C++ Concurrency IN ACTION

Practical Multithreading

Anthony Williams

MANNING

1 Hello, world of concurrency in C++!

This chapter covers

- What is meant by concurrency and multithreading
- Why you might want to use concurrency and multithreading in your applications
- Some of the history of the support for concurrency in C++
- What a simple multithreaded C++ program looks like

These are exciting times for C++ users. Thirteen 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++11 or C++0x) was published in 2011 and brings 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++11 Standard is the support of multithreaded programs. For the first time, the C++ Standard will acknowledge the existence of multithreaded applications in the language and provide components in the library for writing multithreaded applications. This will make it possible to write

multithreaded C++ programs without relying on platform-specific extensions and thus allow writing portable multithreaded code with guaranteed behavior. It also comes at a time when programmers are increasingly looking to concurrency in general, and multithreaded programming in particular, 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 multithreading and why you would want to use concurrency in your applications. After a quick detour into why you might *not* want to use it in your applications, I'll give an overview of the concurrency support in C++, and I'll round off this chapter with a simple example of C++ concurrency in action. Readers experienced with developing multithreaded applications may wish to skip the early sections. In subsequent chapters I'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 C++ Standard Library facilities for multithreading and concurrency.

So, what do I mean by *concurrency* and *multithreading*?

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 while 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, or one after the other. It isn't a new phenomenon: multitasking operating systems that allow a single computer to run multiple applications at the same time through task switching have been commonplace 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 it 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 the tasks are happening concurrently. This is called *task switching*. We still talk about *concurrency* with such systems; because 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 to both the user and the applications themselves. Because there is only an *illusion* of concurrency, the

behavior 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 on processors with more than one core on a single chip (multicore 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 tasks to do, each divided into 10 equal-size chunks. On a dual-core machine (which 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. But they are also spaced out a bit (in the diagram this is shown by the gray bars separating the chunks being thicker than the separator bars shown for the dual-core machine); 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 from executing any instructions, causing further delay.

Though the availability of concurrency in the hardware is most obvious with multiprocessor or multicore 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's 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

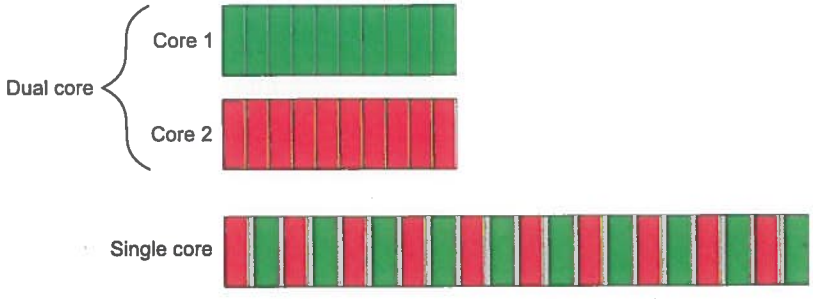


Figure 1.1 Two approaches to concurrency: parallel execution on a dual-core machine versus task switching on a single-core machine

running, performing background operations, even when the computer is nominally idle. It's 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 among four tasks on a dual-core machine, again for an idealized scenario with the tasks divided neatly into equal-size chunks. In practice, many issues will make the divisions uneven and the scheduling irregular. Some of these issues are covered in chapter 8 when we look at factors affecting the performance of concurrent code.

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 on a machine with many multicore processors and are not affected by whether the concurrency is achieved through task switching or by genuine hardware concurrency. But 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 around and talking to each other, they have to use the phone or email or get up and walk to each other's office. Also, you have the overhead of two offices to manage and multiple copies of reference manuals to purchase.

Now imagine that you move your developers into the same office. They can now talk to each other freely to discuss the design of the application, and they can easily draw diagrams on paper or on a whiteboard to help with design ideas or explanations. You now have only 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 organizing 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 their own office, and the second approach is to have multiple threads in a single process, which is like having two developers in the same office.

Figure 1.2 Task switching of four tasks on two cores

You can combine these in an arbitrary fashion and have multiple processes, some of which are multithreaded 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 that are run at the same time, much as you can run your web browser and word processor at the same time. These separate processes can then pass messages to each other through all the normal interprocess communication channels (signals, sockets, files, pipes, and so on), as shown in figure 1.3. One downside is that such communication between processes is often either complicated to set up or slow or both, because operating systems typically provide a lot of protection between processes to avoid one process accidentally modifying data belonging to another process. Another downside is that there's an inherent overhead in running multiple processes: it takes time to start a process, the operating system must devote internal resources to managing the process, and so forth.

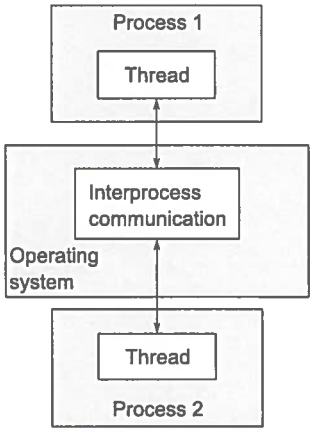


Figure 1.3 Communication between a pair of processes running concurrently

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 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 much like lightweight processes: each thread runs independently of the others, and each thread may run a different sequence of instructions. But all threads in a process share the same address space, and most of the data can be accessed directly from all threads—global variables remain global, and pointers or references to objects or data can be passed around among threads. Although it's often possible to share memory among processes, this is complicated to set up and often hard to manage, because memory addresses of the same data aren't necessarily the same in different processes. Figure 1.4 shows two threads within a process communicating through shared memory.

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

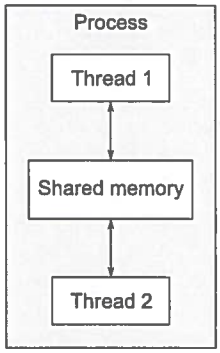


Figure 1.4 Communication between a pair of threads running concurrently in a single process

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 favored approach to concurrency in mainstream languages including C++, despite the potential problems arising from the shared memory. In addition, the C++ Standard doesn't 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 multithreading for concurrency, and future references to concurrency assume that this is achieved by using multiple threads.

Having clarified what we mean by concurrency, let's now look at why you would use concurrency in your 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 that they're 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, you can make your programs easier to understand and test, and thus less likely to contain bugs. You 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 you 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 multithreading 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. 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, because 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 while the request is conveyed to the thread doing the work. Similarly, separate threads are often used to run tasks that 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, because 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, because the division into threads is based on the conceptual design rather than an attempt to increase throughput.

1.2.2 Using concurrency for performance

Multiprocessor systems have existed for decades, but until recently they were mostly found only in supercomputers, mainframes, and large server systems. But chip manufacturers have increasingly been favoring multicore designs with 2, 4, 16, or more processors on a single chip over better performance with a single core. Consequently, multicore desktop computers, and even multicore 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."[1] *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.* Although this sounds straightforward, it can be

[1] "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Herb Sutter, *Dr. Dobb's Journal*, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.

quite a complex process, because 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 while 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 approach is called *data parallelism.*

Algorithms that 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 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 the adage, "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 2 or 10 or 20, as appropriate. Although 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 won't 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's just as important to know *when not* to use concurrency as it is to know *when* to use it. Fundamentally, the 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's a direct intellectual cost to writing and maintaining multithreaded 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 multithreaded code, don't use concurrency.

Also, the performance gain might not be as large as expected; there's an inherent overhead associated with launching a thread, because 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, 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, because each thread requires a separate stack space. This is particularly a problem for 32-bit processes with a flat architecture where there's a 4 GB limit in the available address space: if each thread has a 1 MB 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. Although 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 launches 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 that 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're trying to achieve the best possible performance of the system, it's 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's only worth doing for those performance-critical parts of the application where there's the potential for measurable gain. Of course, if the potential for performance gains is only secondary to clarity of design or separation of concerns, it may still be worth using a multithreaded 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 "multithreading Monday," what does that mean for C++ programmers?

1.3 Concurrency and multithreading in C++

Standardized support for concurrency through multithreading is a new thing for C++. It's only with the upcoming C++11 Standard that you'll be able to write multithreaded 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.

2 Managing threads

This chapter covers

- Starting threads, and various ways of specifying code to run on a new thread
- Waiting for a thread to finish versus leaving it to run
- Uniquely identifying 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, I'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's 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()`. Your program can then launch additional threads that 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 exits. As you'll see, if you have a `std::thread` object for a thread, you can wait for it to finish; but first you have to start it, so let's look at launching threads.

2.1.1 Launching a thread

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

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

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

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

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

One thing to consider when passing a function object to the thread constructor is to avoid what is dubbed "C++'s most vexing parse." If you pass a temporary rather

than a named variable, then the syntax can be the same as that of a function declaration, in which case the compiler interprets it as such, rather than an object definition. For example,

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

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

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

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

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

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

Once you've started your thread, you need to explicitly decide whether to wait for it to finish (by joining with it—see section 2.1.2) or leave it to run on its own (by detaching it—see section 2.1.3). If you don't decide before the `std::thread` object is destroyed, then your program is terminated (the `std::thread` destructor calls `std::terminate()`). It's therefore imperative that you ensure that the thread is correctly joined or detached, even in the presence of exceptions. See section 2.1.3 for a technique to handle this scenario. Note that you only have to make this decision before the `std::thread` object is destroyed—the thread itself may well have finished long before you join with it or detach it, and if you detach it, then the thread may continue running long after the `std::thread` object is destroyed.

If you don't wait for your thread to finish, then you need to ensure that the data accessed by the thread is valid until the thread has finished with it. This isn't a new problem—even in single-threaded code it is undefined behavior to access an object after it's been destroyed—but the use of threads provides an additional opportunity to encounter such lifetime issues.

One situation in which you can encounter such problems is when the thread function holds pointers or references to local variables and the thread hasn't

finished when the function exits. The following listing shows an example of just such a scenario.

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

```
struct func
{
    int& i;

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

    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};

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

❶ Potential access to dangling reference

❷ Don't wait for thread to finish

❸ New thread might still be running

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

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

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

2.1.2 Waiting for a thread to complete

If you need to wait for a thread to complete, you can do this by calling `join()` on the associated `std::thread` instance. In the case of listing 2.1, replacing the call to `my_thread.detach()` before the closing brace of the function body with a call to `my_thread.join()`

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, because the first thread wouldn't be doing anything useful in the meantime, but in real code 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 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 to check whether a thread is finished, or to wait only a certain period of time, then you have to use alternative mechanisms such as condition variables and futures, which we'll look at in chapter 4. 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 isn't associated with any thread. This means that you can call `join()` only once for a given thread; once you've called `join()`, the `std::thread` object is no longer joinable, and `joinable()` will return `false`.

2.1.3 Waiting in exceptional circumstances

As mentioned earlier, you need to ensure that you've called either `join()` or `detach()` before a `std::thread` object is destroyed. If you're detaching a thread, you can usually call `detach()` immediately after the thread has been started, so this isn't a problem. But if you're intending to wait for the thread, you need to pick carefully the place in the code where you call `join()`. This means that the call to `join()` is liable to be skipped if an exception is thrown after the thread has been started but before the call to `join()`.

To avoid your application being terminated when an exception is thrown, you therefore need to make a decision on what to do in this case. In general, if you were intending to call `join()` in the non-exceptional case, you also need to call `join()` in the presence of an exception to avoid accidental lifetime problems. The next listing shows some simple code that does just that.

Listing 2.2 Waiting for a thread to finish

```
struct func;                                    <-- See definition in listing 2.1

void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();                               <-- (1)
```

3 Sharing data between threads

This chapter covers

- Problems with sharing data between threads
- Protecting data with mutexes
- Alternative facilities for protecting shared data

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

Imagine for a moment that you're sharing an apartment 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 roommate 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 have 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 a cake. Furthermore, we all know the frustration of sharing a space and getting halfway through a task only to find that someone has borrowed something we need or changed something from the way we left 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. Incorrect use of shared data is one of the biggest causes of concurrency-related bugs, and the consequences can be far worse than sausage-flavored 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's no problem, because the data read by one thread is unaffected by whether or not another thread is reading the same data.* However, if data is shared between threads, and one or more threads start modifying the data, there's a lot of potential for trouble. In this case, you must take care to ensure that everything works out OK.

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

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

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

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

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

The simplest potential problem with modifying data that's 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 while another is removing a node, it's 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 skip the node being

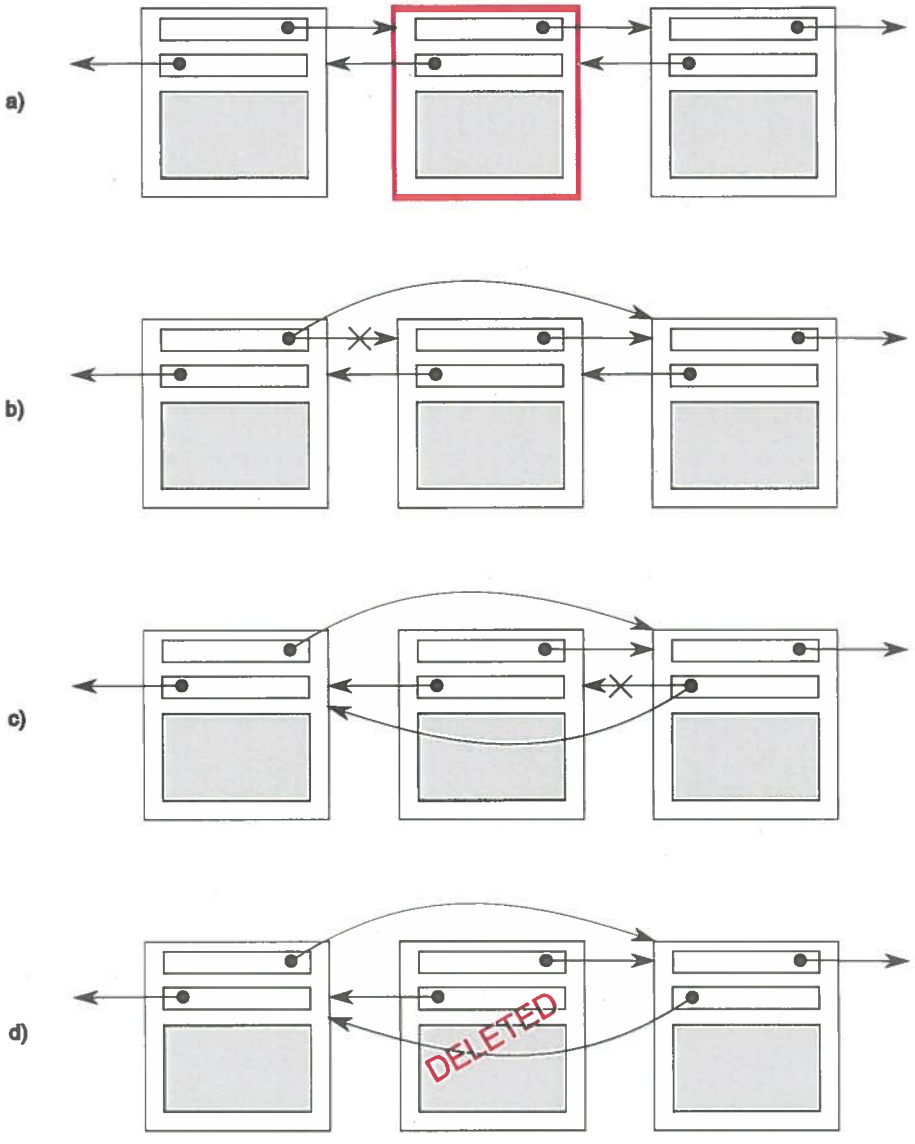


Figure 3.1 Deleting a node from a doubly linked list

deleted. On the other hand, if the second thread is trying to delete the rightmost 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 movie at the cinema. If it's a big cinema, multiple cashiers will be 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 movie as you are, which seats are available for you to choose from depends on whether the

other person actually books first or you do. If there are 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 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's when the race condition leads to broken invariants that there's a problem, such as with the doubly linked list example just mentioned. When talking about concurrency, the term *race condition* is usually used to mean a *problematic* race condition; benign race conditions aren't 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 because of concurrent modification to a single object (see section 5.1.2 for details); data races cause the dreaded *undefined behavior*.

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. 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, 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 load 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's almost inevitable that such problems will show up at the most inconvenient time. Because race conditions are generally timing sensitive, they can often disappear entirely when the application is run under the debugger, because the debugger affects the timing of the program, even if only slightly.

If you're writing multithreaded programs, race conditions can easily be the bane of your life; a great deal 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 either haven't started or have completed. The C++ Standard Library provides several such mechanisms, which are described in this chapter.

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

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

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

3.2 Protecting shared data with mutexes

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

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

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

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

Listing 3.1 Protecting a list with a mutex

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

std::list<int> some_list;                          <-(1)
std::mutex some_mutex;                             <-(2)

void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex);     <-(3)
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{                                                      (4)
    std::lock_guard<std::mutex> guard(some_mutex);     <-'
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

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

Although there are occasions where this use of global variables is appropriate, in the majority of cases it's common to group the mutex and the protected data together in a class rather than use global variables. This is a standard application of object-oriented design rules: by putting them in a class, you're clearly marking them 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, 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, because 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 there's any access to the protected data and that there are no backdoors.

3.2.2 Structuring code for protecting shared data

As you'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's also important to check that they don't pass such pointers or references *in* to functions they call that aren't 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 that are supplied at runtime via a function argument or other means, as in the next listing.

Listing 3.2 Accidentally passing out a reference to protected data

```
class some_data
{
    int a;
    std::string b;
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);
    }
};
```

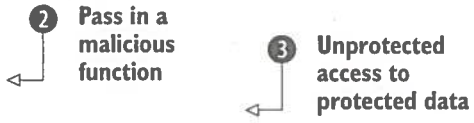
1 **Pass "protected" data to user-supplied function**

```
some_data* unprotected;

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

data_wrapper x;

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



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` ❶ means that `foo` can pass in `malicious_function` to bypass the protection ❷ and then call `do_something()` without the mutex being locked ❸.

Fundamentally, the problem with this code is that it hasn't done what you set out to do: mark all the pieces of code that access the data structure as *mutually exclusive*. In this case, it missed the code in `foo()` that calls `unprotected->do_something()`. Unfortunately, this part of the problem isn't something the C++ Thread Library can help you with; it's up to you as programmers to lock the right mutex to protect your data. On the upside, you have a guideline to follow, which will help you 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.*

Although this is a common mistake when trying to use mutexes to protect shared data, it's far from the only potential pitfall. As you'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 still have to ensure that the appropriate data is protected. Consider the doubly linked list example again. In order for a thread to safely delete a node, you need to ensure that you're preventing concurrent accesses to three nodes: the node being deleted and the nodes on either side. If you protected accesses to the pointers of each node individually, you'd be no better off than with code that used no mutexes, because 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 that protects the entire list, as in listing 3.1.

Just because individual operations on the list are safe, you'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 five things you can do to a `std::stack`:

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, because there's far 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 previously, sometimes the right thing to do is increase the granularity of the data covered by the mutexes, so that only one mutex needs to be locked. However, sometimes that's 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 that 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 that you have 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 have two small children, both of whom like playing with it. If one of them gets both the drum and the drumstick, that child can merrily play the drum until tiring of it. 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 them at the same time, so they go rummaging in the toy box. One finds the drum and the other finds the drumstick. Now they're stuck; unless one decides to be nice and let the other play, each will hold onto whatever they have and demand that the other give them the other piece, so neither gets to play.

Now imagine that you have not 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, because each is waiting for the other to release its mutex. This scenario is called *deadlock*, and it's 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, because the mutexes are serving different purposes, but other times it's not so simple, such as when the mutexes are each protecting a separate instance of the same class. Consider, for example, an operation that

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

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

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

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

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

    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::lock(lhs.m,rhs.m);                                    // 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
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

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

This ensures that the mutexes are correctly unlocked on function exit in the general case where the protected operation might throw an exception; it also allows for a simple return. Also, it's worth noting that locking either `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, this first lock is released automatically: `std::lock` provides all-or-nothing semantics with regard to locking the supplied mutexes.

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

3.2.5 Further guidelines for avoiding deadlock

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

Avoid nested locks

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

Avoid calling user-supplied code while holding a lock

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

Acquire locks in a fixed order

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

4 Synchronizing concurrent operations

This chapter covers

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

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

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

4.1 Waiting for an event or other condition

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

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

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

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

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

❶ Unlock the mutex

❷ Sleep for 100 ms

❸ Relock the mutex

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

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

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

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

4.1.1 Waiting for a condition with condition variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.2 Building a thread-safe queue with condition variables

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

Listing 4.2 `std::queue` interface

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

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

    void swap(queue& q);

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

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

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