_point();
        }

        // rest as before
    };

    template<typename Lockable>
    void interruptible_wait(std::condition_variable_any& cv,
                            Lockable& lk)
    {
        this_thread_interrupt_flag.wait(cv,lk);
```

```
}
```

Our custom lock type acquires the lock on the internal **set_clear_mutex** when it is constructed, and then sets the **thread_cond_any** pointer to refer to the **std::condition_variable_any** passed in to the constructor. The **Lockable** reference is stored for later — this must already be locked. We can now check for an interruption without worrying about races — if the interrupt flag is set at this point it was set before we acquired the lock on **set_clear_mutex**. When the condition variable calls our **unlock()** function inside **wait()** then we unlock the **Lockable** object *and the internal* **set_clear_mutex**. This allows threads that are trying to interrupt us to acquire the lock on **set_clear_mutex** and check the **thread_cond_any** pointer *once we're inside the* **wait()** *call*, but not before. This is exactly what we were after (but couldn't manage) with **std::condition_variable**. Once **wait()** is done waiting (either because it was notified, or because of a spurious wake), then it will call our **lock()** function, which again acquires the lock on the internal **set_clear_mutex** and the lock on the **Lockable** object. We can now check again for interruptions that happened during the **wait()** call before clearing the **thread_cond_any** pointer in our **custom_lock** destructor, where we also unlock the **set_clear_mutex**.

9.2.5 Interrupting other Blocking Calls

That rounds up interrupting condition variable waits, but what about other blocking waits: mutex locks, waiting for futures, etc.? In general we have to go for the timeout option we used for **std::condition_variable** as there is no way to interrupt the wait short of actually fulfilling the condition being waited for, without access to the internals of the mutex or future, however with those other things we do know what we're waiting for, so we can loop within the **interruptible_wait()** function. As an example, here's an overload of **interruptible_wait()** for a **std::unique_future<>:**

```
template<typename T>
void interruptible_wait(std::unique_future<T>& uf)
{
    while(!this_thread_interrupt_flag.is_set() && !uf.is_ready())
    {
        uf.wait_for(lk,std::chrono::milliseconds(1));
    }
    interruption_point();
}
```

This waits until either the interrupt flag is set or the future is ready, but does a blocking wait on the future for 1ms at a time. This means that on average it will be around 0.5ms before an interrupt request is acknowledged. This may or may not be acceptable, depending on the circumstances — you can always reduce the timeout if necessary. The downside of reducing the timeout is that the thread will wake more often to check the flag, and this will increase the task-switching overhead. For sufficiently short waits, you might as well not bother, and just stick a call to **std::this_thread::yield()** in the loop.

OK, so we've looked at how we might detect interruption, with our **interruption_point()** and **interruptible_wait()** functions, how do we handle that?

9.2.6 Handling Interruptions

From the point of view of the thread being interrupted, an interruption is just a **thread_interrupted** exception, which can therefore be handled just like any other exception. In particular, you can catch it in a standard **catch** block:

```
try
{
    do_something();
}
catch(thread_interrupted&)
{
    handle_interruption();
}
```

This means that you could catch the interruption, handle it in some way, and then carry on regardless. If you do this, and another thread calls **interrupt()** again, your thread will be interrupted again the next time it calls an interruption point. You might want to do this if your thread is performing a series of independent tasks: interrupting one task will cause that task to be abandoned, and the thread can then move onto performing the next task in the list.

Since **thread_interrupted** is an exception, all the usual exception-safety precautions must also be taken when calling code that can be interrupted, in order to ensure that resources aren't leaked, and your data structures are left in a coherent state. Often, it will be desirable to let the interruption terminate the thread, so we can just let the exception propagate up. However, if you let exceptions propagate out of the thread function passed to the **std::thread** constructor. **std::terminate()** will be called, and the whole program will be terminated. In order to avoid having to remember to put a **catch(thread_interrupted)** handler in every function we pass to **interruptible_thread**, we can instead put that catch block inside the wrapper we use for initializing the **interrupt_flag**. This then makes it safe to allow the interruption exception to propagate unhandled, as it will then just terminate that individual thread. Our wrapper's function call operator now looks like this:

```
void interruptible_thread::wrapper::operator()(
    interrupt_flag** flag_ptr,
    std::mutex* m,std::condition_variable* c)
{
    {
        std::lock_guard<std::mutex> lk(m);
```

```
            *flag_ptr=&this_thread_interrupt_flag;
        }
        c.notify_one();
        try
        {
            m_f();
        }
        catch(thread_interrupted&)
        {}
    }
```

Let's now look at a concrete example where interruption is useful.

9.2.7 Interrupting Background Tasks on Application Exit

Consider for a moment a desktop search application. As well as interacting with the user, the application also needs to monitor the state of the file system, identifying any changes and updating its index. Such processing is typically left to a background thread, in order to avoid affecting the responsiveness of the GUI. This background thread needs to run for the entire lifetime of the application: it will be started as part of the application initialization, and left to run until the application is shut down. For such an application this is typically only when the machine itself is being shut down, as the application needs to run the whole time in order to maintain an up-to-date index. In any case, when the application is being shut down we need to close down the background threads in an orderly manner; one way to do this by interrupting them.

Listing 9.13 shows a sample implementation of the thread management parts of such a system.

Listing 9.71: Monitoring the File System in the Background

```
    std::mutex config_mutex;
    std::vector<interruptible_thread> background_threads;

    void background_thread(int disk_id)
    {
        while(true)
        {
            interruption_point();                           #7
            fs_change fsc=get_fs_changes(disk_id);          #5
            if(fsc.has_changes())
            {
                update_index(fsc);                          #6
            }
        }
    }
```

```
void start_background_processing()
{
    background_threads.push_back(
        interruptible_thread(background_thread,disk_1));
    background_threads.push_back(
        interruptible_thread(background_thread,disk_2));
}

int main()
{
    start_background_processing();                    #1
    process_gui_until_exit();                          #2
    std::unique_lock<std::mutex> lk(config_mutex);
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].interrupt();             #3
    }
    for(unsigned i=0;i<background_threads.size();++i)
    {
        background_threads[i].join();                  #4
    }
}
```

Cueballs in code and text

At startup, the background threads are launched (#1). The main thread then proceeds with handling the GUI (#2). When the user has requested that the application exit, the background threads are interrupted (#3), and then the main thread waits for each background thread to complete before exiting (#4). The background threads sit in a loop, checking for disk changes (#5) and updating the index (#6). Every time round the loop they check for interruption by calling **`interruption_point()`** (#7).

Why do we interrupt all the threads before waiting for any? Why not interrupt each and then wait for it before moving on to the next? The answer is of course *concurrency* — threads will likely not finish immediately they are interrupted, as they have to proceed to the next interruption point, and then run any destructor calls and exception handling code necessary before they exit. By joining with each thread immediately we therefore cause the interrupting thread to wait, *even though it still has useful work it could do* — interrupt the other threads. Only when we have no more work to do (all the threads have been interrupted) do we then wait. This also allows all the threads being interrupted to process their interruptions in parallel, and potentially finish sooner.

This interruption mechanism could easily be extended to add further interruptible calls, or to disable interruptions across a specific block of code, but this is left as an exercise for the reader.

9.3 Summary

In this chapter, we've looked at various “advanced” thread management techniques: thread pools, and interrupting threads. We've seen how the use of local work queues and work stealing can reduce the synchronization overhead and potentially improve the throughput of the thread pool, and how running other tasks from the queue whilst waiting for a sub-task to complete can eliminate the potential for deadlock.

We've also looked at various ways of allowing one thread to interrupt the processing of another, such as the use of specific interruption points and functions that perform what would otherwise be a blocking wait in a way that can be interrupted.

10

Testing and Debugging Multi-threaded Applications

Up to now, we've been focussing on what's involved in writing concurrent code — the tools we have available, how to use them, and the overall design and structure of the code. However, there's a crucial part of software development that we haven't addressed yet: testing and debugging. If you're reading this chapter hoping for an easy way to test concurrent code, you're going to be sorely disappointed. Testing and debugging concurrent code is *hard*. What I *am* going to give you are some techniques that will make things easier, alongside issues that it is important to think about.

Testing and debugging are like two sides of a coin — you subject your code to tests in order to try and find any bugs that might be there, and you debug it to remove those bugs. With luck, you only ever have to remove the bugs found by your own tests rather than bugs found by the end users of your application. Before we look at either testing or debugging, it's important to understand the problems that might arise, so let's take a look at those.

10.1 Types of Concurrency-related Bugs

You can get just about any sort of bug in concurrent code; it's not special in that regard. However, there are some types of bugs which are directly related to the use of concurrency, and therefore of particular relevance to this book. Typically, these concurrency-related bugs divide into two primary categories:

- Unwanted blocking and
- Race Conditions.

These are huge categories, so let's divide them up a bit. First, let's look at unwanted blocking.

10.1.1 Unwanted Blocking

What do I mean by unwanted blocking? Firstly, a thread is *blocked* when it is unable to proceed because it is waiting for something. This is typically something like a mutex, a condition variable or a future, but it could be waiting for I/O. This is a natural part of multithreaded code, but it's not always desirable — hence the problem of unwanted blocking. This of course leads us to the next question: why is this blocking unwanted? Typically this is because some other thread is in turn waiting for the blocked thread to perform some action, and so that thread in turn is blocked. There are several variations on this theme:

- Deadlock. As we saw in chapter 3, in the case of deadlock one thread is waiting for another, which is in turn waiting for the first. If your threads deadlock then the tasks they are doing will not get done. In the most visible cases, one of the threads involved is the thread responsible for the user interface, in which case the interface will cease to respond. In other cases the interface will remain responsive, but some required task won't complete, such as a search not returning, or a document not printing.

- Livelock. Livelock is similar to deadlock in that one thread is waiting for another, which is in turn waiting for the first. The key difference here is that the wait is not a blocking wait, but an active checking loop, such as a spin-lock. In serious cases, the symptoms are the same as deadlock (app doesn't make any progress), except that the CPU usage is high because threads are still running, but blocking each other. In not-so-serious cases, the livelock will eventually resolve due to the random scheduling, but there will be a long delay in the task that got live-locked, with a high CPU usage during that delay.

- Blocking on I/O or other external input. If your thread is blocked waiting for external input then it cannot proceed, even if the waited-for input is never going to come. It is therefore undesirable to block on external input from a thread that also performs tasks that other threads may be waiting for.

OK, so that's briefly covered unwanted blocking. What about race conditions?

10.1.2 Race Conditions

Race conditions are the most common cause of problems in multithreaded code — many deadlocks and livelocks only actually manifest due to a race condition. Of course, not all race conditions are problematic — a race condition occurs any time the behaviour depends on the relative scheduling of operations in separate threads. A large number of race conditions are entirely benign; for example, which worker thread processes the next task in the task queue is largely irrelevant. However, many concurrency bugs are due to race conditions. In particular, race conditions often cause the following types of problems:

- Data races. A data race is the specific type of race condition that results in undefined behaviour due to unsynchronized concurrent access to a shared memory location. We introduced data races in chapter 5 when we looked at the C++ memory model. Data races usually occur through incorrect usage of atomic operations to synchronize threads, or through access to shared data without locking the appropriate mutex.

- Broken invariants. This can manifest as dangling pointers (other thread deleted data being accessed), random memory corruption (read inconsistent values due to partial updates), double-free (two threads pop the same value from a queue) amongst others.

- Lifetime issues. Though you could bundle these problems in with the broken invariants, this really is a separate category. The basic problem with bugs in this category is that the thread outlives the data that it accesses, so it is accessing data that has been deleted or otherwise destroyed, and potentially even has had the storage reused for another object. You typically get lifetime issues where a thread references local variables which go out of scope before the thread function has completed, but it is not limited to that scenario. Whenever the lifetime of the thread and the data it operates on are not tied together in some way there is the potential for the data to be destroyed before the thread has finished, and for the thread function to have the rug pulled out from under its feet. If you manually call **`join()`** in order to wait for the thread to complete, you need to ensure that the call to **`join()`** cannot be skipped due to an exception being thrown. This is just basic exception safety applied to threads.

It's the problematic race conditions that are the killers. With deadlock and livelock, the application appears to hang, and become completely unresponsive or takes too long to complete a task. Often, you can attach a debugger to the running process to identify which threads are involved in the deadlock or livelock, and which synchronization objects they are fighting over. With data races, broken invariants and lifetime issues, the visible symptoms of the problem (such as random crashes or incorrect output) can manifest anywhere in the code — the code may overwrite memory used by another part of the system which is not touched until much later. The fault will then manifest in code completely unrelated to the location of the buggy code, possibly much later in the execution of the program. This is the true curse of shared memory systems — however much you try and limit which data is accessible by which thread, and try and ensure that correct synchronization is used, any thread can actually overwrite the data being used by any other thread in the application.

OK, now we've briefly identified the sorts of problems we're looking for, let's look at what we can do to locate any instances in our code so we can fix them.

10.2 Techniques for Locating Concurrency-related Bugs

In the previous section we looked at the types of concurrency-related bugs we might see, and how they might manifest in our code. With that information in mind we can then look at our code to see where bugs might lie, and how we can attempt to determine whether or not there are any bugs in a particular section.

Perhaps the most obvious and straightforward thing to do is *look at the code*. Though this might seem obvious, it is actually very hard to do in a thorough way. When you read code you've just written, it's all too easy to read what you intended rather than what is actually there. Likewise, when reviewing code that others have written it is very tempting to just give it a quick read through, check it off against your local coding standards and highlight any glaringly obvious problems. What is needed is to spend the time really going through the code with a fine-toothed comb thinking about the concurrency issues (and the non-concurrency issues as well — you might as well, whilst you're doing it. After all, a bug is a bug). We'll cover specific things to think about when reviewing code shortly.

Even after thoroughly reviewing our code, we still might have missed some bugs, and in any case we need to confirm that it does indeed work, for peace of mind if nothing else. Consequently, we'll follow on from reviewing the code to a few techniques to employ when testing multithreaded code.

10.2.1 Reviewing Code to Locate Potential Bugs

As I've already mentioned, when reviewing multithreaded code to check for concurrency-related bugs it is really important to review it thoroughly, with a fine-toothed comb. If possible, it helps if you can get someone else to review it. Because they haven't written the code, they will have to think through how it works, and this will help to uncover any bugs that there may be. It is important that the reviewer has the time to do the review properly; not a casual two-minute quick glance, but a proper, considered review. Most concurrency bugs require more than a quick glance to spot — they usually rely on subtle timing issues to actually manifest.

By getting one of your colleagues to review the code, they're coming at it fresh, and will therefore see things from a different point of view, and may well spot things that you don't. If you don't have colleagues you can ask, ask a friend, or even post the code on the internet (taking care not to upset your company lawyers). If you can't get anybody to review your code for you, or they don't find anything, don't worry — there's still more you can do. For starters, it might be worth leaving the code alone for a while — work on another part of the application, read a book or go for a walk. If you take a break then your unconscious can work on the problem in the background, whilst you are consciously focused on something else. Also, the code will be less familiar when you come back to it — you might manage to look at it from a different perspective yourself.

An alternative to getting someone else to review your code is to do it yourself. One useful technique is to try and explain how it works *in detail* to someone else. They don't even have to be physically there — many teams have a bear or rubber chicken for this purpose, and I personally find that writing detailed notes can be hugely beneficial. As you explain, think about each line, what could happen, which data it accesses and so forth. Ask yourself questions about the code, and explain the answers. I find this to be an incredibly powerful technique — by asking myself these questions and thinking carefully about the answers, the problem often reveals itself. These questions can be helpful for *any* reviewer, not just when reviewing your own code.

Questions to think about when reviewing multithreaded code

As I've already mentioned, it can be useful for a reviewer (whether the code's author or someone else) to think about specific questions relating to the code being reviewed. These questions can focus the reviewer's mind on the relevant details of the code, and can help identify potential problems. The questions I like to ask include the following, though this is most definitely not a comprehensive list — you might find other questions that help you to focus better. Anyway, the questions are:

- Which data needs to be protected from concurrent access?
- How do you ensure that the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this thread hold?
- Which mutexes might other threads hold?
- Is the data loaded by this thread still valid? Could it have been modified by other threads?
- If we assume that another thread *could* be modifying the data, what would that mean and how could we ensure that this never happened?

This last question is my favourite, as it really makes you think about the relationships between the threads. By assuming the existence of a bug related to a particular line of code, you can then act as a detective and track down the cause. In order to convince yourself that there is no bug, you have to consider every corner case and possible ordering. This is particularly useful where the data is protected by more than one mutex over its lifetime, such as with the thread-safe queue from chapter 6 where we had separate mutexes for the head and tail of the queue: in order to be sure that an access is safe whilst holding one mutex, you have to be certain that a thread holding the *other* mutex can't also access the same element.

The penultimate question in the list is also important, as it addresses what is an easy mistake to make — if you release and then reacquire a mutex you must assume that other threads can have modified the shared data. Though this is obvious, if the mutex locks are not immediately visible — perhaps because they are internal to an object — you may unwittingly be doing exactly that. In chapter 6 we saw how this can lead to race conditions and bugs where the functions provided on a thread-safe data structure are too fine-grained. Whereas for a non-thread-safe stack it makes sense to have separate **top()** and **pop()** operations, for a stack that may be accessed by multiple threads concurrently this is no longer the case because the lock on the internal mutex is released between the two calls, and so another thread can modify the stack. As we saw in chapter 6, the solution is to combine the two operations so they are both performed under the protection of the same mutex lock, thus eliminating the potential race condition.

OK, so you've reviewed your code (or got someone else to review it). You're sure there's no bugs. The proof of the pudding is, as they say, in the eating — how can we test our code to confirm or deny our belief in it's lack of bugs?

10.2.2 Locating concurrency-related bugs by testing

When developing single-threaded applications, testing your applications is relatively straightforward, if time consuming. You could in principle identify all the possible sets of input data (or at least all the interesting cases) and run them through the application. If the application produced the correct behaviour and output then you know it works for that given set of input. Of course, testing for error states such as the handling of disk-full errors is more complicated than that, but the idea is the same — set up the initial conditions, and allow it to run.

Testing multi-threaded code is an order of magnitude harder, because the precise scheduling of the threads is indeterminate, and may vary from run to run. Consequently, even if you run the application with the same input data then it might work correctly some times and fail others if there's a race condition lurking in the code — just because there's a potential race condition doesn't mean the code will fail always, just that it *might* do *sometimes*.

Given the inherent difficulty of reproducing concurrency-related bugs, it pays to design your tests your tests carefully. You want each test to run the smallest amount of code that could potentially demonstrate a problem, so that you can best isolate the code that is faulty if the test fails — it is better to test a concurrent queue directly to verify that concurrent pushes and pops work rather than testing it through a whole chunk of code that uses the queue. It can help if you think about how code should be tested when designing it — see the section on designing for testability later in this chapter.

There's more to testing concurrent code than the structure of the code being tested — the structure of the test is just as important, as is the test environment. If we follow on with the example of testing a concurrent queue then we have to think about various scenarios:

- One thread calling **push()** or **pop()** on its own to verify that the queue does work at a basic level;
- One thread calling **push()** on an empty queue whilst another thread calls **pop()**;
- Multiple threads calling **push()** on an empty queue;
- Multiple threads calling **push()** on a full queue;
- Multiple threads calling **pop()** on an empty queue;
- Multiple threads calling **pop()** on a full queue;
- Multiple threads calling **pop()** on a partially-full queue with insufficient items for all threads;
- Multiple threads calling **push()** whilst one thread calls **pop()** on an empty queue;
- Multiple threads calling **push()** whilst one thread calls **pop()** on a full queue;
- Multiple threads calling **push()** whilst multiple threads call **pop()** on an empty queue;
- Multiple threads calling **push()** whilst multiple threads call **pop()** on a full queue;

Having thought about all these scenarios and more, we then need to consider additional factors about the test environment:

- what we mean by “multiple threads” in each case (3, 4, 1024?);
- whether or not there are enough processing cores in the system for each thread to run on its own core;
- which processor architectures the tests should be run on; and
- how we ensure suitable scheduling for the “whilst” parts of our tests.

There are of course additional factors to think about specific to your particular situation. Of these four environmental considerations, the first and last affect the structure of the test itself, whereas the other two are related to the physical test system being used. The number of threads to use relates to the particular code being tested, but there are various ways of structuring tests to try and obtain suitable scheduling. Before we look at these techniques, let's take a look at how we can design our application code to be easier to test.

10.2.3 Designing for Testability

Testing multithreaded code is hard, so we want to do what we can to make it easier. One of the most important things we can do is design the code for testability. There's a lot that's been written about designing single-threaded code for testability, and much of the advice still applies. In general, code is easier to test if:

- the responsibilities of each function and class are clear;
- the functions are short and to-the-point;
- your tests can take complete control of the environment surrounding the code being tested;
- the code that performs the particular operation being tested is close together rather than spread throughout the system; and
- you thought about how to test the code before you wrote it.

All of this is still true even for multi-threaded code. In fact, I would argue that it's even more important to pay attention to the testability of multi-threaded code than for single-threaded code, since it is inherently that much harder. That last point is important: even if you don't go as far as writing your tests before the code, it's well worth thinking about how you can test the code before you write it — what inputs to use; which conditions are likely to be problematic; how to stimulate the code in potentially problematic ways, etc.

One of the best ways to design concurrent code for testing is to eliminate the concurrency. If you can break the code down into those parts that are responsible for the communication paths between threads and those parts that operate on the communicated data within a single thread then you've greatly reduced the problem. Those parts of the application that operate on data that is only being accessed by that one thread can then be tested using the normal single-threaded techniques. The hard-to-test concurrent code that deals with communicating between threads and ensuring that only the one thread at a time *is* accessing a particular block of data is now much smaller, and the testing more tractable.

For example, if you're application is designed as a multi-threaded state machine then you could split it up into the state logic for each thread, which ensures that the transitions and operations are correct for each possible set of input events, and which can be tested on a single thread, and the core state machine code that ensures that events are correctly delivered to the right thread in the right order, which needs to be tested with multiple concurrent threads.

Alternatively, if you can divide your code up into multiple blocks of *read shared data/transform data/update shared data* then you can test the *transform data* portions using all the usual single-threaded techniques, since this is now just single threaded code. The hard problem of testing a multi-threaded transformation has just been reduced to testing the reading and updating of the shared data, which is much simpler.

One thing to watch out for is library calls can use internal variables to store state, which then becomes shared if multiple threads use the same set of library calls. This can be a problem, because it's not immediately apparent that the code accesses shared data. However, with time you get to learn which library calls these are, and they stick out like a sore thumb. You can then either add appropriate protection and synchronization or use an alternate function that is safe for concurrent access from multiple threads.

There is more to designing multithreaded code for testability than structuring our code to minimize the amount of code that needs to deal with concurrency-related issues, and paying attention to the use of non-threads-safe library calls. It is also helpful to bear in mind the same set of questions we ask ourselves when reviewing the code, from section 10.2.1. Though these questions aren't directly about testing and testability, if we think about the issues with our “testing hat” on and consider how to test the code then it will affect which design choices we make, and make testing easier.

Now we've looked at designing our code to make testing easier, and potentially modified our code into small chunks that manage the threads and communication between them — such as our thread-safe containers or state machine event logic — alongside larger chunks that run solely in a single thread (though interacting with other threads through the concurrent chunks), let's look at the techniques for testing our concurrency-aware code.

10.2.4 Multi-threaded testing techniques

So, you've thought through the scenario you wish to test, and written written a small amount of code that exercises the functions being tested. How do you ensure that the any potentially-problematic scheduling sequences are exercised in order to flush out the bugs?

Well, there's a few ways of approaching this, starting with brute force testing or stress testing.

Brute Force Testing

The idea behind brute force testing or stress testing is to stress the code to see if it breaks. This typically means running the code many many times, possibly with many threads running at once — if there is a bug that only manifests when the threads are scheduled in a particular fashion then the more times the code is run, the more likely it is to manifest. If you run the test once and it passes then you might feel a bit of confidence that the code works. If you run it ten times in a row and it passes every time, you'll likely feel more confident. If you run the test a billion times and it passes every time, then you'll feel more confident still.

The confidence you have in the results does depend on the amount of code being tested by each test. If your tests are quite fine-grained, like the tests outlined for a thread-safe queue above then such brute-force testing can give you quite a high degree of confidence in your code. On the other hand, if the code being tested is considerably larger then the number of possible scheduling permutations is so vast that even a billion test runs might yield a low level of confidence.

The downside to brute force testing is that it might give you false confidence. If the way you've written the test means that the problematic circumstances can't occur then you can run the test as many times as you like and it won't fail, even if it would fail *every* time in slightly different circumstances. The worst example is where the problematic circumstances can't occur on your test system because of the way the particular system you've tested on happens to run. Unless your code is to run only on systems identical to the one being tested then the particular hardware and operating system combination may not allow the circumstances that would cause a problem to arise. The classic example here is testing a multi-threaded application on a single-processor system. Because every thread has to run on the same processor, everything is automatically serialized, and many race conditions and cache ping-pong problems that you may get with a true multi-processor system just evaporate. If you need your application to be portable across a range of target systems then it is important to test it on representative instances of those systems. This is the reason that I listed the processor architectures being used for testing as a consideration in section 10.2.2.

Whilst brute force testing does give us some degree of confidence in our code, it's not guaranteed to find all the problems. There is one technique that *is* guaranteed to find the problems, if you've got the time to apply it to your code and the appropriate software. I call it *Combination Simulation Testing*.

Combination Simulation Testing

That sounds a bit of a mouthful, so I'd best explain what I mean. The idea is that you run your code with a special piece of software that *simulates* the real runtime environment of the code. Many of you may be aware of software that allows you to run multiple “virtual machines” on a single physical computer, where the characteristics of the virtual machine and its hardware are emulated by the supervisor software. The idea here is similar, except rather than just emulating the system, the simulation software records the sequences of data accesses, locks and atomic operations from each thread. It then uses the rules of the

C++ memory model to repeat the run with every permitted *combination* of operations, and thus identify race conditions and deadlocks.

Whilst such exhaustive combination testing is guaranteed to find all the problems the system is designed to detect, for anything but the most trivial of programs it will take a huge amount of time, as the number of combinations increases exponentially with the number of threads and the number of operations performed by each thread. This technique is thus best reserved for fine-grained tests of individual pieces of code, rather than an entire application. The other obvious downside is that it relies on the availability of simulation software that can handle the operations used in your code.

So, we've got a technique that involves running our test many many times under “normal” conditions, but which might miss problems, and we've got a technique that involves running our test many many times under “special” conditions, but which is more likely to find any problems that exist. Are there any other options?

A third option is to use a library that detects problems as they occur in the running of the tests.

Detecting Problems Exposed by Tests with a Special Library

Whilst not providing the exhaustive checking of a combination simulation test, many problems can be identified by using a special implementation of the library synchronization primitives such as mutexes, locks and condition variables. For example, it is common to require that all accesses to a piece of shared data are done with a particular mutex locked. If you could check which mutexes were locked when the data was accessed, you could therefore verify that the appropriate mutex was indeed locked by the calling thread when the data was accessed, and report a failure if this not the case. By marking your shared data in some way you can allow the library to check this for you.

Such a library implementation could also record the sequence of locks if more than one mutex is held by a particular thread at once. If another thread locks the same mutexes in a different order then this could be recorded as a *potential* deadlock even if the test didn't actually deadlock whilst running.

Another type of special library that could be used when testing multi-threaded code is one where the implementations of the threading primitives such as mutexes and condition variables give the test writer control over which thread gets the lock when multiple threads are waiting, or which thread is notified by a **`notify_one()`** call on a condition variable. This would allow you to set up particular scenarios, and verify that your code works as expected in those scenarios.

Some of these testing facilities would have to be supplied as part of the C++ standard library implementation, whilst others can be built on top of the standard library as part of your test harness.

So far we've just been looking at the *correctness* of multithreaded code. Whilst this is the most important issue, it's not the only reason we test: it's also important to test the *performance* of multithreaded code, so let's look at that next.

10.2.5 Testing the Performance of Multithreaded Code

One of the main reasons we might choose to use concurrency in an application is to make use of the increasing prevalence of multi-core processors to improve the performance of our applications. It is therefore important to actually test your code to confirm that the performance does indeed improve, just as one would do with any other attempt at optimization.

The particular issue with using concurrency for performance is the *scalability* — we want code that runs approximately 24 times faster or processes 24 times as much data on a 24-core machine than on a single-core machine, all else being equal, not code that runs twice as fast on a dual-core machine, but is actually slower on a 24-core machine. As we've already seen in previous chapters, contention between processors for access to a data structure can have a big performance impact, and something that scales nicely with the number of processors when that number is small may well perform badly when the number of processors is much larger due to the huge increase in contention.

Consequently, when testing for the performance of multithreaded code, it is really best to check the performance on systems with as many different configurations as possible, so you get a picture of the scalability graph. At the very least, you ought to test on a single-processor system, and a system with as many processing cores as is available to you.

10.3 Summary

In this chapter we've looked at various types of concurrency-related bugs that you might encounter, from deadlocks and livelocks to data races and other problematic race conditions. We've then followed that up with techniques for locating bugs. These included issues to think about during code reviews, and guidelines for writing testable code, and how to structure tests for concurrent code. Finally, we've looked at some utility components that can help with testing.

A

Overview of New C++0x Language Features

Of course the new C++ Standard brings more than just concurrency support: there are a whole host of other language features and new libraries as well. In this appendix I'm going to give a brief overview of the new language features not directly related to concurrency that are used in the thread library and the rest of the book. Though code that uses them may be difficult to understand at first due to lack of familiarity, as the use of C++0x becomes more widespread, code making use of these features will become more common, and thus more familiar and easier to understand.

Without further ado, let's start by looking at *rvalue references*, which are used extensively by the thread library to facilitate transfer of ownership (of threads, locks or whatever) between objects.

A.1 Rvalue References

If you've been doing C++ programming for any time you'll be familiar with references; C++ references allow you to create a new name for an existing object. All accesses and modifications done through the new reference affect the original. e.g.

```
int var=42;
int& ref=var;                                                #1
ref=99;
assert(var==99);                                             #2
```

#1 create a reference to var
#2 original updated due to assignment to reference

The references that we've all been using up to now are *lvalue references* — references to lvalues. The term *lvalue* comes from C, and refers to things that can be on the left-hand side of an assignment expression — named objects, objects allocated on the stack or heap, or members of other objects. Things with a defined storage location. The term *rvalue* also comes from C, and refers to things that can occur only on the right-hand side of an assignment expression — literals and temporaries for example. lvalue references can only be bound to lvalues, not rvalues. You cannot write

```
int& i=42;                                                                    #1
```

#1 Will not compile

for example, as **42** is an rvalue. OK, it's not quite true — you have always been able to bind an rvalue to a **const** lvalue reference:

```
int const& i=42;
```

but this is a deliberate exception on the part of the standard, introduced before we had rvalue references in order to allow you can pass temporaries to functions taking references. This allows implicit conversions, so you can write things like:

```
void print(std::string const& s);
print(“hello”);                                                               #1
```

#1 create temporary std::string object

Anyway, the C++0x standard introduces *rvalue references* which bind *only* to rvalues, not to lvalues, and are declared with two ampersands rather than one:

```
int&& i=42;
int j=42;
int&& k=j;                                                                    #1
```

#1 Will not compile

You can thus use function overloading to determine whether function parameters are lvalues or rvalues by having one overload take an lvalue reference and another take an rvalue reference. This is the cornerstone of *move semantics*.

A.1.1 Move semantics

rvalues are typically temporary, and so can be freely modified: if we know that our function parameter is an rvalue then we can use it as temporary storage, or “steal” it's contents without affecting program correctness. This means that rather than *copying* the contents of an rvalue parameter we can just *move* the contents. For large dynamic structures this can save a lot of memory allocation, and provides a lot of scope for optimization. Consider a function that takes a **std::vector<int>** as a parameter, and needs to have an internal copy for modification, without touching the original. The “old” way of doing this would be to take the parameter as a **const** lvalue reference, and make the copy internally:

```
void process_copy(std::vector<int> const& vec_)
{
     std::vector<int> vec(vec_);
     vec.push_back(42);
}
```

This allows the function to take both lvalues and rvalues, but forces the copy in every case. If we overload the function with a version that takes an rvalue reference then we can avoid the copy in the rvalue case, since we know we can freely modify the original:

```
void process_copy(std::vector<int> && vec)
{
        vec.push_back(42);
    }
```

Now, if the function in question is the constructor of our class we can pilfer the innards of the rvalue and use them for our new instance. Consider the class in listing A.1. In the default constructor it allocates a large chunk of memory which is freed in the destructor.

Listing A.72: A class with a move constructor

```
    class X
    {
    private:
        int* data;
    public:
        X():
            data(new int[1000000])
        {}
        ~X()
        {
        delete data;
        }
        X(const X& other):                                             #1
            data(new int[1000000])
        {
            std::copy(other.data,other.data+1000000,data);
        }
        X(X&& other):                                                  #2
            data(other.data)
        {
            other.data=0;
        }
    };
```

Cueballs in code and text

The *copy constructor* (#1) is defined just like you might expect: allocate a new block of memory and copy the data across. However, we also have a new constructor which takes the old value by rvalue reference (#2). This is the *move constructor*. In this case we just copy the *pointer to* the data, and leave the **`other`** instance with a **`NULL`** pointer, saving ourselves a huge chunk of memory and time when creating variables from rvalues.

For class **`X`** the move constructor is just an optimization, but in some cases it makes sense to provide a move constructor even when it doesn't make sense to provide a copy constructor. For example, the whole point of **`std::unique_ptr<>`** is that each non-**`NULL`** instance is the one and only pointer to its object, so a copy constructor makes no sense. However, a move constructor allows ownership of the pointer to be transferred between instances, and permits **std::unique_ptr<>** to be used as a function return value — the pointer is *moved* rather than *copied*.

If you wish to explicitly move from a named object that you know you will no longer use, you can cast it to an rvalue either by using **static_cast<X&&>** or by calling **std::move()**:

```
X x1;
X x2=std::move(x1);
X x3=static_cast<X&&>(x2);
```

This can be beneficial when you wish to move the parameter value into a local or member variable without copying, since though an rvalue reference parameter can bind to rvalues, within the function itself it is treated as an lvalue.

```
void do_stuff(X&& x_)
{
    X a(x_);                                                          #1
    X b(std::move(x_));                                               #2
}
do_stuff(X());                                                        #3
X x;
do_stuff(x);                                                          #4
```

#1 copies
#2 moves
#3 OK, rvalue binds to rvalue reference
#4 Error, lvalue cannot bind to rvalue reference

Move semantics is used extensively in the thread library, both where copies make no semantic sense but resources can be transferred, and as an optimization to avoid expensive copies where the source is going to be destroyed anyway. We saw an example of this in section 2.2 where we used **std::move()** to transfer a **std::unique_ptr<>** instance into a newly-constructed thread, and then again in section 2.3 where we looked at transferring ownership of threads between **std::thread** instances.

None of **std::thread**, **std::unique_lock<>**, **std::unique_future<>**, **std::promise<>** or **std::packaged_task<>** can be copied, but they all have move constructors to allow the associated resource to be transferred between instances, and support their use as function return values. **std::string** and **std::vector<>** both can be copied as always, but they also have move constructors and move-assignment operators to avoid copying large quantities of data from an rvalue.

The C++ standard library never does anything with an object that has been explicitly moved into another object, except destroy it or assign **to** it (either with a copy or (more likely) a move). However, it is good practice to ensure that the invariant of the class encompasses the moved-from state. A **std::thread** instance that has been used as the source of a move is equivalent to a default-constructed **std::thread** instance, for example, and an instance of **std::string** that has been used as the source of a move will still have a valid state, though no guarantees are made as to what that state is (in terms of how long the string is or what characters it contains).

A.1.2 Rvalue References and Function Templates

There's a final nuance when you use rvalue references for parameters to a function template: if the function parameter is an rvalue reference to a template parameter, then automatic template argument type deduction deduces the type to be an lvalue reference if an lvalue is supplied, or a plain unadorned type if an rvalue is supplied. That is a bit of a mouthful, so let's look at an example. Consider the following function:

```
template<typename T>
void foo(T&& t)
{}
```

If we call it with an rvalue like below, then **T** is deduced to be the type of the value:

```
foo(42);                                                      #1
foo(3.14159);                                                 #2
foo(std::string());                                           #3
```

#1 Calls foo<int>(42)
#2 Calls foo<double>(3.14159)
#3 Calls foo<std::string>(std::string())

However, if we call **foo** with an *lvalue* then **T** is deduced to be an lvalue reference:

```
int i=42;
foo(i);                                                       #1
```

#1 Calls foo<int&>(i)

Since the function parameter is declared **T&&**, this is therefore a reference to a reference, which is treated as just the original reference type. The signature of **foo<int&>()** is thus:

```
void foo<int&>(int& t);
```

This allows a single function template to accept both lvalue and rvalue parameters, and is used by the **std::thread** constructor (section 2.1 and 2.2) so that the supplied callable object can be moved into internal storage rather than copied if the parameter is an rvalue.

A.2 Deleted Functions

Sometimes it doesn't make sense to allow a class to be copied. **std::mutex** is a prime example of this — what would it mean if you did copy a mutex? **std::unique_lock<>** is another — an instance is the one and only owner of the lock it holds. To truly copy it would mean that the copy also held the lock, which doesn't make sense. Moving ownership between instances, as described in section A.1.2 makes sense, but that's not copying. I'm sure you've met other examples.

The standard idiom for preventing copies of a class used to be to declare the copy constructor and copy assignment operator private and then not provide an implementation. This would cause a compile error if any code outside the class in question tried to copy an instance, and a link-time error (due to lack of an implementation) if any of the class's member functions or friends tried to copy an instance.

```
class no_copies
{
```

```
public:
    no_copies(){}
private:
    no_copies(no_copies const&);                                #1
    no_copies& operator=(no_copies const&);                     #1
};
no_copies a;
no_copies b(a);                                                 #2
```

#1 No implementation
#2 Won't compile

With C++0x, the committee realised that this was a common idiom, but also realised that it's a bit of a hack. The committee therefore provided a more general mechanism that can be applied in other cases too: you can declare a function as *deleted*. **no_copies** can thus be written as:

```
class no_copies
{
public:
    no_copies(){}
    no_copies(no_copies const&) = delete;
    no_copies& operator=(no_copies const&) = delete;
};
```

This is much more descriptive than the original code, and clearly expresses the intent. It also allows the compiler to give more descriptive error messages, and moves the error from link-time to compile-time if you try and perform the copy within a member function of your class.

If, as well as deleting the copy constructor and copy-assignment operator you also explicitly write a move constructor and move-assignment operator, then your class becomes move-only, the same as **std::thread** and **std::unique_lock<>**. Listing A.2 shows an example of such a move-only type.

Listing A.73: A simple move-only type

```
class move_only
{
    std::unique_ptr<my_class> data;
public:
    move_only(const move_only&) = delete;
    move_only(move_only&& other):
        data(std::move(other.data))
    {}
    move_only& operator=(const move_only&) = delete;
    move_only& operator=(move_only&& other)
    {
        data=std::move(other.data);
        return *this;
    }
};
```

```
move_only m1;
move_only m2(m1);                                                    #1
move_only m3(std::move(m1));                                         #2
```

#1 Error, copy constructor is declared deleted
#2 OK, move constructor found

Move-only objects can be passed as function parameters and returned from functions, but if you wish to move from an lvalue then you always have to be explicit and use **std::move()** or a **static_cast<T&&>**.

You can apply the "**= delete**" specifier to any function, not just copy constructors and assignment operators. This makes it clear that the function is not available. It does a bit more than that too, though — a deleted function participates in overload resolution in the normal way, and only causes a compilation error if it is selected. This can be used to remove specific overloads. For example, if your function takes a **short** parameter then you can prevent narrowing of **int** values by writing an overload that takes an **int** and declaring it deleted:

```
void foo(short);
void foo(int) = delete;
```

Any attempts to call **foo** with an **int** will now be met with a compilation error, and the caller will have to explicitly cast supplied values to **short**:

```
foo(42);                                                               #1
foo((short)42);                                                        #2
```

#1 Error, int overload declared deleted
#2 OK

A.3 Defaulted Functions

Whereas deleted functions allow you to explicitly declare that a function is not implemented, *defaulted* functions are the opposite extreme — they allow you to specify that the compiler should write the function for you, with its "default" implementation. Of course, you can only do this for functions that the compiler can auto-generate anyway: default constructors, destructors, copy-constructors and copy-assignment operators.

Why would you want to do that? There are several reasons why you might:

- In order to change the accessibility of the function. By default, the compiler-generated functions are **public**. If you wish to make them **protected** or even **private** then you must write them yourself. By declaring them as defaulted you can get the compiler to write the function **and** change the access level.

- As documentation — if the compiler-generated version is sufficient, then it might be worth explicitly declaring it as such so that when you or someone else looks at the code later it is clear that this was intended.

- In order to force the compiler to generate the function when it would not otherwise have done. This is typically done with default constructors, which are only normally compiler-generated if there are no user-defined constructors. If you need to define a custom copy constructor (for example), you can still get a compiler-generated default constructor by declaring it as defaulted.

- In order to make a destructor **virtual** whilst leaving it as compiler-generated.
- To force a particular declaration of the copy constructor, such as having it take the source parameter by a non-**const** reference rather than by a **const** reference.
- To take advantage of the special properties of the compiler-generated function which are lost if you provide an implementation. More on this in a moment.

Just as deleted functions are declared by following the declaration with "**= delete**", defaulted functions are declared by following the declaration by "**= default**". e.g.

```
class Y
{
private:
    Y() = default;                                   #1
public:
    Y(Y&) = default;                                 #2
    T& operator=(const Y&) = default;                #3
protected:
    virtual ~Y() = default;                          #4
};
```

Cueballs in code

#1 Change access of default constructor to private
#2 Change copy constructor to accept a non-const reference
#3 Explicitly declare the copy-assignment operator defaulted with the default declaration and access
#4 Change the destructor to be protected and virtual but still compiler-generated

I mentioned above that compiler-generated functions can have special properties which you cannot get from a user-defined version. The biggest difference is that a compiler-generated function can be *trivial*. This has a few consequences, including:

- objects with trivial copy constructors, trivial copy assignment operators and trivial destructors can be copied with **memcpy** or **memmove**;
- literal types used for **constexpr** functions (see section A.4) must have a trivial constructor, copy constructor and destructor;
- classes with a trivial default constructor, copy constructor, copy assignment operator and destructor can be used in a union with a user-defined constructor and destructor;

- classes with trivial copy assignment operators can be used with the **std::atomic<>** class template (see section 5.2.6) in order to provide a value of that type with atomic operations.

Of course, just declaring the function as "**= default**" doesn't make it trivial — it will only be trivial if the class also supports all the other criteria for the corresponding function to be trivial — but explicitly writing the function in user code does *prevent* it being trivial.

The second difference between classes with compiler-generated functions and user-supplied equivalents is that a class with no user-supplied constructors can be an *aggregate*, and thus can be initialized with an aggregate initializer:

```
struct aggregate
{
    aggregate() = default;
    aggregate(aggregate const&) = default;

    int a;
    double b;
};
aggregate x={42,3.141};
```

In this case, **x.a** is initialized to **42** and **x.b** is initialized to **3.141.**

The third difference between a compiler-generated function and a user-supplied equivalent is quite esoteric and only applies to the default constructor, and only to the default constructor of classes that meet certain criteria. Consider the following class:

```
struct X
{
    int a;
};
```

If you create an instance of class X without an initializer then the contained **int** (**a**) is uninitialized has an an undefined value unless the object has static storage duration, in which case it is initialized to zero.

```
X x1;                                                          #1
```

#1 x1.a has undefined value

If, on the other hand, you initialize your instance of **X** by explicitly invoking the default constructor then **a** is initialized to zero:

```
X x2=X();                                                      #1
```

#1 x1.a==0

This bizarre property also extends to base classes and members: if your class has a compiler-generated default constructor and any of your data members and base classes also have a compiler-generated default constructor then data members of those bases and members that are built-in types are also either left uninitialized or initialized to zero depending on whether or not the outer class has its default constructor explicitly invoked.

Though this rule is confusing and potentially error-prone, it does have its uses, and if you write the default constructor yourself then you lose this property: either data members like **a** are always initialized (because you specify a value or explicitly default construct), or always uninitialized (because you don't).

```
X::X():a(){}                                                     #1
X::X():a(42){}                                                   #2
X::X(){}                                                         #3
```

#1 a==0 always
#2 a==42 always
#3 a uninitialized for non-static instances of X, a==0 for static instances of X

Under normal circumstances, if you write any other constructor manually the compiler will no longer generate the default constructor for you, so if you want one you have to write it, which means you lose this bizarre initialization property. However, by explicitly declaring the constructor as defaulted you can force the compiler to generate the default constructor for you, and this property is retained:

```
X::X() = default;                                                 #1
```

#1 default initialization rules for a apply

This property is used for the atomic types (see section 5.2), which have their default constructor explicitly defaulted — their initial value is always undefined unless either (a) they have static storage duration (and thus are statically initialized to zero), or (b) you explicitly invoke the default constructor to request zero initialization, or (c) you explicitly specify a value. Note that the constructor for initialization with a value is declared **constexpr** (see section A.4) in order to allow static initialization.

A.4 constexpr functions

Integer literals such as **42** are *constant expressions*. So are simple arithmetic expressions such as **23*2-4**. You can even use **const** variables of integral type that are themselves initialized with constant expressions as part of a new constant expression:

```
const int i=23;
const int two_i=i*2;
const int four=4;
const int forty_two=two_i-four;
```

Aside from using constant expressions to create variables that can be used in other constant expressions, there's a few things you can *only* do with constant expressions:

- specify the bounds of an array:

```
int bounds=99;
int array[bounds];                                              #1
const int bounds2=99;
int array2[bounds2];                                            #2
```

#1 Error bounds is not a constant expression
#2 OK, bounds2 is a constant expression

- specify the value of a non-type template parameter:

```
template<unsigned size>
struct test
{};
test<bounds> ia;                                                     #1
test<bounds2> ia2;                                                   #2
```

#1 Error bounds is not a constant expression
#2 OK, bounds2 is a constant expression

- provide an initializer for a **static const** class data member of integral type in the class definition:

```
class X
{
    static const int the_answer=forty_two;
};
```

- provide an initializer for a built-in type or aggregate that can be used for static initialization:

```
struct my_aggregate
{
    int a;
    int b;
};
static my_aggregate ma1={forty_two,123};                               #1
int dummy=257;
static my_aggregate ma2={dummy,dummy};                                 #2
```

#1 static initialization
#2 dynamic initialization

Static initialization like this can be used to avoid order-of-initialization problems and race conditions.

None of this is new — you could do all that with the 1998 edition of the C++ standard. However, with the new standard what constitutes a *constant expression* has been extended with the introduction of the **constexpr** keyword.

The **constexpr** keyword is primarily a function modifier. If the parameter and return type of a function meet certain requirements and the body is sufficiently simple then a function can be declared **constexpr**, in which case it can be used in constant expressions. For example:

```
constexpr int square(int x)
{
    return x*x;
}
int array[square(5)];
```

In this case, **array** will have 25 entries, since **square** is declared **constexpr**. Of course, just because the function *can* be used in a constant expression doesn't mean that all uses are automatically constant expressions:

```
int dummy=4;
int array[square(dummy)];                                            #1
```

#1 error, dummy is not a constant expression

In this example, **dummy** is not a constant expression, so **square(dummy)** isn't either — it's just a normal function call — and thus cannot be used to specify the bounds of **array**.

A.4.1 constexpr and user-defined types

Up to now, all the examples have been with built-in types such as **int**. However, the new C++ standard allows constant expressions to be of any type that satisfies the requirements for a *literal type*. For a class type to be classified as a literal type:

- it must have a trivial copy constructor,
- it must have a trivial destructor,
- all non-**static** data members and base classes must be trivial types, and
- it must have either a trivial default constructor or a **constexpr** constructor other than the copy constructor.

We'll look at **constexpr** constructors shortly. For now we'll focus on classes with a trivial default constructor, such as class **CX** in listing A.3.

Listing A.74: A class with a trivial default constructor

```
class CX
{
private:
    int a;
    int b;
public:
    CX() = default;                                          #1
    CX(int a_, int b_):                                      #2
        a(a_),b(b_)
    {}
    int get_a() const
    {
        return a;
    }
    int get_b() const
    {
        return b;
    }
    int foo() const
    {
        return a+b;
    }
    };
```

Cueballs in code and text

Note that we've explicitly declared the default constructor (#1) as *defaulted* (see appendix A.3) in order to preserve it as trivial in the face of the user-defined constructor (#2). This type therefore fits all the qualifications for being a literal type, and we can use it in constant expressions. We can, for example, provide a **constexpr** function that creates new instances :

```
    constexpr CX create_cx()
    {
        return CX();
    }
```

We can also create a simple **constexpr** function that copies it's parameter:

```
    constexpr CX clone(CX val)
    {
        return val;
    }
```

But that's about all we can do — a **constexpr** function can only call other **constexpr** functions. What we *can* do though is apply **constexpr** to the member functions and constructor of **CX**:

```
    class CX
    {
    private:
        int a;
        int b;
    public:
        CX() = default;
        constexpr CX(int a_, int b_):
            a(a_),b(b_)
        {}
        constexpr int get_a() const                                        #1
        {
            return a;
        }
        constexpr int get_b()                                             #2
        {
            return b;
        }
        constexpr int foo()
        {
            return a+b;
        }
    };
```

Cueballs in code and text

Note that the **const** qualification on **get_a()** (#1) is now superfluous, since it is implied by the use of **constexpr**. **get_b()** is thus **const** even though the **const** qualification is omitted (#2). This now allows more complex **constexpr** functions such as the following:

```
constexpr CX make_cx(int a)
{
    return CX(a,1);
}
constexpr CX half_double(CX old)
{
    return CX(old.get_a()/2,old.get_b()*2);
}
constexpr int foo_squared(CX val)
{
    return square(val.foo());
}
int array[foo_squared(half_double(make_cx(10)))];                    #1
```

#1 49 elements

Interesting though this is, it's a lot of effort to go to if all we get is a fancy way of computing some array bounds or an integral constant. The key benefit of constant expressions and **constexpr** functions involving user-defined types is that objects of a literal type initialized with a constant expression are statically initialized, and so their initialization is free from race conditions and initialization order issues.

```
CX si=half_double(CX(42,19));                                        #1
```

#1 statically initialized

This covers constructors too — if the constructor is declared **constexpr** and the constructor parameters are constant expressions, then the initialization is *constant initialization* and happens as part of the static initialization phase. **This is one of the most important changes in C++0x as far as concurrency goes** — by allowing user-defined constructors that can still undergo static initialization you can avoid any race conditions over their initialization, since they are guaranteed to be initialized before any code is run.

This is particularly relevant for things like **std::mutex** (see section 3.2.1) or **std::atomic<>** (section 5.2.6) where we might want to use a global instance to synchronize access to other variables and avoid race conditions in *that* access. This would not be possible if the constructor of the mutex was subject to race conditions, so the default constructor of **std::mutex** is declared **constexpr** to ensure that mutex initialization is always done as part of the static initialization phase.

A.4.2 constexpr objects

So far we've looked at **constexpr** as applied to functions. **constexpr** can also be applied to objects. This is primarily for diagnostic purposes — it verifies that the object is initialized with a constant expression, **constexpr** constructor or aggregate initializer made of constant expressions. It also declares the object as **const**.

```
constexpr int i=45;                                        #1
constexpr std::string s(“hello”);                          #2
int foo();
constexpr int j=foo();                                     #3
```

#1 OK
#2 Error, `std::string` is not a literal type
#3 Error, `foo()` is not declared `constexpr`

A.4.3 `constexpr` function requirements

In order to declare a function as **`constexpr`** it must meet a few requirements; if it doesn't meet these requirements then declaring it **`constexpr`** is a compilation error. The requirements for a **`constexpr`** function are:

- all parameters must be of a literal type;
- the return type must be a literal type;
- the function body must consist of a single **`return`** statement;
- the expression in the **`return`** statement must qualify as a constant expression; and
- any constructor or conversion operator used to construct the return value from the expression must be **`constexpr`**.

This is pretty straightforward — basically you must be able to inline the function into a constant expression and it will still be a constant expression, and you must not modify anything. **`constexpr`** functions are basically *pure functions* with no side effects.

For **`constexpr`** class member functions there are additional requirements:

- **`constexpr`** member functions cannot be virtual; and
- the class for which the function is a member must be a literal type.

The rules are different for **`constexpr`** constructors:

- the constructor body must be empty;
- every base class must be initialized;
- every non-**`static`** data member must be initialized;
- any expressions used in the member initialization list must qualify as a constant expression;
- the constructors chosen for the initialization of the data members and base classes must be **`constexpr`** constructors; and
- any constructor or conversion operator used to construct the data members and

base classes from their corresponding initialization expression must be **constexpr**.

This is basically the same set of rules as for functions, except that there is no return value, so no **return** statement — instead the constructor initializes all the bases and data members in the member initialization list. Trivial copy constructors are implicitly **constexpr**.

A.4.4 constexpr and templates

When **constexpr** is applied to a function template, or to a member function of a class template then it is ignored if the parameters and return types of a particular instantiation of the template are not literal types. This allows you to write function templates which are **constexpr** if the type of the template parameters is appropriate, and just plain **inline** functions otherwise. e.g.

```
template<typename T>
constexpr T sum(T a,T b)
{
    return a+b;
}
constexpr int i=sum(3,42);                                          #1
std::string s=sum(std::string(“hello”),std::string(“ world”));      #2
```

Cueballs

#1 OK, sum<int> is constexpr
#2 Also OK, but sum<std::string> is not constexpr

Of course, the function must satisfy all the other requirements for a **constexpr** function — you can't declare a function with multiple statements **constexpr** just because it is a function template, that is still a compilation error.

A.5 Lambda Functions

Lambda functions are one of the most exciting features of the C++0x standard, as they have the potential to greatly simplify code, and eliminate much of the boilerplate associated with writing callable objects. The C++0x lambda function syntax allows a function to be defined at the point where it is needed in another expression. This works well for things like predicates provided to the wait functions of **std::condition_variable** (as in the example in section 4.1.1), as it allows the semantics to be quickly expressed in terms of the accessible variables rather than capturing the necessary state in the member variables of a class with a function call operator.

At its simplest, a *lambda expression* defines a self-contained function that takes no parameters, and relies only on global variables and functions. It doesn't even have to return a value. Such a lambda expression is a series of statements enclosed in braces, prefixed with square brackets (the *lambda-introducer*):

```
[]{                                        #1
    do_stuff();
    do_more_stuff();
}();                                       #2
```

#1 Start the lambda expression with `[]`
#2 Finish the lambda expression, and call it with no parameters

In this example, the lambda expression is called by following it with parentheses, but this is unusual. For one thing, if you are going to call it directly you could usually just do away with the lambda and write the statements directly in the source. It is more common to pass it as a parameter to a function template that takes a callable object as one of its parameters, in which case it likely needs to take parameters or return a value or both. If you need to take parameters, this can be done by following the lambda introducer with a parameter list just like for a normal function. For example the following code writes all the elements of the vector to **std::cout** separated by newlines:

```
std::vector<int> data=make_data();
std::for_each(data.begin(),data.end(),[](int i){std::cout<<i<<"\n";});
```

Return values are almost as easy. If your lambda function body consists of a single **return** statement, then the return type of the lambda is the type of the expression being returned. For example, you might use a simple lambda like this to wait for a flag to be set with a **std::condition_variable** (see section 4.1.1) as in listing A.4.

Listing A.75: A simple lambda with a deduced return type

```
std::condition_variable cond;
bool data_ready;
std::mutex m;

void wait_for_data()
{
    std::unique_lock<std::mutex> lk(m);
    cond.wait(lk,[]{return data_ready;});                          #1
}
```

Cueballs in code and text

The return type of the lambda passed to **cond.wait()** (#1) is deduced from the type of **data_ready**, and is thus **bool**. Whenever the condition variable wakes from waiting it then calls the lambda with the mutex locked, and only returns from the call to **wait()** once **data_ready** is **true**.

What if you can't write your lambda body as a single **return** statement? In that case you have to specify the return type explicitly. You can do this even if your body is a single

return statement, but you *have* to do it if your lambda body is more complex. The return type is specified by following the lambda parameter list with an arrow (**->**) and the return type. If your lambda doesn't take any parameters, you must still include the (empty) parameter list in order to specify the return value explicitly. Our condition variable predicate can thus be written:

```
cond.wait(lk,[]()->bool{return data_ready;});
```

By specifying the return type we can expand the lambda to log messages, or do some more complex processing:

```
cond.wait(lk,[]()->bool{
    if(data_ready)
    {
        std::cout<<”Data ready”<<std::endl;
        return true;
    }
    else
    {
        std::cout<<”Data not ready, resuming wait”<<std::endl;
        return false;
    }
});
```

Though simple lambdas like this are powerful, and can simplify code quite a lot, the real power of lambdas comes when they capture local variables.

A.5.1 Lambda functions that reference local variables

Lambda functions with a *lambda introducer* of **[]** cannot reference any local variables from the containing scope: they can only use global variables and anything passed in as a parameter. If you wish to access a local variable, you need to *capture* it. The simplest way to do this is to capture the entire set of variables within the local scope by using a lambda introducer of **[=]**. That's all there is to it — your lambda can now access *copies* of the local variables at the time the lambda was created.

To see this in action, consider the following simple function:

```
std::function<int(int)> make_offseter(int offset)
{
   return [=](int j){return offset+j;};
}
```

Every call to **make_offseter** returns a new lambda function object through the **std::function<>** function wrapper. This returned function adds the supplied offset to any parameter supplied. For example,

```
int main()
{
    std::function<int(int)> offset_42=make_offseter(42);
    std::function<int(int)> offset_123=make_offseter(123);
    std::cout<<offset_42(12)<<”,“<<offset_123(12)<<std::endl;
```

```
        std::cout<<offset_42(12)<<”,“<<offset_123(12)<<std::endl;
    }
```

will write out **54,135** twice since the function returned from the first call to **make_offseter** always adds 42 to the supplied argument, whereas the function returned from the second call to **make_offseter** always adds 123 to the supplied argument.

This is the safest form of local variable capture: everything is copied, so you can return the lambda and call it outside the scope of the original function. It's not the only choice though: you can choose to capture everything by reference instead. In this case it is undefined behaviour to call the lambda once the variables it references have been destroyed by exiting the function or block scope to which they belong, just like it is undefined behaviour to reference a variable that has already been destroyed in any other circumstance.

A lambda function that captures by all the local variables by reference is introduced using **[&]**, as in the following example:

```
    int main()
    {
        int offset=42;                                                     #2
        std::function<int(int)> offset_a=[&](int j){return offset+j;};    #1
        offset=123;                                                        #3
        std::function<int(int)> offset_b=[&](int j){return offset+j;};    #4
        std::cout<<offset_a(12)<<”,”<<offset_b(12)<<std::endl;             #5
        offset=99;                                                         #7
        std::cout<<offset_a(12)<<”,”<<offset_b(12)<<std::endl;             #6
    }
```

Cueballs in code and text

Whereas in our **make_offseter** function from the previous example we used the **[=]** lambda introducer to capture a copy of the offset, our **offset_a** function in this example uses the **[&]** lambda introducer to capture **offset** by reference (#1). It doesn't matter that the initial value of **offset** is 42 (#2): the result of calling **offset_a(12)** will always depend on the current value of **offset**. Even though the value of **offset** is then changed to 123 (#3) before we produce our second (identical) lambda function **offset_b** (#4), this second lambda again captures by reference, so the result depends on the current value of offset.

Now, when we print the first line of output (#5), **offset** is still 123, so the output is **135,135**. However, at the second line of output (#6), **offset** has been changed to 99 (#7), so this time the output is **111,111** — both **offset_a** and **offset_b** add the current value of **offset** (99) to the supplied argument (12).

Now, C++ being C++ you're not stuck with these all-or-nothing options: you can choose to capture some variables by copy and some by reference, and you can choose to capture only those variables you explicitly say to just by tweaking the lambda introducer. If you wish to *copy* all the used variables except for one or two, you can use the **[=]** form of the lambda introducer, but follow the equals sign with a list of variables to capture by reference preceded with ampersands. The following example will thus print **1239**, as **i** is copied into the lambda, but **j** and **k** are captured by reference.

```
    int main()
    {
        int i=1234,j=5678,k=9;
        std::function<int()> f=[=,&j,&k]{return i+j+k;};
        i=1;
        j=2;
        k=3;
        std::cout<<f()<<std::endl;
    }
```

Alternatively, you can capture by reference by default, but capture a specific subset of variables by copying. In this case you use the **[&]** form of the lambda introducer, but follow the ampersand with a list of variables to capture by copy. The following example thus prints **5688** since **i** is captured by reference, but **j** and **k** are copied.

```
    int main()
    {
        int i=1234,j=5678,k=9;
        std::function<int()> f=[&,j,k]{return i+j+k;};
        i=1;
        j=2;
        k=3;
        std::cout<<f()<<std::endl;
    }
```

If you only want to capture the named variables, then you can omit the leading **=,** or **&,** and just list the variables to be captured, prefixing them with an ampersand to capture by reference rather than copy. The following code will thus print **5682** as **i** and **k** are captured by reference, but **j** is copied:

```
    int main()
    {
        int i=1234,j=5678,k=9;
        std::function<int()> f=[&i,j,&k]{return i+j+k;};
        i=1;
        j=2;
        k=3;
        std::cout<<f()<<std::endl;
    }
```

This final variant allows you to ensure that only the intended variables are being captured, since any reference to a local variable not in the capture list will cause a compilation error. If you opt for this option, then you have to be careful when accessing class members if the function containing the lambda is a member function — class members cannot be captured directly; if you wish to access class members from your lambda you have to capture the **this** pointer by adding it to the capture list. In the following example, the lambda captures **this** to allow access to the **some_data** class member:

```
struct X
{
    int some_data;
    void foo(std::vector<int>& vec)
    {
        std::for_each(vec.begin(),vec.end(),
            [this](int& i){i+=some_data;});
    }
};
```

In the context of concurrency, lambdas are most useful as predicates for **std::condition_variable::wait()** (section 4.1.1), and with **std::packaged_task<>** (section 4.2.1) or thread pools for packaging small tasks. They can also be passed to the **std::thread** constructor as a thread function (section 2.1.1), and as the function when using parallel algorithms such as **parallel_for_each()** from section 8.5.1.

A.6 Variadic Templates

Variadic templates are templates with a variable number of parameters. Just as you've always been able to have variadic functions such as **printf** which take a variable number of parameters, you can now have variadic templates which have a variable number of *template* parameters. Variadic templates are used throughout the C++ thread library. For example, the **std::thread** constructor for starting a thread (section 2.1.1) is a variadic function template, and **std::packaged_task<>** (section 4.2.1) is a variadic class template. From a user's point of view, it is enough to know that the template takes an unbounded number of parameters, but if you want to write such a template, or are just interested in how it all works then you need to know the details.

Just as variadic functions are declared with an ellipsis (**...**) in the function parameter list, variadic templates are declared with an ellipsis in the template parameter list:

```
template<typename ... ParameterPack>
class my_template
{};
```

You can use variadic templates for a partial specialization of a template too, even if the primary template isn't variadic. For example, the primary template for **std::packaged_task<>** (section 4.2.1) is just a simple template with a single template parameter:

```
template<typename FunctionType>
class packaged_task;
```

However, this primary template is never defined anywhere — it's just a place holder for the partial specialization:

```
template<typename ReturnType,typename ... Args>
class packaged_task<ReturnType(Args...)>;
```

It is this partial specialization that contains the real definition of the class — we saw in chapter 4 that you can write **std::packaged_task<int(std::string,double)>** to declare a task that takes a **std::string** and a **double** as parameters when you call it, and which provides the result through a **std::unique_future<int>**.

This declaration shows two additional features of variadic templates. The first feature is relatively simple: you can normal template parameters (such as **ReturnType**) as well as variadic ones (**Args**) in the same declaration. The second feature demonstrated is the use of **Args...** in the template argument list of the specialization to show that the types that form of **Args** when the template is instantiated are to be listed here. Actually, because this is a partial specialization it works as a pattern match — the types that occur in this context in the actual instantiation are captured as **Args**. The variadic parameter **Args** is called a *parameter pack*, and the use of **Args...** is called a *pack expansion*.

Just like with variadic functions, the variadic part may be an empty list or may have many entries. For example, with **std::packaged_task<my_class()>** the **ReturnType** parameter is **my_class**, and the **Args** parameter pack is empty, whereas with **std::packaged_task<void(int,double,my_class&,std::string*)>** the **ReturnType** is **void**, and **Args** is the list **int**, **double**, **my_class&**, **std::string***.

A.6.1 Expanding the parameter pack

The power of variadic templates comes from what you can do with that pack expansion: you aren't limited to just expanding the list of types as-is. First off, you can use a pack expansion directly anywhere a list of types is required, such as in the argument list for another template:

```
template<typename ... Params>
struct dummy
{
    std::tuple<Params...> data;
};
```

In this case the single member variable **data** is an instantiation of **std::tuple<>** containing all the types specified, so **dummy<int,double,char>** has a member of type **std::tuple<int,double,char>**. You can of course combine pack expansions with normal types:

```
template<typename ... Params>
struct dummy2
{
    std::tuple<std::string,Params...> data;
};
```

This time, the tuple has an additional (first) member of type **std::string**. The really nifty part is that you can create a pattern with the pack expansion, which is then copied for each element in the expansion. You do this by putting the **...** that marks the pack expansion at the end of the pattern. For example, rather than just creating a tuple of the elements supplied in our parameter pack, we can create a tuple of pointers to the elements, or even a tuple of **std::unique_ptr<>**'s to our elements:

```
template<typename ... Params>
struct dummy3
{
    std::tuple<Params* ...> pointers;
    std::tuple<std::unique_ptr<Params> ...> unique_pointers;
};
```

The type expression can be as complex as you like, provided the parameter pack occurs in the type expression, and provided it is followed by the **...** that mark the expansion. When the parameter pack is expanded, for each entry in the pack that type is substituted into the type expression to generate the corresponding entry in the resulting list. Thus, if your parameter pack **Params** contains the types **int,int,char** then the expansion of **std::tuple<std::pair<std::unique_ptr<Params>,double> ... >** is **std::tuple<std::pair<std::unique_ptr<int>,double>, std::pair<std::unique_ptr<int>,double>, std::pair<std::unique_ptr<char>,double> >**. If the pack expansion is used as a template argument list, that template doesn't have to have variadic parameters, but if it doesn't then the size of the pack must exactly match the number of template parameters required:

```
template<typename ... Types>
struct dummy4
{
    std::pair<Types...> data;
};
dummy4<int,char> a;                                                  #1
dummy4<int> b;                                                       #2
dummy4<int,int,int> c;                                               #3
```

#1 OK, data is std::pair<int,char>
#2 Error, std::pair<int> is lacking a second type
#3 Error, std::pair<int,int,int> has too many types

The second thing you can do with a pack expansion is use it to declare a list of function parameters:

```
template<typename ... Args>
void foo(Args ... args);
```

This creates a new parameter pack **args** which is a list of the function parameters rather than a list of types, which you can expand with **...** as before. Now, you can use a pattern with the pack expansion for declaring the function parameters, just as you can use a pattern when you expand the pack elsewhere. For example, this is used by the **std::thread** constructor to take all the function arguments by rvalue-reference (see section A.1):

```
template<typename CallableType,typename ... Args>
thread::thread(CallableType&& func,Args&& ... args);
```

The function parameter pack can then be used to call another function, by specifying the pack expansion in the argument list of the called function. Just as with the type expansions, you can use a pattern for each expression in the resulting argument list. For example, one common idiom with rvalue references is to use **std::forward<>** to preserve the rvalue-ness of the supplied function arguments:

```
template<typename ... ArgTypes>
void bar(ArgTypes&& ... args)
{
    foo(std::forward<ArgTypes>(args)...);
}
```

Note that in this case, the pack expansion contains both the type pack **ArgTypes** and the function parameter pack **args**, and the ellipsis follows the whole expression. If you call **bar** like this:

```
int i;
bar(i,3.141,std::string(“hello”));
```

Then the expansion becomes:

```
template<>
void bar<int&,double,std::string>(
    int& args_1,
    double&& args_2,
    std::string&& args_3)
{
    foo(std::forward<int&>(args_1),
        std::forward<double>(args_2),
        std::forward<std::string>(args_3));
}
```

which correctly passes the first argument on to **foo** as an lvalue reference, whilst passing the others as rvalue references.

The final thing you can do with a parameter pack is find its size with the **sizeof...** operator. This is quite simple: **sizeof...(p)** is the number of elements in the parameter pack **p**. It doesn't matter whether this is a type parameter pack or a function argument parameter pack, the result is the same. This is probably the only case where you can use a parameter pack and not follow it with an ellipsis — the ellipsis is already part of the **sizeof...** operator. The following function returns the number of arguments supplied to it:

```
template<typename ... Args>
unsigned count_args(Args ... args)
{
    return sizeof... (Args);
}
```

Just as with the normal **sizeof** operator, the result of **sizeof...** is a constant expression, so can be used for specifying array bounds and so forth.

A.7 Summary

This appendix has only scratched the surface of the new language features introduced with the C++0x standard, as we've only looked at those features that actively affect the usage of the thread library. Other new language features include automatic variable type deduction, static assertions, strongly-typed enumerations, delegating constructors, Unicode support, template aliases, and a new uniform initialization sequence, along with a host of smaller changes. Describing all the new features in detail is outside the scope of this book — it would probably require a book in itself. The best overview of the entire standard is probably Bjarne Stroustrup's C++0x FAQ [Stroustup2009].

Hopefully the brief introduction to the new features covered in this appendix has provided enough depth to show how they relate to the thread library, and to enable you to write and understand multithreaded code that uses these new features. Though this appendix should provide enough depth for simple uses of the features covered, this is still only a brief introduction and not a complete reference or tutorial for the use of these features. If you intend to make extensive use of them then I would recommend acquiring such a reference or tutorial in order to gain the most benefit from them.