Michael Whittaker

C++ Concurrency In Action

Thoughts and Notes

cppconcurrency github

Contents

Listings

Preface

This document contains the notes, musings, and thoughts generated during my reading of "C++ Concurrency in Action" by Anthony Williams. The notes were taken primarily to encourage a thorough reading of the book and to help me recall the most important tidbits from the book upon a rereading of my notes. I can imagine the notes may be helpful to more than just me, so I am making them publicly available. A concurrent programming novice, I cannot guarantee my notes are entirely correct, or even sensical at times. If you ever encounter a mistake, please contact me at mjw297@cornell.edu.

Along with the notes, I've thrown together some source code and other resources. Some of the code is taken directly from the text while some is original. All notes, code, and resources can be found at the cppconcurrency github.

Enjoy!

1 Hello, world of concurrency in C++!

For the first time since its original publication in 1998, the C++ acknowledges multithreaded applications. The C++11 standard offers components in the library to support portable, platform-agnostic multithreaded code with guaranteed behaviour. In this section, we'll learn what exactly multithreaded and concurrent programs are.

1.1 What is concurrency?

At the most basic level, concurrency is about two independent actions happening at the same time. For example, you can watch football while I go swimming.

1.1.1 Concurrency in computer systems

In computer systems, *concurrency* means a single system performing multiple independent activities in parallel, rather than sequentially.

Historically, most computers have only one processor, but give the illusion of concurrency by *task switching*: executing one application for a bit, then executing another for a bit, and so on. Some computers have multiple processors or multiple cores in a single processor. This provides *hardware concurrency*.

When operating in a single core system, the system has to perform a *context switch* when switching between tasks. This involves saving state, loading state, modifying memory, etc. These context switches take time. They also may not occur in a system that offers hardware concurrency.

All library features mentioned in this book will work on systems with or without hardware concurrency.

1.1.2 Approaches to concurrency

Imagine organizing a set of workers in an office. You could place each worker in her own office. This has the overhead of office space and communication costs, but each worker has her own set of resources. Imagine instead that all workers share the same office. You avoid the overheads of multiple offices but introduce the hassle of sharing resources. The former scenario represents multiple processes, and the latter represents multiple threads within the same process. You can implement any combination of the two.

Concurrency with multiple processes There are advantages and disadvantages to running multiple, single-threaded processes.

The disadvantages:

- Communication between processes can be complicated or slow.
- There is overhead launching and managing separate processes.

- Some languages (e.g. C++) don't offer many interprocess communication libraries.

The advantages:

- There is added security between processes. It becomes easier to write *safe* concurrent code.
- Multiple processes can be run on multiple computers connected on a network.

Concurrency with multiple threads Each thread in a multi-threaded application acts as a light-weight process. All threads share the same address space. Again, there are advantages and disadvantages.

The disadvantages:

- The coder must guarantee a consistent view of data between threads.

The advantages:

- Less overhead.
- Better library support.

This book will discuss only concurrency via multi-threading.

1.2 Why use concurrency?

The two main, almost only, reasons to use concurrency are separation of concerns and performance.

1.2.1 Using concurrency for separation of concerns

Separation of concerns and modularity is always a good idea when writing software. Threads allow you to modularize independent code that needs to run in parallel. For example, imagine a DVD player application. One thread needs to read, decode, and process the DVD data. Another thread needs to respond to user input. Separating each task to its own thread increases modularity. Note that here, the number of threads is independent of the number of processors.

1.2.2 Using concurrency for performance

Computers are getting increasingly faster with the introduction of more and more processors or cores. It is software's responsibility to take advantage of the resources. There are two ways to take advantage of threads.

Divide a single task into parallel parts Dividing a task is called *task parallelism*. Dividing an algorithm to operate on certain parts of data is called *data parallelism*. Algorithms that are readily susceptible to parallelism are called *embarrassingly parallel*, *naturally parallel*, or *conveniently parallel*.

Solve the same task multiple times Perform the same operation on multiple chunks of data. For example, read multiple files at the same time. It still takes the same amount of time to process one chunk of data, but we can now process multiple chunks all at once.

1.2.3 When not to use concurrency

- Concurrent code can be difficult to understand. This makes it hard to read and maintain and can also introduce bugs. Make sure the benefits of concurrency outweigh these costs.
- There is overhead to launch a thread. The OS has to allocate stack, adjust the scheduler, etc. Make sure the thread's execution is long enough to warrant its spawning.
- Launching too many threads could slow down a system. Some computers have a finite amount of memory to distribute, and each thread needs its own stack. Many threads also increases the amount of context switching required.

1.3 Concurrency and multithreading in C++

Only recently has C++ supported standard multithreading. Understanding the history of threading will help motivate some of the new standard's design decisions.

1.3.1 History of multithreading in C++

The 1998 C++ standard does not acknowledge multiple threads or a memory model. Thus, you need compiler-specific extensions to write multithreaded code. There are some extensions, like POSIX C and the Microsoft Windows API, that have allowed the construction of multithreaded C++ programs.

Not content with the purely imperative multithreading support of the C API, people have also turned to object-oriented C++ libraries such as Boost. Most libraries take advantage of RAII to ensure mutexes are unlocked when the relevant scope is exited.

Though there has been lots of support, there still existed the need for a standard for portability and performance.

1.3.2 Concurrency support in the new standard

C++ supports many concurrency facilities and borrowed a lot from Boost. Other C++11 additions also support the addition of standard concurrency. The result is improved semantics and performance.

1.3.3 Efficiency in the C++ Thread Library

One concern of developers is that of efficiency. It's important to know the implementation costs associated with the abstracted C++ libraries that can be circumvented using the lower-level facilities. Such cost is an *abstraction penalty*. The new C++ standard is fast and often prevents bugs. Even if profiling indicates that C++ standard facilities are the bottleneck, it still may be a result of poorly designed code.

1.3.4 Platform-specific facilities

If a developer wants to take advantage of platform-specific utility, the types in the C++ Thread Library offer a `native_handle()` member function that grants access to such facilities.

1.4 Getting Started

And now for some code.

1.4.1 Hello, Concurrent World

Listing 1: A simple Hello Concurrent World program

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

void hello() {
    std::cout << "Hello Concurrent World!" << std::endl;
}

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

. Some items to notice in Listing 1

- The `<thread>` library declares thread management utilities. Other data management utilities are elsewhere.
- Every thread object needs an *initial function*. For the initial thread, this is `main()`. For user spawned threads, the function must be specified in the `std::thread` constructor. Here, it is `hello()`.
- We call `join` to wait for the thread to finish.

2 Managing threads

In this chapter, we'll discuss the basics of launching a thread, waiting for it to finish, or running it in the background. We'll also look at passing additional parameters to a thread, transferring thread ownership, and identifying a good number of threads to use.

2.1 Basic thread management

Every C++ has at least one thread. The one started by the C++ runtime invokes `main()`. When we launch threads, we have to provide an entry point, similar to `main()`. When the entry point terminates, the thread exits.

2.1.1 Launching a thread

To launch a thread, pass a callable object to the `std::thread` constructor.

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

This callable can be a function, class, lambda, whatever! Don't forget to include the `<thread>` header.

After spawning a thread, you must decide to wait for it to finish (by joining) or let it run on its own (by detaching it). If you don't decide before the `std::thread` object is destroyed, then `std::terminate()` is called within the thread. Joining or detaching the thread must happen even in the event of exceptions.

Functor An example thread spawn using a functor is given in Listing 2. Note that the functor is **copied** into the storage of the newly created thread and invoked from there. In order to avoid the most vexing parse, you can initialize in a variety of ways shown in Listing 2. Also note that the output of the execution of Listing 2 is non-deterministic due to the nature of threads and messy because we do not lock standard output.

Listing 2: Spawning a thread using a functor

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

struct background_task {
    int x_;
    background_task(int x): x_(x) {}
    void operator()() const {
        std::cout << "Hello from background_task " << x_ << std::endl;
    }
};

int main() {
    background_task b1(1);
    std::thread t1(b1);                     // explicit initialization
```

```cpp
    std::thread t2((background_task(2))); // extra parentheses
    std::thread t3{background_task(3)};   // uniform initialization

    t1.join();
    t2.join();
    t3.join();
}
```

Lambda We can also spawn threads using lambdas as shown in Listing 3.

Listing 3: Spawning a thread using a lambda

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

int main() {
    std::thread t([]() -> void {std::cout << "hi" << std::endl;});
    t.join();
}
```

Dangling references If you decide to let your thread run, you must ensure that it does not reference any invalid data. This occurs when the thread has a pointer or reference to automatically allocated memory. See Listing 4 for an example.

Listing 4: Dangling references and threads

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

struct fun {
    int& x;
    fun(int x_): x(x_) {}
    void operator()() const {
        for (int i = 0; i < 1000000; ++i) {
            ++x;
            --x;
        }
    }
};

int main() {
    int local_state = 0;
    std::thread t{fun(local_state)};
    t.detach();
}
```

2.1.2 Waiting for a thread to complete

If you need to wait for a thread to complete, call `join()` on the associated `std::thread` object. In Listing 4, changing `t.detach()` to `t.join()` will remove the problem of a dangling reference. Listing 4 is a rather contrived example. Spawning a thread and immediately joining it doesn't do much. In less contrived code, the spawning thread would do other work or spawn multiple working threads.

`join()` also cleans up storage associated with a thread, so the `std::thread` object is no longer associated with its now finished thread. In fact, it's not associated with any thread. You can only call `join()` once on a thread. `joinable()` will return false, as shown in Listing 5.

Listing 5: You can only join a thread once

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

int main() {
    std::thread t([]() -> void {});
    t.join();
    std::cout << "t is joinable? " << t.joinable() << std::endl;
}
```

2.1.3 Waiting in exceptional circumstances

If you want to let a thread run in the background, you can usually call `detach()` immediately after spawning the thread. However, if you want to wait for a thread to finish, you must call `join()` in all circumstances, exceptional or otherwise. You could use `try-catch` blocks, as shown in Listing 6. See Listing 4 for the definition of `struct fun`.

Listing 6: Try-catch for exceptional joins

```cpp
struct fun;

void f() {
    int local_state = 0;
    std::thread t{fun(local_state)};
    try {
        // stuff
    }
    catch (/*something*/) {
        t.join();
        throw;
    }
    t.join();
}
```

Using `try-catch` blocks is verbose and bug-prone. A better means of guaranteeing a thread joins is to use RAII, as shown in Listing 7, Listing 8, and Listing 9.

Listing 7: thread guard header file

```cpp
#ifndef THREAD_GUARD_H
#define THREAD_GUARD_H

```

```
#include <thread>

namespace conc {

/*********************************************************************//**
 * \brief This class uses RAII to join a thread.
 *
 * A thread_guard instance is instantiated with an <std::thread>.
 * When the instance goes out of scope, the destructor will join the thread if
 * it is joinable. Otherwise, it does nothing.
 *
 * RAII is a powerful way to guarantee a thread is joined even in the face of
 * exceptional circumstances. Other means of joining despite exceptions, like
 * try-catch blocks, are more verbose and more prone to bugs.
 ************************************************************************/
class thread_guard {
public:
    /*****************************************************************//**
     * Constructs a thread_guard instance from <t_>.
     *
     * <t_> will be joined when this thread_guard instance goes out of scope,
     * if possible.
     *
     * \param t_ A thread to be joined by the destructor.
     * \see thread_guard::~thread_guard()
     ********************************************************************/
    explicit thread_guard(std::thread& t_);

    /*****************************************************************//**
     * Joins the thread associated with this thread_guard instance.
     *
     * If <t.joinable()> is true, <t> is joined. Otherwise, nothing happens.
     ********************************************************************/
    ~thread_guard();

    /*****************************************************************//**
     * Deleted copy constructor.
     ********************************************************************/
    thread_guard(const thread_guard&) = delete;

    /*****************************************************************//**
     * Deleted copy assignment operator.
     ********************************************************************/
    thread_guard& operator=(const thread_guard&) = delete;
private:
    /*****************************************************************//**
     * The thread managed by this instance of thread_guard. The thread will be
     * joined by the destructor if possible.
     ********************************************************************/
    std::thread& t;
};

} // namespace conc

```

```
#endif // THREAD_GUARD_H
```

Listing 8: thread guard cpp file

```
#include "thread_guard.h"
#include <thread>

namespace conc {

thread_guard::thread_guard(std::thread& t_): t(t_) {}

thread_guard::~thread_guard() {
    if (t.joinable()) {
        t.join();
    }
}

} // namespace conc
```

Listing 9: Joining using thread-guard

```
#include "thread_guard.h"
#include <iostream>
#include <thread>

using namespace conc;

struct fun {
    int& x;
    fun(int x_): x(x_) {}
    void operator()() const {
        for (int i = 0; i < 1000000; ++i) {
            ++x;
            --x;
        }
    }
};

int main() {
    int local_state = 0;
    std::thread t{fun(local_state)};
    thread_guard tg(t);
}
```

Notice in Listing 8 that we delete the default copy constructor and default copy assignment operator. Copying or assigning a `thread_guard` is dangerous because it could outlive the scope of the thread it is joining, thereby joining a non-joinable thread.

2.1.4 Running threads in the background

Calling `detach` on a thread will leave the thread to run in the background with no means of communicating with it. You can no longer create an `std::thread` object that owns the thread. You can no longer join the thread. The thread now belongs to the C++ runtime.

Detached threads are called *daemon threads* named after UNIX *daemon processes*.

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

Just like joining, you can only detach a thread object when it is associated with a thread. That is, when `joinable()` returns true.

One prime example of a daemon thread is a word processor that can edit multiple documents within the same application. An example of such a word processor is given in Listing 10.

Listing 10: Detaching a thread to handle other documents

```cpp
void edit_document(const std::string& filename) {
    open_document_and_display_gui(filename);
    while(!done_editing()) {
        user_command cmd = get_user_input();
        if (cmd.type == open_new_document) {
            const std::string new_name = get_filename_from_user();
            std::thread t(edit_document, new_name);
            t.detach();
        }
        else {
            process_user_input(cmd);
        }
    }
}
```

Also consider the code in Listing 11. If you detach a thread, but the main thread of execution terminates, then the detached thread will also be killed. Also note that you can detach an rvalue thread. This prevents you from inadvertently mucking with a thread object that has no associated thread anymore.

Listing 11: Detaching a thread until the main thread terminates

```cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <chrono>
#include <thread>

void cin_print() {
    std::string s;
    std::cin >> s;
    std::reverse(s.begin(), s.end());
    std::cout << "You typed " << s << std::endl;
}

int main() {
    std::thread(cin_print).detach();
    std::this_thread::sleep_for(std::chrono::seconds(5));
}
```

2.2 Passing arguments to a thread function

Passing arguments to the callable argument passed to a thread is as simple as passing additional arguments to the `std::thread` constructor. However, these additional arguments are **copied** into the internal storage of the thread!

In Listing 12, a pointer to a local variable, `buffer` is copied into the thread. If `oops` exits before `f` accesses `buffer`, we invoke undefined behaviour. We remedy the situation in Listing 13 by explicitly the `char *` to an `std::string` **before** passing the buffer to the constructor.

Listing 12: A multiple argument gotcha

```
#include <iostream>
#include <thread>
#include <cstdio>
#include <chrono>

void f(int i, const std::string& s) {
    std:: cout << i << s << std::endl;
}

void oops() {
    char buffer[1024];
    sprintf(buffer, "%i", 42);
    std::thread(f, 42, buffer).detach();
}

int main() {
    oops();
}
```

Listing 13: A multiple argument gotcha resolved

```
#include <iostream>
#include <string>
#include <thread>
#include <cstdio>

void f(int i, const std::string& s) {
    std:: cout << i << s << std::endl;
}

void not_oops() {
    char buffer[1024];
    sprintf(buffer, "%i", 42);
    std::thread(f, 42, std::string(buffer)).detach();
}

int main() {
    not_oops();
}
```

It is also possible to encounter the reverse scenario, when we want to pass a reference, but end up passing a copy. Listing 14 shows one such example. The local int, `i` is copied first into the local storage of the thread and then referenced. The copy of `i` would never change. In fact, on my machine, this code does not even compile. Instead, we have to make a ref using `std::ref` as shown in Listing 15.

Listing 14: Failing to pass a reference to a thread

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

void f(int& i) {
    ++i;
}

int main() {
    int i = 0;
    std::thread t(f, i);
    thread_guard(t);
}
```

Listing 15: Succeeding in passing a reference to a thread

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

void f(int& i) {
    ++i;
}

int main() {
    int i = 0;
    std::thread t(f, std::ref(i));
    t.join();
    std::cout << i << std::endl;
}
```

You can also pass a member function to a thread as long as the next argument is a pointer to the appropriate object, as shown in Listing 16.

Listing 16: Initializing thread with method

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

struct Foo {
    void bar(int i) {
        std::cout << "bar " << i << std::endl;
    }
};
```

```

int main() {
    Foo foo;
    std::thread t(&Foo::bar, &foo, 43);
    t.join();
}
```

Another interesting argument passing involves movable datatypes, such as in Listing 17.

Listing 17: Moving data into a thread.

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

void foo(std::unique_ptr<int> ip) {
    (*ip)++;
    std::cout << *ip << std::endl;
}

int main() {
    std::thread t1(foo, std::unique_ptr<int>(new int(42)));

    std::unique_ptr<int> ip(new int(0));
    std::thread t2(foo, std::move(ip));

    t1.join();
    t2.join();
}
```

2.3 Transferring ownership of a thread

Threads are *movable* but not *copyable*. In Listing 18, we see threads being moved around. In the last line, we try to assign a thread to `t1`, but `t1` is already associated with a thread. `std::terminate` is called.

Listing 18: Moving Threads

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

Threads can also be returned from functions as shown in Listing 19.

Listing 19: Returning a thread from a function

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

```cpp

void f() {
    std::cout << "f" << std::endl;
}

void g(int i) {
    std::cout << "g" << i << std::endl;
}

std::thread getf() {
    return std::thread(f);
}

std::thread getg() {
    std::thread t(g, 42);
    return t;
}

int main() {
    std::thread ft(getf());
    std::thread gt(getg());
    ft.join();
    gt.join();
}
```

Threads can also be passed into functions as shown in Listing 20.

Listing 20: Passing a thread into a function

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

void join(std::thread t) {
    t.join();
}

int main() {
    join(std::thread([]() -> void {std::cout << "hi" << std::endl;}));
    std::thread t([]() -> void {std::cout << "bye" << std::endl;});
    join(std::move(t));
}
```

One benefit of the movability of `std::thread`s is a modification of our previously written `thread_guard`. We can now write a `scoped_thread`, as shown in Listing 21, Listing 22, and Listing 23. Now, we no longer have to instantiate an lvalue thread. We can directly construct the thread within the constructor of the `scoped_thread`.

Listing 21: Scoped thread header

```cpp
#ifndef SCOPED_THREAD_H
#define SCOPED_THREAD_H

#include <thread>

```

```
namespace conc {

/*************************************************************************//**
 * \brief This class takes ownership of a thread and joins it using RAII.
 *
 * An instance of a scoped_thread takes ownership of a thread by moving it into
 * the instance. It verifies that the thread is joinable in the contructor and
 * throws and exception if it is not. Then, it joins the thread in the
 * destructor. Copy and assignment operators have been deleted to avoid
 * erroneous copies.
 *
 * Class Invariants:
 * - t is joinable.
 * - Upon destruction of a scoped_thread object, the thread is owns will be
 *   joined.
 *****************************************************************************/
class scoped_thread {
public:
    /*********************************************************************//**
     * Constructs a scoped_thread instance with <t_>.
     *
     * If <t_> is not joinable, an std::logic_error is thrown. Otherwise, this
     * instance succesfully moves t_ into a local copy. NOTE that the thread
     * <t_> is MOVED. If you isntantiate a scoped_thread instance with an
     * lvalue, do not attempt to touch that variable again!
     *
     * <t_> will be joined by the destructor.
     *
     * \param t_ A thread to be moved into this instance.
     * \see scoped_thread::~scoped_thread()
     *************************************************************************/
    explicit scoped_thread(std::thread t_);

    /*********************************************************************//**
     * Joins the thread associated with this instance.
     *************************************************************************/
    ~scoped_thread();

    /*********************************************************************//**
     * Deleted copy constructor.
     *************************************************************************/
    scoped_thread(const scoped_thread&) = delete;

    /*********************************************************************//**
     * Deleted copy assignment operator.
     *************************************************************************/
    scoped_thread& operator=(const scoped_thread&) = delete;

private:
    /*********************************************************************//**
     * The thread managed by this instance of scoped_thread.
     *
     * This thread is guaranteed to be joinable and will be joined in the
     * destructor of this instance.
     *************************************************************************/
    std::thread t;
};

} // namespace conc

#endif // SCOPED_THREAD_H
```

Listing 22: Scoped thread cpp

```
#include "scoped_thread.h"
#include <thread>

namespace conc {

scoped_thread::scoped_thread(std::thread t_): t(std::move(t_)) {
    if (!t.joinable()) {
        throw std::logic_error("scoped_thread constructed from non-joinable");
    }
}
scoped_thread::~scoped_thread() {
    t.join();
}

} // namespace conc
```

Listing 23: Scoped thread example

```
#include "scoped_thread.h"
#include <iostream>
#include <thread>
#include <chrono>

using namespace conc;

void f() {
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "bye!" << std::endl;
}

int main() {
    scoped_thread st{std::thread(f)};
    std::cout << "hi" << std::endl;
}
```

Threads can also be moved nicely into STL containers, as shown in Listing 24. Using the `scoped_thread` here doesn't work because a `scoped_thread` isn't movable. A better version of `scoped_thread` would fix this issue.

Listing 24: Making a vector of threads.

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

```
#include <thread>
#include <chrono>
#include <algorithm>

int main() {
    auto f = [](uint i) -> void {
        std::this_thread::sleep_for(std::chrono::seconds(2));
        std::cout << i << std::endl;
    };

    std::vector<std::thread> threads;
    for (uint i = 0; i <= 20; ++i) {
        threads.push_back(std::thread(f,i));
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

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

    std::cout << "bye!" << std::endl;
}
```

2.4 Choosing the number of threads at runtime

The function `std::thread::hardware_concurrency()` returns the number of actual hardware cores available, or 0 if the information is not accessible. Listing 25 is a simple example.

Listing 25: Displaying the number of hardware cores

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

int main() {
    std::cout << std::thread::hardware_concurrency() << std::endl;
}
```

We can use this information to implement a parallel version of `std::accumulate`, as shown in Listing 26. A slower, non-concurrent version is shown in Listing 27. Here, we don't handle exceptions. That material is saved for later.

Listing 26: Fast accumulation

```
#include "conc_numeric.h"
#include <iostream>
#include <vector>
#include <chrono>

int main() {
    std::vector<int> v(100000000, 1);

    using std::chrono::duration_cast;
    using std::chrono::nanoseconds;
    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    std::cout << conc::accumulate(v.begin(), v.end(), 0) << std::endl;
    auto end = clock::now();
    std::cout << duration_cast<nanoseconds>(end-start).count()/1000000 << "ns\n";
}
```

Listing 27: Slow accumulation

```
#include "conc_numeric.h"
#include <iostream>
#include <vector>
#include <chrono>

int main() {
    std::vector<int> v(100000000, 1);

    using std::chrono::duration_cast;
    using std::chrono::nanoseconds;
    typedef std::chrono::high_resolution_clock clock;

    auto start = clock::now();
    std::cout << std::accumulate(v.begin(), v.end(), 0) << std::endl;
    auto end = clock::now();
    std::cout << duration_cast<nanoseconds>(end-start).count()/1000000 << "ns\n";
}
```

The source code for the accumulation is in Listing 28 and Listing 29.

Listing 28: Fast accumulation header

```
#ifndef CONC_NUMERIC_H
#define CONC_NUMERIC_H

#include <thread>
#include <algorithm>
#include <numeric>

namespace conc {

/*****************************************************************************//**
 * Concurrent implementation of std::accumulate.
 *
 * Computes the sum of the given value init and the elements in the range
 * [first, last). This version uses operator+ to sum up the elements.
 * Everything is done concurrently.
 *******************************************************************************/
template <typename Iterator, typename T>
T accumulate(Iterator first, Iterator last, T init);

namespace details {

/*****************************************************************************//**
```

```
 * Helper functor for accumulate.
 *
 * Invokes std::acumulate on first, last, result and stores the result to
 * result.
 *
 * \see std::accumulate()
 * \see conc::accumulate()
 ******************************************************************************/
template <typename Iterator, typename T>
struct accumulate_block {
    void operator()(Iterator first, Iterator last, T& result);
};

} // namespace details
} // namespace conc

#include "conc_numeric.hpp"

#endif // CONC_NUMERIC_H
```

Listing 29: Fast accumulation “source”

```
namespace conc {

template <typename Iterator, typename T>
T accumulate(Iterator first, Iterator last, T init) {
    const ulong length = std::distance(first, last);

    if (length == 0) {
        return init;
    }

    const ulong min_per_t = 25;
    const ulong max_ts = (length + min_per_t - 1) / min_per_t;
    const ulong hw_ts = std::thread::hardware_concurrency();
    const ulong num_ts = std::min(hw_ts != 0 ? hw_ts : 2, max_ts);
    const ulong block_size = length / num_ts;
    std::vector<T> results(num_ts);
    std::vector<std::thread> threads(num_ts - 1);

    Iterator block_start = first;
    for (ulong i = 0; i < (num_ts - 1); ++i) {
        Iterator block_end = block_start;
        std::advance(block_end, block_size);
        threads[i] = std::thread(
            details::accumulate_block<Iterator, T>(),
            block_start, block_end, std::ref(results[i])
        );
        block_start = block_end;
    }
    details::accumulate_block<Iterator, T>()
            (block_start, last, results[num_ts - 1]);

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

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


namespace details {

template <typename Iterator, typename T>
void accumulate_block<Iterator, T>::operator()(Iterator first, Iterator last,
        T& result) {
    result = std::accumulate(first, last, result);
}

} // namespace details
} // namespace conc
```

2.5 Identifying threads

Threads can be identified with `std::thread::get_id()` or `std::this_thread::get_id()` which both return an id of type `std::thread::id`. id's have a total ordering, can be hashed, and can be outputted. Some fun with id's is shown in Listing 30.

Listing 30: Fun with thread ids

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

int main() {
    auto print_id = []() -> void {
        std::cout << std::this_thread::get_id() << std::endl;
    };

    std::thread t(print_id);
    t.join();
    print_id();
}
```

3 Sharing data between threads

One benefit of concurrency is sharing data between threads, but this can also be inconvenient and bug-prone. This chapter covers how to safely share data between threads in C++.

3.1 Problems with sharing data between threads

Data sharing problems arise because of data modification. **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 one thread begins modifying data, then trouble arises.

Often, programs have *invariants*: guarantees about the state or nature of the program. Temporarily, these invariants can be broken. For example, linked list invariants can be broken when deleting a node; heap invariants can be broken when updating a heap. If another thread accesses shared data when an invariant is broken, bugs occur. These are called *race conditions*.

3.1.1 Race conditions

In concurrency, a *race condition* is anything where the outcome depends on the relative ordering of execution of operations on two or more threads. Connotatively, race conditions are meant to mean problematic race conditions, which typically occur when modifying data in multiple steps yielding temporarily broken invariants.

3.1.2 Avoiding problematic race conditions

There are a couple ways to avoid problematic race conditions.

- Wrap your data structure in a protection mechanism so that only one thread will see the temporarily broken invariants.
- Redesign your invariants so modifications are sequences of indivisible steps. This is known as *lock-free programming* and is discussed more later.
- Use *transactions*. Read memory locally, make modifications, then commit the modifications. This is still an area of research.

3.2 Protecting share data with mutexes

Mutexes allow you to make some code mutually exclusive. That is, only one thread will be able to execute the code at any given time.

3.2.1 Using mutexes in C++

You create a mutex by constructing an `std::mutex` instance. You can invoke `lock()` and `unlock()` methods. Typically you use an `std::lock_guard` to automatically lock and unlock the mutex. Some fun with mutexes is shown in Listing 31.

Listing 31: Protecting a list with a mutex

```
#include "scoped_thread.h"
#include <iostream>
#include <algorithm>
#include <list>
#include <thread>
#include <mutex>

std::list<int> l;
std::mutex l_mutex;

void add_to_list(int v) {
    std::lock_guard<std::mutex> guard(l_mutex);
    std::cout << "adding " << v << std::endl;
    l.push_back(v);
}

bool list_contains(int v) {
    std::lock_guard<std::mutex> guard(l_mutex);
    return std::find(l.begin(), l.end(), v) != l.end();
}

int main() {
    auto add = []() -> void {
        for (int i = 0; i < 100; ++i) {
            add_to_list(i);
        }
    };
    conc::scoped_thread st{std::thread(add)};

    for (int i = 90; i < 100; ++i) {
        std::cout << "l has " << i << "? " << list_contains(i) << std::endl;
    }
}
```

In Listing 31, we use a global list and associated global mutex. In practice, we would pair both in a class. The two functions would become methods. If we wrap all methods in mutexes, everything becomes safe. Well, not quite! If a method returns a pointer or reference, then we could still trigger undefined behaviour and run into some trouble. Concurrent data structures require careful design to ensure that this doesn't happen.

3.2.2 Structuring code for protecting share data

We don't want to return any references or pointers to internal data. Less obviously, we also don't want to pass any references or pointers to user supplied functions. This could lead to trouble, as shown in Listing 32.

Listing 32: Accidentally passing out a reference to protected data.

```cpp
#include <iostream>
#include <mutex>

class some_data {
public:
    void do_something() {
        std::cout << "do_something!" << std::endl;
    }
private:
    int i;
    std::string s;
};

class data_wrapper {
public:
    template<typename Function>
    void process_data(Function f) {
        std::cout << "locking mutex" << std::endl;
        std::lock_guard<std::mutex> lg(m);
        f(data);
        std::cout << "unlocking mutex" << std::endl;
    }
private:
    some_data data;
    std::mutex m;
};


int main() {
    some_data *unprotected;
    data_wrapper x;

    auto malicious = [&](some_data& protected_data) {
        unprotected = &protected_data;
    };

    x.process_data(malicious);
    unprotected->do_something();
}
```

By passing internal data to a user provided function, we provide a back door for bugs. In general, don't pass pointers and references to protected data outside the scope of the lock by

- returning them
- storing them in externally visible memory
- passing them as arguments to user-supplied functions

3.2.3 Spotting race conditions inherent in interfaces

Even if you wrap everything nicely with mutexes, there may still be race conditions. For example, lets consider the stack interface provided by the STL as shown in Listing 33.

Listing 33: The interface to the `std::stack` container adapter

```
template <typename T, typename Container = std::deque<T>>
class stack {
public:
    /* constructors */
    bool empty() const;
    size_t size() const;
    T& top();
    T const& top() const;
    void push(T const&);
    void push(T&&);
    void pop();
    void swap(stack&&);
};
```

If we switch `top` to not return a reference and we wrapped every method with a mutex, there would still be issues. The return of `size` and `empty` could not be trusted in a concurrent environment. For example, consider the simple code in Listing 34. A thread could call `top` on a stack that is empty. Or, two threads could perform `do_something` on the same element.

Listing 34: The problem with the standard stack interface

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

In order to avoid this conundrum, we want to combine the `pop()` and `top()` calls. However, this leads into other issues. Imagine a `stack<vector<int>>`. If `pop` removed and returned an element, the stack could be modified and then you could run out of memory trying to return the element. The stack interface solves this problem by separating `top` and `pop`, but we are trying to combine them in our multithreaded environment. We can solve this problem in a couple of ways.

Option 1: Pass in a reference The user could pass a reference to a variable which will be assigned the popped value.

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

This is impractical because the user is forced to construct an instance of the popping class. Also, the type must be assignable.

Option 2: Require a no-throw copy constructor or move constructor There only exists danger if returning an element by value throws an exception. We could limit our types to those that can guarantee a copy will not throw an exception. However, these no-throw copy constructible objects are far and few between.

Option 3: Return a pointer to the popped item We can return a pointer to the popped item instead of the item itself. However, we run into issues managing the memory. This is where `std::shared_ptr` becomes useful.

Option 4: Provide both option 1 and either option 2 or 3 If we choose either option 1 or 2, it is easy to provide option 1, so we might as well.

Example definition of a thread-safe stack An example definition of a thread-safe stack is given in Listing 35, Listing 36, and Listing 37. Note that in Listing 37, an exception may still be thrown because the stack may be emptied after `empty()` is checked and `pop()` is called.

Listing 35: Concurrent stack header

```
#ifndef CONC_STACK_H
#define CONC_STACK_H

#include <exception>
#include <mutex>
#include <stack>
#include <memory>

namespace conc {

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

/***************************************************************************//**
 *
 ******************************************************************************/
template <typename T>
class stack {
public:
    /***********************************************************************//**
     *
     **************************************************************************/
    stack();

    /***********************************************************************//**
     *
     **************************************************************************/
    stack(const stack& other);

    /******************************************************************//**
     *
     ********************************************************************/
    stack& operator=(const stack&) = delete;

    /******************************************************************//**
     *
     ********************************************************************/
    void push(T val);

    /******************************************************************//**
     *
     ********************************************************************/
    std::shared_ptr<T> pop();

    /******************************************************************//**
     *
     ********************************************************************/
    void pop(T& value);

    /******************************************************************//**
     *
     ********************************************************************/
    bool empty() const;

private:
    /******************************************************************//**
     *
     ********************************************************************/
    std::stack<T> data;

    /******************************************************************//**
     *
     ********************************************************************/
    mutable std::mutex m;
};

} // namespace conc

#include "conc_stack.hpp"

#endif // CONC_STACK_H
```

Listing 36: Concurrent stack "source"

```
namespace conc {

const char *empty_stack::what() const throw() {
    return "empty stack";
}

template <typename T> stack<T>::stack() {}

template <typename T>
stack<T>::stack(const stack& other) {
    std::lock_guard<std::mutex> lock(other.m);
    data = other.data;
}

template <typename T>
void stack<T>::push(T val) {
    std::lock_guard<std::mutex> lock(m);
    data.push(val);
}

template <typename T>
std::shared_ptr<T> stack<T>::pop() {
    std::lock_guard<std::mutex> lock(m);

    if (data.empty()) {
        throw empty_stack();
    }

    const std::shared_ptr<T> res(std::make_shared<T>(data.top()));
    data.pop();
    return res;
}

template <typename T>
void stack<T>::pop(T& val) {
    std::lock_guard<std::mutex> lock(m);

    if (data.empty()) {
        throw empty_stack();
    }

    val = data.top();
    data.pop();
}

template <typename T>
bool stack<T>::empty() const {
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
}

} // namespace conc
```

Listing 37: Example with concurrent stack

```
#include "conc_stack.h"
#include "scoped_thread.h"
#include <iostream>
#include <thread>

int main() {
    conc::stack<int> s;
    for (int i = 0; i <= 1000000; ++i) {
        s.push(i);
    }

    auto f = [](conc::stack<int>& s) -> void {
        while (!s.empty()) {
            s.pop(); // an exception can still be thrown
        }
    };

    conc::scoped_thread st1{std::thread(f, std::ref(s))};
    conc::scoped_thread st2{std::thread(f, std::ref(s))};
}
```

Our experience with the thread-safe stack has shown that race conditions can arise from locking at too small a granularity. Problems can also arise at too large a granularity. Early implementations of the Linux kernel, for example, had a single global kernel lock. This slowed things tremendously.

Thus, there is a decision to make in the granularity of locking. If you choose a finer grained locking scheme, sometimes you need to hold multiple locks at once. This can lead to an issue called *deadlock*.

3.2.4 Deadlock: the problem and a solution

Imagine a toy that comes in two parts: a drum and a drumstick. Imagine you have two children. If one child gets both parts, he can play merrily. If each child gets one part, then they wait for the other part forever. Replace the children with threads and the toy with mutexes and you've got yourself *deadlock*.

Typically to avoid deadlock, you lock items in a certain prescribed order. However, this isn't always straight forward. If we have a function that takes to instances of some class and attempts to swap members of the two classes, there is no order we can follow. Luckily, the standard provides us with `std::lock`, as shown in Listing 38. It offers all-or-nothing semantics when locking mutexes.

Listing 38: Using `std::lock` and `std::lock_guard` in a swap operation

```
#include <iostream>
#include <ostream>
#include <mutex>

class X {
public:
    X(const int& i) : detail(i) {}

    friend void swap(X& lhs, X& rhs) {
        if (&lhs == &rhs) {
            return;
        }

        std::lock(lhs.m, rhs.m);
        std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
        std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
        std::swap(lhs.detail, rhs.detail);
    }

    friend std::ostream& operator<<(std::ostream& stream, const X& x) {
        stream << x.detail;
        return stream;
    }

private:
    int detail;
    std::mutex m;
};

int main() {
    int i = 0;
    int j = 42;
    X a(i);
    X b(j);

    std::cout << a << " " << b << std::endl;
    swap(a, b);
    std::cout << a << " " << b << std::endl;
}
```

3.2.5 Further guidelines for avoiding deadlock

Deadlock can occur even without locks. For example, if two threads join on one another, there will be deadlock. Generally, deadlock can occur whenever a thread is waiting for another thread and that thread could be waiting for the other. The guidelines for avoiding deadlock boil down to one rule: **don't wait for another thread if there's a chance it's waiting for you**.

Avoid nested locks Don't acquire a lock if you already have one. This will eliminate lock based deadlock (but not all deadlock). If you really need to lock multiple mutexes, use `std::lock()`.

Avoid calling user-supplied code while holding a lock If you acquire a lock and then call a user supplied function, it may acquire a lock, thus violating the first rule. Sometimes, this is unavoidable.

Acquire locks in a fixed order If you must acquire multiple threads and you can't call `std::lock()`, then acquire the locks in the same order in every thread. For example, consider a linked list where every node has a mutex. You want to lock mutexes as you traverse a list so that multiple threads can access the list all at once. If you lock hand over hand, there exists the possibility for deadlock.

Imagine two threads traversing the nodes in opposite directions. They could meet in the middle and create deadlock where one node locked A and is waiting for B while the other node locked B and is waiting for A.

By mandating a specific ordering on lock acquisition, you can avoid these potential deadlocks.

Use a lock hierarchy A particular case of defining a lock ordering, we can divide our program into layers and assign priorities to the mutexes at each level. A thread cannot lock a mutex if it holds a mutex at a lower level. An implementation of this hierarchical mutex is given in Listing 39 and Listing 40. A proper use of the mutex is given in Listing 41 and a hierarchy violating use is given in Listing 42.

Listing 39: Hierarchical mutex header

```
#ifndef HIERARCHICAL_MUTEX_H
#define HIERARCHICAL_MUTEX_H

#include <mutex>

namespace conc {

/***************************************************************************//**
 * A hierarchical mutex.
 *
 * Each mutex is assigned a priority. If a thread attempts to lock a
 * hierarchical mutex with a priority lower than the lowest priority mutex it
 * currently has locked, an std::logic_error is thrown. Otherwise, all lockable
 * operations are the same.
 ******************************************************************************/
class hierarchical_mutex {
public:
    /***********************************************************************//**
     * Constructs a hierarchical_mutex with the given priority.
     *
     * \param val Priority. Lower priorities mutex must be locked after higher
     * priority mutexes.
     **************************************************************************/
    explicit hierarchical_mutex(ulong val);

    /***********************************************************************//**
     * If the priority of this mutex is lower than the lowest priority mutex
     * owned by the thread, then lock occurs just as std::mutex::lock() occurs.
     * Otherwise, an std::logic_error is thrown.
     *
     * \see std::mutex::lock()
     **************************************************************************/
    void lock();

    /***********************************************************************//**
     * Same as std::mutex::unlock()
     *
     * \see std::mutex::unlock()
     ***********************************************************************/
    void unlock();

    /*********************************************************************//**
     * \see conc::hierarchical_mutex::lock()
     * \see std::mutex::unlock()
     ***********************************************************************/
    bool try_lock();

private:
    /*********************************************************************//**
     * The internal mutex managed by this instance.
     ***********************************************************************/
    std::mutex m;

    /*********************************************************************//**
     * The priority associated with this instance.
     ***********************************************************************/
    const ulong hier_val;

    /*********************************************************************//**
     * The lowest priority associated with any mutex locked at the time this
     * mutex was locked.
     ***********************************************************************/
    ulong prev_hier_val;

    /*********************************************************************//**
     * The lowest priority of a locked mutex in this thread.
     ***********************************************************************/
    static thread_local ulong this_thread_hier_val;

    /*********************************************************************//**
     * If the mutex being locked is a lower prioirty that the current thread’s
     * priority, an std::logic_error is thrown. Otherwise, nothing happens.
     ***********************************************************************/
    void check_hier_violation();

    /*********************************************************************//**
     * Updates prev_hier_val and this_thread_hier_val.
     ***********************************************************************/
    void update_hier_val();
};

} // namespace conc

#endif // HIERARCHICAL_MUTEX_H
```

Listing 40: Hierarchical mutex source

```
#include "hierarchical_mutex.h"
#include <climits>

namespace conc {
```

```cpp

thread_local ulong hierarchical_mutex::this_thread_hier_val(ULONG_MAX);

hierarchical_mutex::hierarchical_mutex(ulong val) :
        hier_val(val),
        prev_hier_val(0)
{}

void hierarchical_mutex::lock() {
    check_hier_violation();
    m.lock();
    update_hier_val();
}

void hierarchical_mutex::unlock() {
    this_thread_hier_val = prev_hier_val;
    m.unlock();
}

bool hierarchical_mutex::try_lock() {
    check_hier_violation();
    if (!m.try_lock()) {
        return false;
    }
    update_hier_val();
    return true;
}

void hierarchical_mutex::check_hier_violation() {
    if (this_thread_hier_val <= hier_val) {
        throw std::logic_error("mutex hierarchy violated");
    }
}

void hierarchical_mutex::update_hier_val() {
    prev_hier_val = this_thread_hier_val;
    this_thread_hier_val = hier_val;
}

} // namespace conc
```

Listing 41: Hierarchical mutex conforming example

```cpp
#include "hierarchical_mutex.h"
#include "scoped_thread.h"
#include <iostream>
#include <thread>
#include <mutex>

conc::hierarchical_mutex high_m(1000000);
conc::hierarchical_mutex medium_m(1000);
conc::hierarchical_mutex low_m(1);

void high_f();
void medium_f();
void low_f();

void high_f() {
    std::lock_guard<conc::hierarchical_mutex> lock(high_m);
    std::cout << "high" << std::endl;
    medium_f();
}

void medium_f() {
    std::lock_guard<conc::hierarchical_mutex> lock(medium_m);
    std::cout << "medium" << std::endl;
    low_f();
}

void low_f() {
    std::lock_guard<conc::hierarchical_mutex> lock(low_m);
    std::cout << "low" << std::endl;
}

int main() {
    conc::scoped_thread high_t{std::thread(high_f)};
}
```

Listing 42: Hierarchical mutex violating example

```cpp
#include "hierarchical_mutex.h"
#include "scoped_thread.h"
#include <iostream>
#include <thread>
#include <mutex>

conc::hierarchical_mutex high_m(2000000);
conc::hierarchical_mutex medium_m(2000);
conc::hierarchical_mutex low_m(2);
conc::hierarchical_mutex oops_m(1);

void high_f();
void medium_f();
void low_f();
void oops_f();

void high_f() {
    std::lock_guard<conc::hierarchical_mutex> lock(high_m);
    std::cout << "high" << std::endl;
    medium_f();
}

void medium_f() {
    std::lock_guard<conc::hierarchical_mutex> lock(medium_m);
    std::cout << "medium" << std::endl;
    low_f();
}

```

```cpp
void low_f() {
    std::lock_guard<conc::hierarchical_mutex> lock(low_m);
    std::cout << "low" << std::endl;
}

void oops_f() {
    std::lock_guard<conc::hierarchical_mutex> oops_lock(oops_m);
    high_f();
}

int main() {
    conc::scoped_thread oops_t{std::thread(oops_f)};
}
```

The downfall of the lock hierarchy is that a thread cannot own multiple mutexes of the same priority. For a hand-over-hand locking scheme, as with a linked list for example, this becomes impractical.

Extending these guidelines beyond locks The same precautions taken to avoid lock based deadlock can apply to other types of deadlock. For example, it is generally a bad idea to wait for a thread when owning a mutex. That thread may be waiting for your mutex.

3.2.6 Flexible locking with `std::unique_lock`

`std::unique_lock` is a more flexible version of `std::lock_guard`. It doesn't always own the mutex it is maintaining. As a result, it takes up slightly more space and can be slightly slower. But, it offers more flexibility.

You modify the behaviour of an `std::unique_lock` by passing in `std::defer_lock` or `std::adopt_lock`. Unique locks can also be unlocked manually. The swap operation from Listing 38 is rewritten using unique locks in Listing 43.

Listing 43: Using `std::lock()` and `std::unique_lock` in a swap operation

```cpp
#include <iostream>
#include <ostream>
#include <mutex>

class X {
public:
    X(const int& i) : detail(i) {}

    friend void swap(X& lhs, X& rhs) {
        if (&lhs == &rhs) {
            return;
        }

        std::unique_lock<std::mutex> lock_a(lhs.m, std::defer_lock);
        std::unique_lock<std::mutex> lock_b(rhs.m, std::defer_lock);
        std::cout << "lock_a owns lock? " << lock_a.owns_lock() << std::endl;
        std::cout << "lock_b owns lock? " << lock_b.owns_lock() << std::endl;

        std::lock(lock_a, lock_b);
        std::cout << "lock_a owns lock? " << lock_a.owns_lock() << std::endl;
        std::cout << "lock_b owns lock? " << lock_b.owns_lock() << std::endl;

        std::swap(lhs.detail, rhs.detail);
    }

    friend std::ostream& operator<<(std::ostream& stream, const X& x) {
        stream << x.detail;
        return stream;
    }

private:
    int detail;
    std::mutex m;
};

int main() {
    int i = 0;
    int j = 42;
    X a(i);
    X b(j);

    std::cout << a << " " << b << std::endl;
    swap(a, b);
    std::cout << a << " " << b << std::endl;
}
```

3.2.7 Transferring ownership between scopes

Because an `std::unique_lock` does not need to own the mutex it governs, it can be used to transfer the ownership of a lock. This can by done by moving, but not copying, the unique lock as shown in Listing 44.

Listing 44: Moving a unique lock

```
#include <iostream>
#include <mutex>

std::mutex m;

std::unique_lock<std::mutex> get_lock() {
    std::unique_lock<std::mutex> lock(m);
    std::cout << "got lock" << std::endl;
    return lock;
}

int main() {
    std::unique_lock<std::mutex> lock(get_lock());
    std::cout << "main" << std::endl;
}
```

3.2.8 Locking at an appropriate granularity

Lock granularity is a loosely defined term that describes the amount of data protected by a single lock. A fine-grained lock protects a small amount of data and a coarse-grained lock protects a large amount of data. It is important to choose an appropriate lock granularity. It is also important to not hold the lock when you don't need to; this can slow down all other threads waiting for the lock. You should especially avoid long operations when holding a lock, such as file I/O or obtaining another lock. An `std::unique_lock` is good for managing lock granularity, as shown in Listing 45.

Listing 45: Obtaining appropriate lock granularity using `std::unique_lock`

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

Listing 46 shows how to reduce the amount of time holding a lock by copying `int`. Note however, that this changes the semantics of the equality. The equality operator may return true even if the two instances were never equal at any given point in time.

Listing 46: Fine-grained locking on two int wrappers.

```
#include <iostream>
#include <mutex>

class Integer {
public:
    Integer(int data_) : data(data_) {}

    friend bool operator==(const Integer& lhs, const Integer& rhs) {
        return &lhs == &rhs || lhs.get_data() == rhs.get_data();
    }

private:
    int data;
    mutable std::mutex m;

    int get_data() const {
        std::lock_guard<std::mutex> lg(m);
        return data;
    }
};

int main() {
    Integer i1(1);
    Integer i2(2);
    Integer i3(1);

    std::cout << std::boolalpha << "i1 == i2? " << (i1 == i2) << std::endl;
    std::cout << std::boolalpha << "i1 == i1? " << (i1 == i1) << std::endl;
    std::cout << std::boolalpha << "i1 == i3? " << (i1 == i3) << std::endl;
}
```

3.3 Alternative facilities for protecting shared data

Although mutexes are the most general mechanism for protecting shared data, they are not the only mechanism. Some specific scenarios have more appropriate alternatives. For example, some data needs protection only during initialization. Using a mutex for this scenario can come with inefficiencies.

3.3.1 Protecting shared data during initialization

Some data is so expensive to construct – say loading a database or allocating a large block of memory – that we want to perform the construction only if necessary. That is, we want to *lazily* construct the object.

Single-threaded implementation Listing 47 shows a simple lazy construction typical of single threaded code.

Listing 47: Simple lazy construction

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>

std::shared_ptr<std::vector<int>> p;

void square() {
    if (!p) {
        p.reset(new std::vector<int>(10, 42));
    }

    std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
}

std::ostream& operator<<(std::ostream& os, std::vector<int> v) {
    os << "[";
    for (auto p = v.begin(); p != v.end(); ++p) {
        os << *p << (p == v.end() - 1 ? "" : ",");
    }
    os << "]";

    return os;
}

int main() {
    std::cout << "p is " << p << std::endl;
    square();
    std::cout << "p is " << p << std::endl;
    std::cout << "*p is " << *p << std::endl;
}
```

Naive multi-threaded implementation If the shared resource is inherently thread safe, then the only modification required to make the code in Listing 47 is to add a mutex to the initialization. However, this unnecessarily serializes the code as all threads must wait for the mutex to check if the shared resources has already been initialized. This is shown in Listing 48.

Listing 48: Lazy construction using a mutex that serializes code

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
#include <mutex>

std::shared_ptr<std::vector<int>> p;
std::mutex p_m;

void square() {
    std::unique_lock<std::mutex> lock(p_m);
    if (!p) {
        p.reset(new std::vector<int>(10, 42));
    }
    lock.unlock();

    std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
}

std::ostream& operator<<(std::ostream& os, std::vector<int> v) {
    os << "[";
    for (auto p = v.begin(); p != v.end(); ++p) {
        os << *p << (p == v.end() - 1 ? "" : ",");
    }
    os << "]";

    return os;
}

int main() {
    std::cout << "p is " << p << std::endl;
    square();
    std::cout << "p is " << p << std::endl;
    std::cout << "*p is " << *p << std::endl;
}
```

Undefined multi-threaded implementation The code in Listing 48 is so common, people have designed "better" ways to lazily construct. One such technique is the infamous *Double-Checked Locking* pattern shown in Listing 49. The pattern is infamous because it is incorrect. The unprotected pointer check and the pointer allocation are not synchronized. This can lead to undefined behaviour when operating on the pointer.

Listing 49: Double-checked locking undefined lazy construction

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
#include <thread>
#include <mutex>

std::shared_ptr<std::vector<int>> p;
std::mutex p_m;

void square() {
    if (!p) {
        std::unique_lock<std::mutex> lock(p_m);
        if (!p) {
            p.reset(new std::vector<int>(100, 42));
        }
    }

    std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
}

std::ostream& operator<<(std::ostream& os, std::vector<int> v) {
    os << "[";
    for (auto p = v.begin(); p != v.end(); ++p) {
        os << *p << (p == v.end() - 1 ? "" : ",");
    }
    os << "]";

    return os;
}

int main() {
    std::cout << "p is " << p << std::endl;

    std::thread t1(square);
    std::thread t2(square);
    t1.join();
    t2.join();

    std::cout << "p is " << p << std::endl;
    std::cout << "*p is " << *p << std::endl;
}
```

std implementation The C++ Standards Committee foresaw such scenarios and developed std::once_flag and std::call_once to handle it. The same implementation in Listing 47, Listing 48, and Listing 49 is rewritten using these features in Listing 50. The overhead of std::call_once typically less than the overhead of using a mutex explicitly.

Listing 50: Lazy construction using std::call_once and std::once_flag

```cpp
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
#include <thread>
#include <mutex>

std::shared_ptr<std::vector<int>> p;
std::once_flag p_flag;


void square() {
    std::call_once(p_flag, [](){p.reset(new std::vector<int>(100, 42));});
    std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
}

std::ostream& operator<<(std::ostream& os, std::vector<int> v) {
    os << "[";
    for (auto p = v.begin(); p != v.end(); ++p) {
        os << *p << (p == v.end() - 1 ? "" : ",");
    }
    os << "]";

    return os;
}

int main() {
    std::cout << "p is " << p << std::endl;

    std::thread t1(square);
    std::thread t2(square);
    t1.join();
    t2.join();

    std::cout << "p is " << p << std::endl;
    std::cout << "*p is " << *p << std::endl;
}
```

std::once_flag and std::call_once can also be used within a class, as shown in Listing 51.

Listing 51: Lazy construction using std::call_once and std::once_flag in a class

```cpp
#include "scoped_thread.h"
#include <iostream>
#include <vector>
#include <memory>
```

```
#include <algorithm>
#include <thread>
#include <mutex>

using namespace std;

class lunchline {
public:
    lunchline(): entre("salisbury steak") {}

    lunchline(const std::string& entre_): entre(entre_) {}

    std::string front() {
        std::call_once(flag, &lunchline::init, this);
        return menu.front();
    }

    std::string back() {
        std::call_once(flag, &lunchline::init, this);
        return menu.back();
    }

private:
    std::vector<std::string> menu;
    std::string entre;
    std::once_flag flag;

    void init() {
        menu = std::vector<std::string>(100, entre);
    }
};

int main() {
    lunchline ll;
    conc::scoped_thread t1{thread([&]() {cout << ll.front() << endl;})};
    conc::scoped_thread t2{thread([&]() {cout << ll.back() << endl;})};
}
```

static In C++11, static variable initialization is guaranteed to be thread safe.

3.3.2 Protecting rarely updated data structures

Some data structures are read often but rarely written to: a generalization of the lazy initialization presented in the previous section. For example, consider a DNS cache that resolved domain names to their corresponding IP addresses. Such a data structure is rarely updated, but often read.

We need to use a `boost:shared_mutex` and `boost::shared_lock`. Using `std::lock_guard <boost::shared_mutex>` and `std::unique_lock<boost::shared_mutex>` ensure exclusive access as usual. Using `boost::shared_lock<boost::shared_mutex>` allows for shared access; multiple threads can share the lock at once.

If any thread has shared access to the lock, a thread trying to get exclusive access will block until all sharing threads relinquish their lock. Likewise, if a thread has exclusive access, any thread trying to get exclusive or shared access will block until the thread relinquishes the lock.

An example dns cache using these boost mechanisms is given in Listing 52.

Listing 52: DNS cache using `boost::shared_mutex` and `boost::shared_lock`

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

class dns_entry {
public:
    dns_entry(int ip_=-1) : ip(ip_) {}

    friend bool operator==(const dns_entry& lhs, const dns_entry& rhs) {
        return lhs.ip == rhs.ip;
    }

private:
    int ip;
};

class dns_cache {
public:
    dns_entry find_entry(const std::string& domain) const {
        // multiple threads will share this lock
        boost::shared_lock<boost::shared_mutex> lk(entry_mutex);
        const auto it = entries.find(domain);
        return (it == entries.end()) ? dns_entry() : it->second;
    }

    void updated_or_add_entry(const std::string& domain, const dns_entry &dns) {
        // only one thread will own this lock
        std::lock_guard<boost::shared_mutex> lk(entry_mutex);
        entries[domain] = dns;
    }

private:
    std::map<std::string, dns_entry> entries;
    mutable boost::shared_mutex entry_mutex;
};


int main() {
    dns_cache cache;
}
```

3.3.3 Recursive locking

It is undefined behaviour for to lock an `std::mutex` that you already own. An `std::recursive_mutex` works exactly like an `std::mutex` except that it can be relocked by the same thread that already owns it. The mutex must be unlocked the same number of times it was locked. If you ever find yourself using a recursive mutex, however, you should rethink your design and make sure it is necessary.

4 Synchronizing concurrent operations

In the last chapter we looked at how to protect shared data. However, threads also often need to synchronize actions. For example, one thread may need to wait for another thread to complete a task before it executes some action. This section will discuss *condition variables* and *futures* to accomplish this synchronization.

4.1 Waiting for an event or other condition

Imagine you are travelling overnight on a train. You don't want to miss your stop. There are a couple ways to make sure you get off at your stop. The first is to stay awake all night. You won't miss your stop, but you'll be tired. The second is where you look up the estimated time of arrival and set an alarm for that time. This works too, but you may wake up too early or, if your alarm clock dies, too late. Third and ideally, you would simply go to sleep and have someone else wake you up when you get to your stop.

If a thread is waiting for another thread to complete a task, it has similar options.

Option 1: Staying up all night If a thread is waiting for another thread to complete a task, it can simply check a flag protected by a mutex. When the other thread completes its work, it sets the flag. This is akin to staying up all night talking to the driver of the train.

This strategy is a poor one because the waiting thread takes up precious processing time and, by grabbing the mutex protecting the flag, it can actually prevent the waited on thread from setting the flag.

Option 2: Sleeping a little The second alternative is to sleep a little bit in between flag checks. This, like setting an alarm, can lead to a thread waking up too early or too late. An example of short sleeps is shown in Listing 53.

Listing 53: Waiting for an event with short sleeps

```cpp
#include "scoped_thread.h"
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

bool flag;
std::mutex flag_mutex;

void wait() {
    std::unique_lock<std::mutex> lock(flag_mutex);

    while (!flag) {
        std::cout << "unlocking mutex" << std::endl;
        lock.unlock();

        std::cout << "sleeping" << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        std::cout << "locking mutex" << std::endl;
        lock.lock();
    }

    std::cout << "done!" << std::endl;
}

void set() {
    std::this_thread::sleep_for(std::chrono::seconds(5));
    std::lock_guard<std::mutex> lock(flag_mutex);
    flag = true;
}

int main() {
    conc::scoped_thread waiter{std::thread(wait)};
    conc::scoped_thread setter{std::thread(set)};
}
```

Option 3: Getting woken up The ideal option of waiting for an event is to be woken up by the thread you're waiting for. This facility is provided by the C++ Standard Library in the form of a *condition variable*.

4.1.1 Waiting for a condition with condition variables

The C++ Standard Library offers two forms of a condition variable: `std::condition_variable` and `std::contition_variable_any`, both of which are declared in the `<condition_variable>` header. The former, `std::condition_variable`, must be used with mutexes while the second, `std::condition_variable_any`, can be used with anything mutex-like.

Listing 54 shows a simple way for a thread to wait for a condition using condition variables. One thread pushes `int`s to a queue while the other thread waits for the queue to be non-empty, pops an element, and prints it.

The popping thread waits for a condition by calling `cv.wait(lk, /*callable predicate */)`. First, the callable predicate is invoked. If it returns true, the function returns. If if returns false, the lock is relinquished and the thread goes to sleep. The thread can wake up if another thread notifies the condition variable. If it wakes up without being signalled, which it can do, it is called a *spurious wake*. When the thread wakes up, it reacquires the lock and repeats the process. That is, if the callable predicate returns true, it returns otherwise it relinquishes the lock and goes back to sleep. As the callable predicate will be called an indeterminate number of times, it is generally good idea to avoid functions with side effects.

The pushing thread pushes an `int` to the queue every second. It then wakes up a thread waiting on the condition variable, if there is one.

Listing 54: Simple waiting using a condition variable

```
#include "scoped_thread.h"
```

```cpp
#include <iostream>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>

std::queue<int> data_queue;
std::mutex data_mutex;
std::condition_variable data_cv;

void push() {
    for (int i = 0; i <= 20; ++i) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock(data_mutex);
        data_queue.push(i);
        data_cv.notify_one();
    }
}

void pop() {
    bool done = false;

    while (!done) {
        std::unique_lock<std::mutex> lock(data_mutex);
        data_cv.wait(lock, []{return !data_queue.empty();});
        int i = data_queue.front();
        data_queue.pop();
        lock.unlock();

        std::cout << i << std::endl;
        if (i == 20) {
            done = true;
        }
    }
}

int main() {
    conc::scoped_thread pusher{std::thread(push)};
    conc::scoped_thread popper{std::thread(pop)};
}
```

4.1.2 Building a thread-safe queue with condition variables

If we are going to design a thread-safe queue, it is beneficial to look at `std::queue` for inspiration on the operations we wish to provide. The `std::queue` interface can be found at cppreference's queue documentation.

If we ignore the more sophisticated constructors, assignment operators, and swap operations, we are left with three main groups of operations: queue information (`size` and `empty`), queue query (`front()` and `backk()`), and queue modification (`push()`, `pop()`, and `emplace()`). Much as in Listing 35 and Listing 36, we must combine `front()` and `pop()`. Slightly different, we will make a `try_pop()` function and a `wait_pop()` function. The first will attempt to pop, but return if it fails. The second will wait until it can successfully pop from the queue.

A fully implemented concurrent queue with example usage is given in Listing 55, Listing 56, and Listing 57.

Listing 55: Concurrent queue header

```cpp
#ifndef CONC_QUEUE_H
#define CONC_QUEUE_H

#include <queue>
#include <mutex>
#include <condition_variable>
#include <memory>

namespace conc {

template <typename T>
class queue {
public:
    explicit queue();
    explicit queue(const queue& other);
    void push(T val);
    void wait_pop(T& val);
    std::shared_ptr<T> wait_pop();
    bool try_pop(T& val);
    std::shared_ptr<T> try_pop();
    bool empty() const;

private:
    std::queue<T> data;
    mutable std::mutex m;
    std::condition_variable not_empty_cv;
};

} // namespace conc

#include "conc_queue.hpp"

#endif // CONC_QUEUE_H
```

Listing 56: Concurrent queue “source”

```cpp
namespace conc {

template <typename T> queue<T>::queue() {}

template <typename T>
queue<T>::queue(const queue& other) {
    std::lock_guard<std::mutex> lock(other.m);
    data = other.data;
}
```

```

template <typename T>
void queue<T>::push(T val) {
    std::lock_guard<std::mutex> lock(m);
    data.push(val);
    not_empty_cv.notify_one();
}

template <typename T>
void queue<T>::wait_pop(T& val) {
    std::unique_lock<std::mutex> lock(m);
    not_empty_cv.wait(lock, [this]{return !data.empty();});
    val = data.front();
    data.pop();
}

template <typename T>
std::shared_ptr<T> queue<T>::wait_pop() {
    std::unique_lock<std::mutex> lock(m);
    not_empty_cv.wait(lock, [this]{return !data.empty();});
    std::shared_ptr<T> popped(std::make_shared<T>(data.front()));
    data.pop();
    return popped;
}

template <typename T>
bool queue<T>::try_pop(T& val) {
    std::lock_guard<std::mutex> lock(m);

    if (data.empty()) {
        return false;
    }

    val = data.front();
    data.pop();
    return true;
}

template <typename T>
std::shared_ptr<T> queue<T>::try_pop() {
    std::lock_guard<std::mutex> lock(m);

    if (data.empty()) {
        return nullptr;
    }

    std::shared_ptr<T> popped(std::make_shared<T>(data.front()));
    data.pop();
    return popped;
}

template <typename T>
bool queue<T>::empty() const {
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
}

} // namespace conc
```

Listing 57: Concurrent queue example that prints prime factors

```
#include "scoped_thread.h"
#include "conc_queue.h"
#include <iostream>
#include <string>
#include <memory>
#include <algorithm>
#include <boost/multiprecision/cpp_int.hpp>

using boost::multiprecision::cpp_int;

namespace {
    conc::queue<cpp_int> numbers;

    std::mutex cout_mutex;

    const int hw_conc = std::thread::hardware_concurrency();
    const int num_threads = (hw_conc == 0) ? 2 : hw_conc - 1;
    const int killnum = -1;
}

std::string get_factors(cpp_int n) {
    cpp_int factor = 2;
    std::string factors = "";

    if (n < 0) {
        n *= -1;
    }

    while (factor * factor <= n) {
        if (n % factor == 0) {
            factors += boost::lexical_cast<std::string>(factor);
            factors += " ";
            n /= factor;
        }
        else {
            ++factor;
        }
    }

    if (n > 1) {
        factors += boost::lexical_cast<std::string>(n);
    }

    return factors;
}

void push() {
    cpp_int i;
    while (std::cin >> i) {
        if (killnum == i) {
            for (int j = 0; j < num_threads; ++j) {
                numbers.push(killnum);
            }
            return;
        }
        numbers.push(i);
    }
}

void pop() {
    while (true) {
        cpp_int i = *(numbers.wait_pop());
        if (killnum == i) {
            return;
        }

        std::lock_guard<std::mutex> cout_lock(cout_mutex);
        std::cout << i << ": " << get_factors(i) << std::endl;
    }
}

int main() {
    conc::scoped_thread pusher{std::thread(push)};

    std::vector<std::thread> poppers;
    for (int i = 0; i < num_threads; ++i) {
        poppers.push_back(std::thread(pop));
    }
    std::for_each(poppers.begin(), poppers.end(), std::mem_fn(&std::thread::join));
}
```

4.2 Waiting for one-off events with futures

Often times, a thread needs to wait for a one-off event. The C++ Standard Library models a one-off event with a *future*. A thread can periodically check on the future or complete another task until it needs to wait for the future to complete.

The C++ Standard Library offers two kinds of features: *unique futures* (`std::future<>`) and *shared futures* (`std::shared_future<>`), modelled after `std::unique_ptr` and `std::shared_ptr`. An `std::future` instance is the only object associated with its event whereas multiple `std::shared_future` objects may refer to the same event. The futures themselves don't offer an synchronization; they still must be protected with something like a mutex.

4.2.1 Returning values from background tasks

Suppose you have a long-running calculation that you expect will eventually yield a useful result. Instead of spawning a new thread, we can use `std::async` which returns an `std::`

`future`. We call `get()` on the future when we need the result. A very simple usage of futures and asynchronous calculations is shown in Listing 58. Futures also play well with additional arguments and move semantics as shown in Listing 59.

Listing 58: Very simple future usage

```
#include <iostream>
#include <future>
#include <thread>
#include <chrono>

int get_five() {
    std::this_thread::sleep_for(std::chrono::seconds(5));
    return 5;
}

int main() {
    std::future<int> five = std::async(get_five);
    std::cout << "patience is a virtue" << std::endl;
    std::cout << five.get() << std::endl;
}
```

Listing 59: Multiple arguments and move semantics with futures

```
#include <iostream>
#include <string>
#include <future>

struct Cat {
    void meow(int times, std::string mew) {
        for (int i = 0; i < times; ++i) {
            std::cout << mew << std::endl;
        }
    }

    std::string catify(const std::string& phrase) {
        return "meow " + phrase;
    }

    friend std::ostream& operator<<(std::ostream& os, const Cat&) {
        os << "cat";
        return os;
    }
};

struct Dog {
    double operator() (double treats) {
        return treats * 2;
    }
};

struct mover {
    mover() {}
    mover(mover&&) {}
    mover(const mover&) = delete;
    mover& operator=(mover&& m) {return m;}
    mover& operator=(const mover&) = delete;

    void operator()(){}
};

Cat adopt(Cat&) {
    return Cat();
}

int main() {
    Cat c;
    auto f1 = std::async(&Cat::meow, &c, 2, "meow"); // (&c)->meow(2, "meow")
    auto f2 = std::async(&Cat::catify, c, "purrr");  // c_copy.catify("purrr")
    auto f3 = std::async(adopt, std::ref(c));        // adopt(c);

    Dog d;
    auto f4 = std::async(Dog(), 3.1415962); // moved_d(3.1...)
    auto f5 = std::async(std::ref(d), 1.0); // d(1.0)

    auto f6 = std::async(mover()); // moved_tmp()

    f1.get();
    std::cout << f2.get() << std::endl;
    std::cout << f3.get() << std::endl;
    std::cout << f4.get() << std::endl;
    std::cout << f5.get() << std::endl;
    f6.get();
}
```

By default, it is up to the implementation to decide whether or `std::async` starts a new thread or runs sequentially when the future value is requested or waited on. You can specify the behaviour of the future by passing a first argument to `std::async` of type `std::future`. There are three possibilities:

1. `std::launch::async` A new thread is launched.
2. `std::launch::deferred` The function call is deferred until `wait()` or `get()` is called.
3. `std::launch::async | std::launch::deferred` (default) The implementation is free to choose.

Note that if the function is deferred, it may never actually be invoked.

4.2.2 Associating a task with a future

`std::packaged_task<>` ties a future to a function or callable object. The template parameter of the `std::packaged_task<>` is a function signature, and the packaged task must be constructed with a callable that can accept the parameters and return a value convertible to the template's return value. Simple packaged task usage is shown in Listing **??**.

Listing 60: Simple packaged task usage

```cpp
#include <iostream>
#include <future>

int mult2(int i) {
    std::cout << "mult2(" << i << ")" << std::endl;
    return i * 2;
}

int main() {
    // task 1
    std::packaged_task<int(int)> task1(mult2);
    std::future<int> fut1 = task1.get_future();
    task1(3);
    std::cout << fut1.get() << std::endl;

    // task 2
    std::packaged_task<long(int)> task2(mult2);
    std::future<long> fut2 = task2.get_future();
    task2(4);
    std::cout << fut2.get() << std::endl;

    // task 3
    std::packaged_task<int(int)> task3(mult2);
    task3(5);
    std::future<int> fut3 = task3.get_future();
    std::cout << fut3.get() << std::endl;

    // task 4
    std::packaged_task<void()> task4([]{std::cout << "void" << std::endl;});
    task4();
    task4.get_future().get();
}
```

Packaged tasks can also be passed between threads. Listing 61 shows this in action. A printing thread reads and sorts lines of input from standard input. The main thread sends tasks for that thread to perform.

Listing 61: Passing packaged tasks between threads

```cpp
#include "scoped_thread.h"
#include <iostream>
#include <algorithm>
#include <queue>
#include <chrono>
#include <memory>
#include <thread>
#include <future>
#include <cwctype>

using std::cout;
using std::endl;

```

```
std::queue<std::packaged_task<void()>> tasks;
std::mutex tasks_mutex;
bool done;

void print() {
    while (!done) {
        std::string l;
        std::getline(std::cin, l);
        l.erase(remove_if(l.begin(), l.end(), std::iswspace), l.end());
        std::sort(l.begin(), l.end());
        std::cout << l << std::endl;

        std::packaged_task<void()> task;
        {
            std::lock_guard<std::mutex> tasks_lock(tasks_mutex);
            if (tasks.empty()) {
                continue;
            }
            task = std::move(tasks.front());
            tasks.pop();
        }
        task();
    }
}

int main() {
    conc::scoped_thread printer{std::thread(print)};

    tasks.push(std::packaged_task<void()>([]{cout << "task 1" << endl;}));
    tasks.push(std::packaged_task<void()>([]{cout << "hello!" << endl;}));
    tasks.push(std::packaged_task<void()>([]{done = true;}));
}
```

4.2.3 Making (std::)promises

`std::async` and `std::packaged_task<>` both allowed for the asynchronous return of a one-off value, in the form of an `std::future<>`. Both implicitly set some shared state that the `std::future<>` could access. Async objects were constructed with callables and set the shared state to the return of the callable. Packaged tasks did the same, but the return value was set when the packaged task was invoked using the function call operator. `std::promise`s allow you to explicitly set the value of an `std::future`. A simple usage of `std::promise<>` is given in Listing 62. A slightly more advanced usage is given in Listing 63.

Listing 62: Very simple promise usage

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

void promise_five(std::promise<int> p) {
    p.set_value(5);
}
```

```

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();
    std::thread t(promise_five, std::move(p));

    std::cout << f.get() << std::endl;

    t.join();
}
```

Listing 63: Simple promise usage

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

void promise_cin(std::promise<int> p) {
    int i;
    std::cin >> i;
    p.set_value(i);
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();
    std::thread t(promise_cin, std::move(p));

    std::cout << "waiting for the future!" << std::endl;
    std::cout << "future: " << f.get() << std::endl;

    t.join();
}
```

`std::promise`s are used to represent asynchronous tasks that cannot be expressed as a single function or ones whose result may come from multiple sources. For example, if a single thread is processing multiple incoming packets in a single function, it may process multiple promises. A simpler more contrived example is given in Listing 64.

Listing 64: More advanced promising

```
#include "scoped_thread.h"
#include <iostream>
#include <thread>
#include <future>

using conc::scoped_thread;

void promise_cin(std::promise<int> ip, std::promise<std::string> sp) {
    int i;
    std::string tmp;
    std::string s;

```

```cpp
    std::cout << "> [num] [string]" << std::endl << "> ";
    std::cin >> i >> tmp;
    for (int j = 0; j < i; ++j) {
        s += tmp;
    }

    ip.set_value(i);
    sp.set_value(s);
}

int main() {
    std::promise<int> ip;
    std::promise<std::string> sp;
    std::future<int> ifut = ip.get_future();
    std::future<std::string> sfut = sp.get_future();

    scoped_thread t{std::thread(promise_cin, std::move(ip), std::move(sp))};

    std::cout << "i: " << ifut.get() << std::endl;
    std::cout << "s: " << sfut.get() << std::endl;
}
```

4.2.4 Saving an exception for the future

Consider the code in Listing 65 in which an exception is thrown in a single-threaded piece of code. If we are expecting an asynchronous return value in the form of an `std::future`, it would be ideal if the future was aware of the exception much as in the single-threaded case. In fact, this is the case. If `std::async` or `std::packaged_task<>` thrown an exception, the exception is placed in the `std::future` and is thrown when `std::future::get()` is invoked. This is shown in Listing 66.

Listing 65: Throwing an exception in a single-threaded piece of code

```cpp
#include <iostream>
#include <stdexcept>
#include <cmath>

template <typename Integral>
double sqrt(Integral i) {
    if (i < 0) {
        throw std::out_of_range("i < 0");
    }
    return std::sqrt(i);
}

int main() {
    std::cout << "sqrt(4):  " << sqrt(4) << std::endl;
    std::cout << "sqrt(-1): " << sqrt(-1) << std::endl;
}
```

Listing 66: Catching exceptions thrown by `std::async` and `std::packaged_task<>`

```cpp
#include <iostream>
#include <future>

char throws_two() {
    throw 2;
}

int main() {
    std::future<char> f1 = std::async(throws_two);
    std::future<char> f2 = std::async(throws_two);
    std::packaged_task<char()> pt(throws_two);
    std::future<char> f3 = pt.get_future();
    pt();

    // catching async future
    try {
        f1.get();
    }
    catch (int e) {
        std::cerr << e << std::endl;
    }

    // waiting async future
    f2.wait();

    // catching packaged task future
    try {
        f3.get();
    }
    catch (int e) {
        std::cerr << e << std::endl;
    }
}
```

Naturally, `std::promise` provides the same functionality as `std::async` and `std::packaged_task<>` with an explicit function call, `std::promise::set_exception`. `std::promise::set_exception` can be invoked with `std::current::exception` or `std::make_exception_ptr(/*exception */)` as shown in Listing 67. When possible, using `std::make_exception_ptr()` is preferred over `std::current_exception()`.

Listing 67: Storing exceptions using promises

```cpp
#include <iostream>
#include <exception>
#include <stdexcept>
#include <future>

int main() {
    std::promise<char> p1;
    std::promise<char> p2;
    std::future<char>  f1 = p1.get_future();
    std::future<char>  f2 = p2.get_future();

    auto future_catch = [](std::future<char> f) {
        try {
            std::cout << f.get() << std::endl;
        }
        catch (int i) {
            std::cerr << i << std::endl;
        }
    };

    // current exception
    try {
        throw 2;
    }
    catch (...) {
        p1.set_exception(std::current_exception());
    }

    // make_exception_ptr
    p2.set_exception(std::make_exception_ptr(2));

    // catch exceptions
    future_catch(std::move(f1));
    future_catch(std::move(f2));
}
```

Also, if a future or packaged task is destroyed before a value is stored to the future, an `std::future_error` is stored in the future. This is shown in Listing 68.

Listing 68: Breaking promises

```
#include <iostream>
#include <future>

int main() {
    std::future<int> f1;
    std::future<int> f2;

    {
        std::promise<int> p;
        std::packaged_task<int()> t([]{return 1;});
        f1 = p.get_future();
        f2 = t.get_future();
    }

    try {
        std::cout << f1.get() << std::endl;
    }
    catch (const std::future_error&) {
        std::cerr << "broken promise :(" << std::endl;
    }

    try {
        std::cout << f2.get() << std::endl;
    }
    catch (const std::future_error&) {
        std::cerr << "broken promise :(" << std::endl;
    }
}
```

4.2.5 Waiting from multiple threads

One pitfall of the `std::future<>` is that only one thread can wait for the result. If you require that multiple threads wait for a single result, you'll need an `std::shared_future<>`. `std::shared_future<>` instances, unlike `std::future` instances, are copyable.

Individual access to future methods are not inherently synchronized. You can protect the shared data using a mutex to create mutual exclusion to the shared future. Preferably, you create a local copy of the shared future for each thread and each thread only accesses the shared data through its local copy of the shared future. Shared futures are constructed from futures; specifically, by moving futures into shared futures. Futures also have a member function `share()` that returns a shared future directly. This can be used with `auto` to infer types. A simple example using shared futures is given in Listing 69.

Listing 69: Simple shared future example

```
#include "scoped_thread.h"
#include <iostream>
#include <string>
#include <future>

using std::cout;
using std::endl;
using std::string;

void f(int thread_num, std::shared_future<std::string> sf) {
    cout << "thread" << thread_num << ": " << sf.get() << endl;
}

int main() {
    auto hi = []() -> string {return "hi";};
    std::shared_future<std::string> sf = std::async(hi);
    conc::scoped_thread t1{std::thread(f, 1, sf)};
    conc::scoped_thread t2{std::thread(f, 2, sf)};
    conc::scoped_thread t3{std::thread(f, 3, sf)};
    conc::scoped_thread t4{std::thread(f, 4, sf)};
}
```

4.3 Waiting with a time limit

All blocking calls introduced until now can block indefinitely. Sometimes it is useful to set a time limit on the blocking. Most of the blocking procedures have alternate functions that allow you to wait on a *duration-based* timeout and wait on an *absolute* timeout. With the former, you specify the amount of time to wait until timeout. With the latter, you give a specific point in time in which to timeout. Duration-based functions are suffixed with `_for` and absolute functions are suffixed with `_until`. Before we dive into the alternate timeout functions, let's review how time works in C++.

4.3.1 Clocks

According to the C++ Standard, a clock is a class that provides four distinct pieces of information:

- The time `now`
- The type of the value used to represent the times obtained from the clock
- The tick period of the clock
- Whether or not the clock ticks at a uniform rate and is thus considered to be a *steady* clock

The current time of the clock can be displayed by calling the static member function `now`. The return of `now` is specified by the `time_point` `typedef` member of the clock. Unfortunately, times are not so easily printable. This is shown in Listing 70.

Listing 70: Getting the current time using now

```cpp
#include <iostream>
#include <chrono>
#include <ctime>

template <typename Clock>
void print_now() {
    std::cout << Clock::now().time_since_epoch().count() << std::endl;
}

int main() {
    print_now<std::chrono::system_clock>();
    print_now<std::chrono::steady_clock>();
    print_now<std::chrono::high_resolution_clock>();
}
```

The tick period of the clock is specified as a fractional number of seconds given by the `period` `typedef` member. For example, a clock that ticks 25 times per second has a period of `std::ratio<1,25>`. Periods are shown in Listing 71.

Listing 71: Getting clock periods

```cpp
#include <iostream>
#include <chrono>
#include <ratio>

int main() {
    using std::chrono::system_clock;
    using std::chrono::steady_clock;
    using std::chrono::high_resolution_clock;

    std::cout << system_clock::period::num
              << "/"
              << system_clock::period::den
              << std::endl;

    std::cout << steady_clock::period::num
              << "/"
              << steady_clock::period::den
              << std::endl;

    std::cout << high_resolution_clock::period::num
              << "/"
              << high_resolution_clock::period::den
              << std::endl;
}
```

If a clock ticks at a uniform rate and cannot be adjusted, it is considered a *steady* clock. If a clock is steady, the `is_steady` static data memeber is `true`.

4.3.2 Durations

Durations are represented as a `std::chrono::duration<Rep, Period>` where `Rep` is the type of representation such as `int` or `double` and `Period` is the number of seconds a single unit of the duration represents. The C++ Standard Library offers many predefined typedefs including `std::chrono::seconds` and `std::chrono::milliseconds`. Explicit duration casts can be done with `std::chrono::duration_cast<>()`. The result of a duration cast is truncated. Durations also support regular arithmetic operators. This is shown in Listing 72.

Listing 72: Chrono durations

```cpp
#include <iostream>
#include <chrono>

using std::cout;
using std::endl;
using std::chrono::duration_cast;

typedef std::chrono::milliseconds MS;
typedef std::chrono::seconds S;
typedef std::chrono::minutes M;
typedef std::chrono::hours H;

void print_times(S s) {
    cout << s.count()                   << " seconds"      << endl;
    cout << MS(s).count()               << " milliseconds" << endl;
    cout << duration_cast<M>(s).count() << " minutes"      << endl;
    cout << duration_cast<H>(s).count() << " hours"        << endl;
    cout <<                                                   endl;
}

int main() {
    S s(1);
    print_times(s);
    print_times(s*60);
    print_times(s*60*60);
    print_times(s*60*60*24);
    print_times(s*60*60*24*365);
}
```

You can use durations with `*_for()` functions. Listing 73 shows a timeout on an asynchronous instance.

Listing 73: Waiting with a duration timeout

```cpp
#include <iostream>
#include <future>
#include <chrono>

int main() {
    std::chrono::seconds sec(1);

    auto sleep = [sec] {
        std::this_thread::sleep_for(5 * sec);
        return 43;
    };
    std::future<int> f = std::async(std::launch::async, sleep);

    std::cout << "waiting" << std::endl;
    std::future_status status;
    do {
        status = f.wait_for(sec);
        switch (status) {
            case std::future_status::deferred:
                std::cout << "deferred" << std::endl;
                break;
            case std::future_status::timeout:
                std::cout << "timed out" << std::endl;
                break;
            case std::future_status::ready:
                std::cout << "ready" << std::endl;
                break;
        }
    } while(status != std::future_status::ready);

    std::cout << "future: " << f.get() << std::endl;
}
```

4.3.3 Time points

The time point for a clock is represented as an instance of `std::chrono::time_point<>`. The first template parameter is a clock and the second is a duration. The value of a time point is the length of time, in multiples of the duration, since the *epoch* of the clock. The epoch is not defined by the standard or queryable. Typical epochs include 00:00 on January

1, 1970 or the boot time of the computer running the application. You can get the time since the epoch using `time_since_epoch()`.

You can subtract and add durations from instances of time points to get other timepoints. You can also subtract two time points on the same clock. This is shown in Listing 74

Listing 74: Using timepoints to time an operation

```cpp
#include <iostream>
#include <chrono>
#include <vector>
#include <thread>

using std::cout;
using std::endl;
using std::chrono::high_resolution_clock;
using std::chrono::seconds;
using std::chrono::milliseconds;
using std::chrono::duration_cast;

int main() {
    auto f = []() -> void {
        std::this_thread::sleep_for(seconds(1));
    };

    auto start = high_resolution_clock::now();
    f();
    auto stop  = high_resolution_clock::now();

    auto time = duration_cast<milliseconds>(stop - start).count();
    std::cout << time << " milliseconds" << std::endl;
}
```

You can also use timepoints to set absolute timeouts. This is shown in Listing 75.

Listing 75: Waiting with an absolute timeout

```cpp
#include <iostream>
#include <future>
#include <chrono>
#include <thread>

using namespace std::chrono;

int pause() {
    std::this_thread::sleep_for(seconds(3));
    return 43;
}

int main() {
    const auto timeout = steady_clock::now() + seconds(2);
    const auto to = std::future_status::timeout;

    std::future<int> f1 = std::async(std::launch::async, pause);
    std::future<int> f2 = std::async(std::launch::async, pause);
    std::future<int> f3 = std::async(std::launch::async, pause);

    std::future_status s1 = f1.wait_until(timeout);
    std::future_status s2 = f2.wait_until(timeout);
    std::future_status s3 = f3.wait_until(timeout);

    if (to == s1 || to == s2 || to == s3) {
        std::cout << "someone timed out!" << std::endl;
    }
    else {
        std::cout << "future: " << f1.get() + f2.get() + f3.get() << std::endl;
    }
}
```

4.3.4 Functions that accept timeouts

The following list includes the classes and namespaces that include functions that accept timeouts.

- `std::this_thread`
- `std::condition_variable`
- `std::condition_variable_any`
- `std::timed_mutex`
- `std::recursive_timed_mutex`
- `std::unique_lock`
- `std::future`
- `std::shared_future`

4.4 Using synchronization of operations to simplify code

The synchronization utilities presented in this chapter allow for a concurrent, functional style of programming.

4.4.1 Functional programming with futures

Purely functional programs are easy to reason about concurrently, because they do not modify shared state. While the functional paradigm is not C++'s principle paradigm, C++ does offer many functional functionalities. Futures can be used to support functional concurrency: the results of computations can be passed around *without any explicit access to shared data.*

FP-style quicksort A sequential implementation of quick sort is given in Listing 76.

Listing 76: Sequential functional quicksort

```cpp
#include <iostream>
#include <list>
#include <algorithm>
#include <string>

template <typename T>
std::list<T> quicksort(std::list<T> l) {
    // base case
    if (l.size() <= 1) {
        return l;
    }

    // behead l into sorted
    std::list<T> sorted;
    sorted.splice(sorted.begin(), l, l.begin());
    const T& pivot = *(sorted.begin());

    // partition l
    auto less = [&](const T& t) {return t < pivot;};
    auto split = std::partition(l.begin(), l.end(), less);

    // create lower
    std::list<T> lower;
    lower.splice(lower.end(), l, l.begin(), split);

    // recursively sort
    auto new_lower = quicksort(std::move(lower));
    auto new_upper = quicksort(std::move(l));

    // combine and return
    sorted.splice(sorted.end(), new_upper);
    sorted.splice(sorted.begin(), new_lower);
    return sorted;
}

template <typename T>
std::ostream& operator<<(std::ostream& os, const std::list<T>& l) {
    std::for_each(l.begin(), l.end(), [&](const T& t){os << t << " ";});
    return os;
}

int main() {
    std::list<int> il{6,3,2,9,5,8,1,4,7};
    std::list<char> cl{'g','y','s','u','q','b','o','v'};
    std::list<std::string> sl{"who", "goes", "there"};

    std::cout << quicksort(il) << std::endl;
    std::cout << quicksort(cl) << std::endl;
    std::cout << quicksort(sl) << std::endl;
}
```

FP-style parallel quicksort A parallel implementation of quick sort is given in Listing 77. The major difference from Listing 76 is the future used to sort the bottom half of the list.

Listing 77: Parallel functional quicksort

```cpp
#include <iostream>
#include <list>
#include <algorithm>
#include <string>
#include <future>
#include <thread>

template <typename T>
std::list<T> quicksort(std::list<T> l) {
    // base case
    if (l.size() <= 1) {
        return l;
    }

    // behead l into sorted
    std::list<T> sorted;
    sorted.splice(sorted.begin(), l, l.begin());
    const T& pivot = *(sorted.begin());

    // partition l
    auto less = [&](const T& t) {return t < pivot;};
    auto split = std::partition(l.begin(), l.end(), less);

    // create lower
    std::list<T> lower;
    lower.splice(lower.end(), l, l.begin(), split);

    // recursively sort
    auto new_lower = std::async(quicksort<T>, std::move(lower));
    auto new_upper = quicksort(std::move(l));

    // combine and return
    sorted.splice(sorted.end(), new_upper);
    sorted.splice(sorted.begin(), new_lower.get());
    return sorted;
}

template <typename T>
std::ostream& operator<<(std::ostream& os, const std::list<T>& l) {
    std::for_each(l.begin(), l.end(), [&](const T& t){os << t << " ";});
    return os;
}

int main() {
    std::list<int> il{6,3,2,9,5,8,1,4,7};
    std::list<char> cl{'g','y','s','u','q','b','o','v'};
    std::list<std::string> sl{"who", "goes", "there"};

    std::cout << quicksort(il) << std::endl;
    std::cout << quicksort(cl) << std::endl;
    std::cout << quicksort(sl) << std::endl;
}
```

Depending on your implementation of `std::async`, the parallel implementation of quicksort may or may not speed up performance.

4.4.2 Synchronizing operations with message passing

Another programming paradigm that avoids the use of shared mutable data is CSP, Communicating Sequential Processes. There does not exist shared data between threads but there do exist communication channels allowing messages to be passed between them. CSP is the paradigm adopted by MPI.

The idea of CSP is simple: *if there's no shared data, each thread can be reasoned about entirely independently, purely on the basis of how it behaves in response to the messages that it received.* Essentially, each thread is a finite state machine. When it receives a message, it updates it state, performs some operations, and perhaps sends messages with the result of the processing.

True communicating sequential processes share no data. C++ threads share an address space, so it is up to the implementer to ensure that threads do not share any data aside from the message queues.

Imagine you are implementing an ATM machine. You could divide the program into three communicating sequential threads, as in the *Actor model*.

- A thread to handle the **physical machine**
- A thread to handle **ATM logic**
- A thread to communicate with the **bank**

The ATM logic can be modelled with a finite state machine, as shown in 1. The logic can be implemented as a class with a method for each state.

The full implementation of the ATM system is left to the book itself.

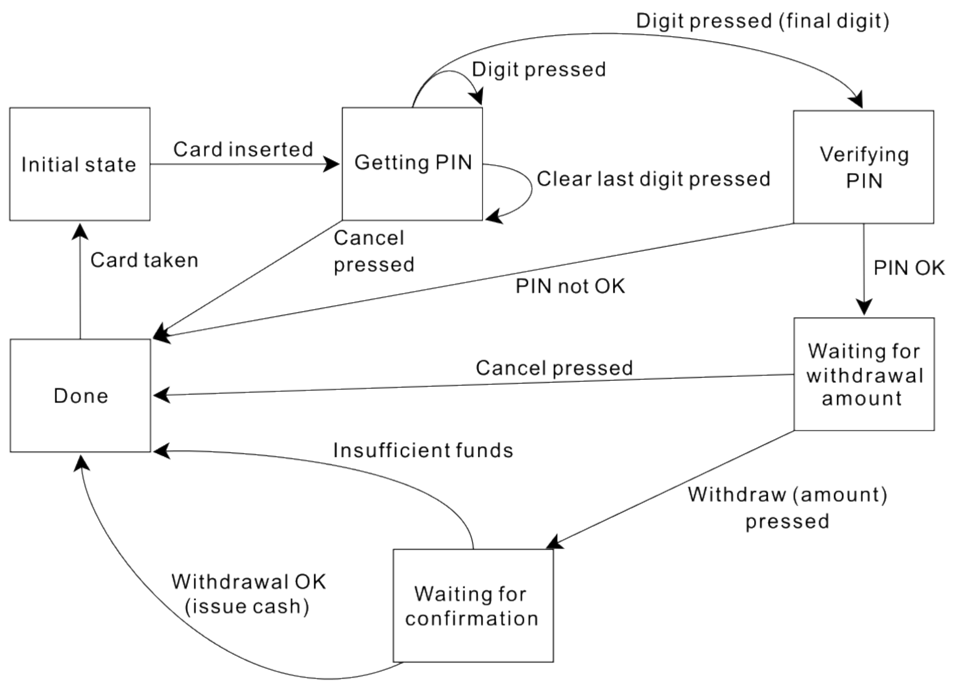

Figure 1: A simple state machine model for an ATM

5 The C++ memory model and operations on atomic types

6 Designing lock-based concurrent data structures

7 Designing lock-free concurrent data structures

8 Designing concurrent code

9 Advanced thread management

10 Testing and debugging multithreaded applications