



C++ Concurrency in Action

Practical Multithreading

Anthony Williams

MEAP

Unedited Draft

 MANNING



**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

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

Please post comments or corrections to the Author online forum at
<http://www.manningsandbox.com/forum.jspa?forumID=437>

C++ Concurrency In Action: Practical Multithreading

Chapter 1: Introduction

Chapter 2: Using Concurrency

Chapter 3: Managing Threads

Chapter 4: Sharing Data

Chapter 5: Synchronizing Concurrent Operations

Chapter 6: Memory Model Details

Chapter 7: Writing Concurrent Containers and Parallel Algorithms in C++

Chapter 8: High Level Thread Management

Chapter 9: Testing Concurrent Programs

Appendix A: New features of the C++ language used by the thread library

Appendix B: A simple message-passing subsystem

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 for writing multi-threaded applications in the library. 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 concurrent programs in C++ using multiple threads, 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 have been common place for many years, and high-end server machines with multiple processors 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

Please post comments or corrections to the Author online forum at

<http://www.manningsandbox.com/forum.jspa?forumID=437>

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 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.

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 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 tasks to do, each divided into ten equally-sized chunks. On a dual-core machine, 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.

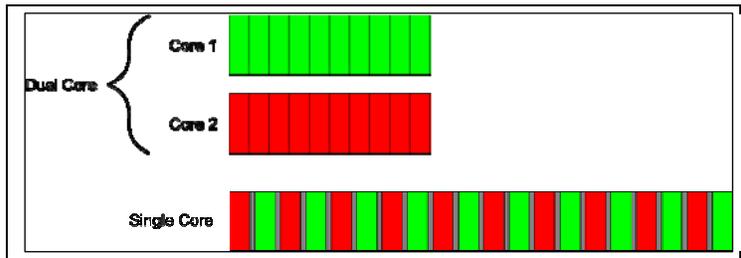


Figure 1.1 Two approaches to concurrency: parallel execution on a dual-core machine vs task-switching on a single core machine.

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.

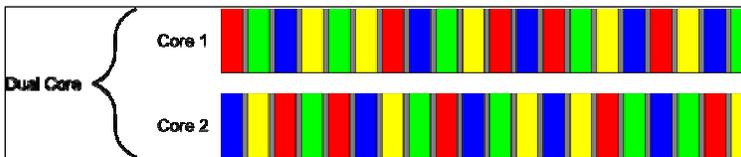


Figure 1.2 Task switching with two cores

All the techniques, functions and classes covered in this book can be used whether your 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 7, where I discuss the issues involved with writing parallel algorithms in C++.

Please post comments or corrections to the Author online forum at

<http://www.manningsandbox.com/forum.jspa?forumID=437>

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. 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.

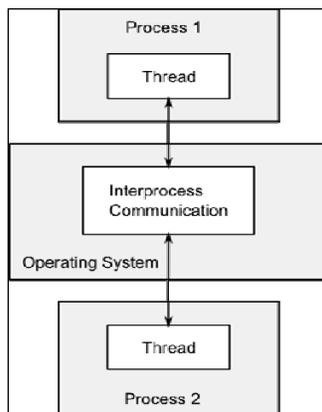


Figure 1.3 Communication between a pair of processes running concurrently

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.

Please post comments or corrections to the Author online forum at

<http://www.manningsandbox.com/forum.jspa?forumID=437>

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.

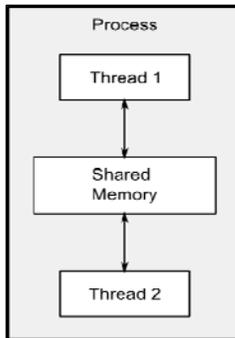


Figure 1.4 Communication between a pair of threads running concurrently in a single process

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 are covered in Chapter 4, where we shall see that the problems are not insurmountable, provided suitable care is taken when writing the code.

The low overhead associated with communicating between multiple threads within a process compared to multiple single-threaded processes means that this is the favoured approach to concurrency in mainstream languages including C++. 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

Please post comments or corrections to the Author online forum at

<http://www.manningsandbox.com/forum.jspa?forumID=437>

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. 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

Please post comments or corrections to the Author online forum at

<http://www.manningsandbox.com/forum.jspa?forumID=437>

that aren't embarrassingly parallel, you might be able to divide the algorithm into a fixed (and therefore not scalable) number of parallel tasks, for example by making use of a pipeline.

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, 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.

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 3).

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 measurable gain.

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 with a compiler that adheres strictly 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 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 3), protecting shared data (see chapter 4), synchronizing operations between threads (see chapter 5) and low-level atomic operations (see chapter 6).

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

Please post comments or corrections to the Author online forum at <http://www.manningsandbox.com/forum.jspa?forumID=437>

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 actually been a two-way flow, and the Boost thread library has itself changed to match the C++ Standard, 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, thus making code 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 comes with a performance cost: there is more code to run, after all. 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. 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 very rare cases, 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

Please post comments or corrections to the Author online forum at <http://www.manningsandbox.com/forum.jspa?forumID=437>

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 4. 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"<<std::endl;
}
```

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"<<std::endl;
}

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.

Please post comments or corrections to the Author online forum at <http://www.manningsandbox.com/forum.jspa?forumID=437>

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 3, 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 get stuck into some more detailed examples of using concurrency in C++.