

Michael Whittaker

# C++ Concurrency In Action

*Thoughts and Notes*

[cppconcurrency github](https://github.com/mwhittaker/cppconcurrency)

# Contents

<b>1</b>	<b>Hello, world of concurrency in C++!</b>	<b>5</b>
1.1	What is concurrency? . . . . .	5
1.1.1	Concurrency in computer systems . . . . .	5
1.1.2	Approaches to concurrency . . . . .	5
1.2	Why use concurrency? . . . . .	6
1.2.1	Using concurrency for separation of concerns . . . . .	6
1.2.2	Using concurrency for performance . . . . .	6
1.2.3	When not to use concurrency . . . . .	7
1.3	Concurrency and multithreading in C++ . . . . .	7
1.3.1	History of multithreading in C++ . . . . .	7
1.3.2	Concurrency support in the new standard . . . . .	7
1.3.3	Efficiency in the C++ Thread Library . . . . .	8
1.3.4	Platform-specific facilities . . . . .	8
1.4	Getting Started . . . . .	8
1.4.1	Hello, Concurrent World . . . . .	8
<b>2</b>	<b>Managing threads</b>	<b>9</b>
2.1	Basic thread management . . . . .	9
2.1.1	Launching a thread . . . . .	9
2.1.2	Waiting for a thread to complete . . . . .	10
2.1.3	Waiting in exceptional circumstances . . . . .	11
2.1.4	Running threads in the background . . . . .	13
2.2	Passing arguments to a thread function . . . . .	15
2.3	Transferring ownership of a thread . . . . .	17
2.4	Choosing the number of threads at runtime . . . . .	21
2.5	Identifying threads . . . . .	24
<b>3</b>	<b>Sharing data between threads</b>	<b>25</b>
3.1	Problems with sharing data between threads . . . . .	25
3.1.1	Race conditions . . . . .	25
3.1.2	Avoiding problematic race conditions . . . . .	25
3.2	Protecting share data with mutexes . . . . .	25
3.2.1	Using mutexes in C++ . . . . .	26
3.2.2	Structuring code for protecting share data . . . . .	26
3.2.3	Spotting race conditions inherent in interfaces . . . . .	28
3.2.4	Deadlock: the problem and a solution . . . . .	32

3.2.5	Further guidelines for avoiding deadlock . . . . .	33
3.2.6	Flexible locking with <code>std::unique_lock</code> . . . . .	38
3.2.7	Transferring ownership between scopes . . . . .	39
3.2.8	Locking at an appropriate granularity . . . . .	40
3.3	Alternative facilities for protecting shared data . . . . .	41
3.3.1	Protecting shared data during initialization . . . . .	41
3.3.2	Protecting rarely updated data structures . . . . .	45
3.3.3	Recursive locking . . . . .	47
4	<b>Synchronizing concurrent operations</b>	<b>48</b>
5	<b>The C++ memory model and operations on atomic types</b>	<b>49</b>
6	<b>Designing lock-based concurrent data structures</b>	<b>50</b>
7	<b>Designing lock-free concurrent data structures</b>	<b>51</b>
8	<b>Designing concurrent code</b>	<b>52</b>
9	<b>Advanced thread management</b>	<b>53</b>
10	<b>Testing and debugging multithreaded applications</b>	<b>54</b>

## Listings

1	A simple Hello Concurrent World program . . . . .	8
2	Spawning a thread using a functor . . . . .	9
3	Spawning a thread using a lambda . . . . .	10
4	Dangling references and threads . . . . .	10
5	You can only join a thread once . . . . .	11
6	Try-catch for exceptional joins . . . . .	11
7	thread guard header file . . . . .	11
8	thread guard cpp file . . . . .	13
9	Joining using thread-guard . . . . .	13
10	Detaching a thread to handle other documents . . . . .	14
11	Detaching a thread until the main thread terminates . . . . .	14
12	A multiple argument gotcha . . . . .	15
13	A multiple argument gotcha resolved . . . . .	15
14	Failing to pass a reference to a thread . . . . .	16
15	Succeeding in passing a reference to a thread . . . . .	16
16	Initializing thread with method . . . . .	16
17	Moving data into a thread. . . . .	17
18	Moving Threads . . . . .	17
19	Returning a thread from a function . . . . .	17
20	Passing a thread into a function . . . . .	18
21	Scoped thread header . . . . .	18
22	Scoped thread cpp . . . . .	20
23	Scoped thread example . . . . .	20
24	Making a vector of threads. . . . .	20
25	Displaying the number of hardware cores . . . . .	21
26	Fast accumulation . . . . .	21
27	Slow accumulation . . . . .	22
28	Fast accumulation header . . . . .	22
29	Fast accumulation “source” . . . . .	23
30	Fun with thread ids . . . . .	24
31	Protecting a list with a mutex . . . . .	26
32	Accidentally passing out a reference to protected data. . . . .	27
33	The interface to the <code>std::stack</code> container adapter . . . . .	28
34	The problem with the standard stack interface . . . . .	28
35	Concurrent stack header . . . . .	29
36	Concurrent stack “source” . . . . .	30
37	Example with concurrent stack . . . . .	31
38	Using <code>std::lock</code> and <code>std::lock_guard</code> in a swap operation . . . . .	32
39	Hierarchical mutex header . . . . .	34
40	Hierarchical mutex source . . . . .	35
41	Hierarchical mutex conforming example . . . . .	36
42	Hierarchical mutex violating example . . . . .	37
43	Using <code>std::lock()</code> and <code>std::unique_lock</code> in a swap operation . . . . .	38

44	Moving a unique lock . . . . .	39
45	Obtaining appropriate lock granularity using <code>std::unique_lock</code> . . . . .	40
46	Fine-grained locking on two int wrappers. . . . .	40
47	Simple lazy construction . . . . .	41
48	Lazy construction using a mutex that serializes code . . . . .	42
49	Double-checked locking undefined lazy construction . . . . .	43
50	Lazy construction using <code>std::call_once</code> and <code>std::once_flag</code> . . . . .	44
51	Lazy construction using <code>std::call_once</code> and <code>std::once_flag</code> in a class . .	44
52	DNS cache using <code>boost::shared_mutex</code> and <code>boost::shared_lock</code> . . . . .	46

## 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](mailto: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

```
1 #include <iostream>
2 #include <thread>
3
4 void hello() {
5     std::cout << "Hello Concurrent World!" << std::endl;
6 }
7
8 int main() {
9     std::thread t(hello);
10    t.join();
11 }
```

. 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.

```
1 void do_work();
2 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

```
1 #include <iostream>
2 #include <thread>
3
4 struct background_task {
5     int x_;
6     background_task(int x): x_(x) {}
7     void operator()() const {
8         std::cout << "Hello from background_task " << x_ << std::endl;
9     }
10 };
11
12 int main() {
13     background_task b1(1);
14     std::thread t1(b1);                // explicit initialization
```

```

15     std::thread t2((background_task(2))); // extra parentheses
16     std::thread t3{background_task(3)};  // uniform initialization
17
18     t1.join();
19     t2.join();
20     t3.join();
21 }

```

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

Listing 3: Spawning a thread using a lambda

```

1 #include <iostream>
2 #include <thread>
3
4 int main() {
5     std::thread t([]() -> void {std::cout << "hi" << std::endl;});
6     t.join();
7 }

```

**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

```

1 #include <iostream>
2 #include <thread>
3
4 struct fun {
5     int& x;
6     fun(int x_): x(x_) {}
7     void operator()() const {
8         for (int i = 0; i < 1000000; ++i) {
9             ++x;
10            --x;
11        }
12    }
13 };
14
15 int main() {
16     int local_state = 0;
17     std::thread t{fun(local_state)};
18     t.detach();
19 }

```

### 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

```
1 #include <iostream>
2 #include <thread>
3
4 int main() {
5     std::thread t([]() -> void {});
6     t.join();
7     std::cout << "t is joinable? " << t.joinable() << std::endl;
8 }
```

### 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

```
1 struct fun;
2
3 void f() {
4     int local_state = 0;
5     std::thread t{fun(local_state)};
6     try {
7         // stuff
8     }
9     catch (/*something*/) {
10         t.join();
11         throw;
12     }
13     t.join();
14 }
```

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

```
1 #ifndef THREAD_GUARD_H
2 #define THREAD_GUARD_H
3
```

```

4 #include <thread>
5
6 namespace conc {
7
8 /*****
9  * \brief This class uses RAII to join a thread.
10  *
11  * A thread_guard instance is instantiated with an <std::thread>.
12  * When the instance goes out of scope, the destructor will join the thread if
13  * it is joinable. Otherwise, it does nothing.
14  *
15  * RAII is a powerful way to guarantee a thread is joined even in the face of
16  * exceptional circumstances. Other means of joining despite exceptions, like
17  * try-catch blocks, are more verbose and more prone to bugs.
18  *****/
19 class thread_guard {
20 public:
21 /*****
22  * Constructs a thread_guard instance from <t_>.
23  *
24  * <t_> will be joined when this thread_guard instance goes out of scope,
25  * if possible.
26  *
27  * \param t_ A thread to be joined by the destructor.
28  * \see thread_guard::~~thread_guard()
29  *****/
30 explicit thread_guard(std::thread& t_);
31
32 /*****
33  * Joins the thread associated with this thread_guard instance.
34  *
35  * If <t.joinable()> is true, <t> is joined. Otherwise, nothing happens.
36  *****/
37 ~thread_guard();
38
39 /*****
40  * Deleted copy constructor.
41  *****/
42 thread_guard(const thread_guard&) = delete;
43
44 /*****
45  * Deleted copy assignment operator.
46  *****/
47 thread_guard& operator=(const thread_guard&) = delete;
48 private:
49 /*****
50  * The thread managed by this instance of thread_guard. The thread will be
51  * joined by the destructor if possible.
52  *****/
53 std::thread& t;
54 };
55
56 } // namespace conc
57

```

```
58 #endif // THREAD_GUARD_H
```

#### Listing 8: thread\_guard.cpp file

```
1 #include "thread_guard.h"
2 #include <thread>
3
4 namespace conc {
5
6 thread_guard::thread_guard(std::thread& t_): t(t_) {}
7
8 thread_guard::~thread_guard() {
9     if (t.joinable()) {
10         t.join();
11     }
12 }
13
14 } // namespace conc
```

#### Listing 9: Joining using thread-guard

```
1 #include "thread_guard.h"
2 #include <iostream>
3 #include <thread>
4
5 using namespace conc;
6
7 struct fun {
8     int& x;
9     fun(int x_): x(x_) {}
10    void operator()() const {
11        for (int i = 0; i < 1000000; ++i) {
12            ++x;
13            --x;
14        }
15    }
16 };
17
18 int main() {
19     int local_state = 0;
20     std::thread t{fun(local_state)};
21     thread_guard tg(t);
22 }
```

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*.

```
1 std::thread t(do_background_work);
2 t.detach();
3 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

```
1 void edit_document(const std::string& filename) {
2     open_document_and_display_gui(filename);
3     while(!done_editing()) {
4         user_command cmd = get_user_input();
5         if (cmd.type == open_new_document) {
6             const std::string new_name = get_filename_from_user();
7             std::thread t(edit_document, new_name);
8             t.detach();
9         }
10        else {
11            process_user_input(cmd);
12        }
13    }
14 }
```

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

```
1 #include <iostream>
2 #include <string>
3 #include <algorithm>
4 #include <chrono>
5 #include <thread>
6
7 void cin_print() {
8     std::string s;
9     std::cin >> s;
10    std::reverse(s.begin(), s.end());
11    std::cout << "You typed " << s << std::endl;
12 }
13
14 int main() {
15     std::thread(cin_print).detach();
16     std::this_thread::sleep_for(std::chrono::seconds(5));
```



```
17 }
```

## 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

```
1 #include <iostream>
2 #include <thread>
3 #include <cstdio>
4 #include <chrono>
5
6 void f(int i, const std::string& s) {
7     std::cout << i << s << std::endl;
8 }
9
10 void oops() {
11     char buffer[1024];
12     sprintf(buffer, "%i", 42);
13     std::thread(f, 42, buffer).detach();
14 }
15
16 int main() {
17     oops();
18 }
```

Listing 13: A multiple argument gotcha resolved

```
1 #include <iostream>
2 #include <string>
3 #include <thread>
4 #include <cstdio>
5
6 void f(int i, const std::string& s) {
7     std::cout << i << s << std::endl;
8 }
9
10 void not_oops() {
11     char buffer[1024];
12     sprintf(buffer, "%i", 42);
13     std::thread(f, 42, std::string(buffer)).detach();
14 }
15
16 int main() {
```

```
17     not_oops();
18 }
```

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

```
1 #include <iostream>
2 #include <thread>
3
4 void f(int& i) {
5     ++i;
6 }
7
8 int main() {
9     int i = 0;
10    std::thread t(f, i);
11    thread_guard(t);
12 }
```

Listing 15: Succeeding in passing a reference to a thread

```
1 #include <iostream>
2 #include <thread>
3
4 void f(int& i) {
5     ++i;
6 }
7
8 int main() {
9     int i = 0;
10    std::thread t(f, std::ref(i));
11    t.join();
12    std::cout << i << std::endl;
13 }
```

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

```
1 #include <iostream>
2 #include <thread>
3
4 struct Foo {
5     void bar(int i) {
6         std::cout << "bar " << i << std::endl;
7     }
8 };
```

```

9
10 int main() {
11     Foo foo;
12     std::thread t(&Foo::bar, &foo, 43);
13     t.join();
14 }

```

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

#### Listing 17: Moving data into a thread.

```

1 #include <iostream>
2 #include <thread>
3 #include <memory>
4
5 void foo(std::unique_ptr<int> ip) {
6     (*ip)++;
7     std::cout << *ip << std::endl;
8 }
9
10 int main() {
11     std::thread t1(foo, std::unique_ptr<int>(new int(42)));
12
13     std::unique_ptr<int> ip(new int(0));
14     std::thread t2(foo, std::move(ip));
15
16     t1.join();
17     t2.join();
18 }

```

## 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

```

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

```

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

#### Listing 19: Returning a thread from a function

```

1 #include <iostream>
2 #include <thread>

```

```

3
4 void f() {
5     std::cout << "f" << std::endl;
6 }
7
8 void g(int i) {
9     std::cout << "g" << i << std::endl;
10 }
11
12 std::thread getf() {
13     return std::thread(f);
14 }
15
16 std::thread getg() {
17     std::thread t(g, 42);
18     return t;
19 }
20
21 int main() {
22     std::thread ft(getf());
23     std::thread gt(getg());
24     ft.join();
25     gt.join();
26 }

```

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

#### Listing 20: Passing a thread into a function

```

1 #include <iostream>
2 #include <thread>
3
4 void join(std::thread t) {
5     t.join();
6 }
7
8 int main() {
9     join(std::thread([]() -> void {std::cout << "hi" << std::endl;});
10     std::thread t([]() -> void {std::cout << "bye" << std::endl;});
11     join(std::move(t));
12 }

```

One benefit of the movability of `std::threads` 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

```

1 #ifndef SCOPED_THREAD_H
2 #define SCOPED_THREAD_H
3
4 #include <thread>
5

```

```

6 namespace conc {
7
8  /*****
9   * \brief This class takes ownership of a thread and joins it using RAII.
10  *
11  * An instance of a scoped_thread takes ownership of a thread by moving it into
12  * the instance. It verifies that the thread is joinable in the constructor and
13  * throws an exception if it is not. Then, it joins the thread in the
14  * destructor. Copy and assignment operators have been deleted to avoid
15  * erroneous copies.
16  *
17  * Class Invariants:
18  * - t is joinable.
19  * - Upon destruction of a scoped_thread object, the thread is owned will be
20  *   joined.
21  *****/
22 class scoped_thread {
23 public:
24     /*****
25      * Constructs a scoped_thread instance with <t_>.
26      *
27      * If <t_> is not joinable, an std::logic_error is thrown. Otherwise, this
28      * instance successfully moves t_ into a local copy. NOTE that the thread
29      * <t_> is MOVED. If you instantiate a scoped_thread instance with an
30      * lvalue, do not attempt to touch that variable again!
31      *
32      * <t_> will be joined by the destructor.
33      *
34      * \param t_ A thread to be moved into this instance.
35      * \see scoped_thread::~scoped_thread()
36      *****/
37     explicit scoped_thread(std::thread t_);
38
39     /*****
40      * Joins the thread associated with this instance.
41      *****/
42     ~scoped_thread();
43
44     /*****
45      * Deleted copy constructor.
46      *****/
47     scoped_thread(const scoped_thread&) = delete;
48
49     /*****
50      * Deleted copy assignment operator.
51      *****/
52     scoped_thread& operator=(const scoped_thread&) = delete;
53
54 private:
55     /*****
56      * The thread managed by this instance of scoped_thread.
57      *
58      * This thread is guaranteed to be joinable and will be joined in the
59      * destructor of this instance.

```

```

60      *****/
61      std::thread t;
62  };
63
64  } // namespace conc
65
66  #endif // SCOPED_THREAD_H

```

#### Listing 22: Scoped thread cpp

```

1  #include "scoped_thread.h"
2  #include <thread>
3
4  namespace conc {
5
6  scoped_thread::scoped_thread(std::thread t_): t(std::move(t_)) {
7      if (!t.joinable()) {
8          throw std::logic_error("scoped_thread constructed from non-joinable");
9      }
10 }
11 scoped_thread::~scoped_thread() {
12     t.join();
13 }
14
15 } // namespace conc

```

#### Listing 23: Scoped thread example

```

1  #include "scoped_thread.h"
2  #include <iostream>
3  #include <thread>
4  #include <chrono>
5
6  using namespace conc;
7
8  void f() {
9      std::this_thread::sleep_for(std::chrono::seconds(2));
10     std::cout << "bye!" << std::endl;
11 }
12
13 int main() {
14     scoped_thread st{std::thread(f)};
15     std::cout << "hi" << std::endl;
16 }

```

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.

```

1  #include <iostream>
2  #include <vector>

```

```

3 #include <thread>
4 #include <chrono>
5 #include <algorithm>
6
7 int main() {
8     auto f = [](uint i) -> void {
9         std::this_thread::sleep_for(std::chrono::seconds(2));
10        std::cout << i << std::endl;
11    };
12
13    std::vector<std::thread> threads;
14    for (uint i = 0; i <= 20; ++i) {
15        threads.push_back(std::thread(f,i));
16        std::this_thread::sleep_for(std::chrono::milliseconds(100));
17    }
18
19    std::for_each(threads.begin(), threads.end(),
20                std::mem_fn(&std::thread::join));
21
22    std::cout << "bye!" << std::endl;
23 }

```

## 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

```

1 #include <iostream>
2 #include <thread>
3
4 int main() {
5     std::cout << std::thread::hardware_concurrency() << std::endl;
6 }

```

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

```

1 #include "conc_numeric.h"
2 #include <iostream>
3 #include <vector>
4 #include <chrono>
5
6 int main() {
7     std::vector<int> v(100000000, 1);
8
9     using std::chrono::duration_cast;
10    using std::chrono::nanoseconds;

```

```

11     typedef std::chrono::high_resolution_clock clock;
12
13     auto start = clock::now();
14     std::cout << conc::accumulate(v.begin(), v.end(), 0) << std::endl;
15     auto end = clock::now();
16     std::cout << duration_cast<nanoseconds>(end-start).count()/1000000 << "ns\n";
17 }

```

#### Listing 27: Slow accumulation

```

1  #include "conc_numeric.h"
2  #include <iostream>
3  #include <vector>
4  #include <chrono>
5
6  int main() {
7      std::vector<int> v(100000000, 1);
8
9      using std::chrono::duration_cast;
10     using std::chrono::nanoseconds;
11     typedef std::chrono::high_resolution_clock clock;
12
13     auto start = clock::now();
14     std::cout << std::accumulate(v.begin(), v.end(), 0) << std::endl;
15     auto end = clock::now();
16     std::cout << duration_cast<nanoseconds>(end-start).count()/1000000 << "ns\n";
17 }

```

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

#### Listing 28: Fast accumulation header

```

1  #ifndef CONC_NUMERIC_H
2  #define CONC_NUMERIC_H
3
4  #include <thread>
5  #include <algorithm>
6  #include <numeric>
7
8  namespace conc {
9
10 /*****
11  * Concurrent implementation of std::accumulate.
12  *
13  * Computes the sum of the given value init and the elements in the range
14  * [first, last). This version uses operator+ to sum up the elements.
15  * Everything is done concurrently.
16  *****/
17 template <typename Iterator, typename T>
18 T accumulate(Iterator first, Iterator last, T init);
19
20 namespace details {
21
22 /*****

```



```

23  * Helper functor for accumulate.
24  *
25  * Invokes std::accumulate on first, last, result and stores the result to
26  * result.
27  *
28  * \see std::accumulate()
29  * \see conc::accumulate()
30  *****/
31  template <typename Iterator, typename T>
32  struct accumulate_block {
33      void operator()(Iterator first, Iterator last, T& result);
34  };
35
36  } // namespace details
37  } // namespace conc
38
39  #include "conc_numeric.hpp"
40
41  #endif // CONC_NUMERIC_H

```

Listing 29: Fast accumulation “source”

```

1  namespace conc {
2
3  template <typename Iterator, typename T>
4  T accumulate(Iterator first, Iterator last, T init) {
5      const ulong length = std::distance(first, last);
6
7      if (length == 0) {
8          return init;
9      }
10
11     const ulong min_per_t = 25;
12     const ulong max_ts = (length + min_per_t - 1) / min_per_t;
13     const ulong hw_ts = std::thread::hardware_concurrency();
14     const ulong num_ts = std::min(hw_ts != 0 ? hw_ts : 2, max_ts);
15     const ulong block_size = length / num_ts;
16     std::vector<T> results(num_ts);
17     std::vector<std::thread> threads(num_ts - 1);
18
19     Iterator block_start = first;
20     for (ulong i = 0; i < (num_ts - 1); ++i) {
21         Iterator block_end = block_start;
22         std::advance(block_end, block_size);
23         threads[i] = std::thread(
24             details::accumulate_block<Iterator, T>(),
25             block_start, block_end, std::ref(results[i])
26         );
27         block_start = block_end;
28     }
29     details::accumulate_block<Iterator, T>()
30         (block_start, last, results[num_ts - 1]);
31
32     std::for_each(threads.begin(), threads.end(),

```

```

33         std::mem_fn(&std::thread::join));
34
35     return std::accumulate(results.begin(), results.end(), init);
36 }
37
38
39 namespace details {
40
41 template <typename Iterator, typename T>
42 void accumulate_block<Iterator, T>::operator()(Iterator first, Iterator last,
43         T& result) {
44     result = std::accumulate(first, last, result);
45 }
46
47 } // namespace details
48 } // 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

```

1 #include <iostream>
2 #include <thread>
3
4 int main() {
5     auto print_id = []() -> void {
6         std::cout << std::this_thread::get_id() << std::endl;
7     };
8
9     std::thread t(print_id);
10    t.join();
11    print_id();
12 }

```

## 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**

```

1  #include "scoped_thread.h"
2  #include <iostream>
3  #include <algorithm>
4  #include <list>
5  #include <thread>
6  #include <mutex>
7
8  std::list<int> l;
9  std::mutex l_mutex;
10
11 void add_to_list(int v) {
12     std::lock_guard<std::mutex> guard(l_mutex);
13     std::cout << "adding " << v << std::endl;
14     l.push_back(v);
15 }
16
17 bool list_contains(int v) {
18     std::lock_guard<std::mutex> guard(l_mutex);
19     return std::find(l.begin(), l.end(), v) != l.end();
20 }
21
22 int main() {
23     auto add = []() -> void {
24         for (int i = 0; i < 100; ++i) {
25             add_to_list(i);
26         }
27     };
28     conc::scoped_thread st{std::thread(add)};
29
30     for (int i = 90; i < 100; ++i) {
31         std::cout << "l has " << i << "? " << list_contains(i) << std::endl;
32     }
33 }
```

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.**

```

1  #include <iostream>
2  #include <mutex>
3
4  class some_data {
5  public:
6      void do_something() {
7          std::cout << "do_something!" << std::endl;
8      }
9  private:
10     int i;
11     std::string s;
12 };
13
14 class data_wrapper {
15 public:
16     template<typename Function>
17     void process_data(Function f) {
18         std::cout << "locking mutex" << std::endl;
19         std::lock_guard<std::mutex> lg(m);
20         f(data);
21         std::cout << "unlocking mutex" << std::endl;
22     }
23 private:
24     some_data data;
25     std::mutex m;
26 };
27
28
29 int main() {
30     some_data *unprotected;
31     data_wrapper x;
32
33     auto malicious = [&](some_data& protected_data) {
34         unprotected = &protected_data;
35     };
36
37     x.process_data(malicious);
38     unprotected->do_something();
39 }

```

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, let's consider the stack interface provided by the STL as shown in Listing 33.

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

```

1 template <typename T, typename Container = std::deque<T>>
2 class stack {
3 public:
4     /* constructors */
5     bool empty() const;
6     size_t size() const;
7     T& top();
8     T const& top() const;
9     void push(T const&);
10    void push(T&&);
11    void pop();
12    void swap(stack&&);
13 };

```

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

```

1 stack<int> s;
2 if (!s.empty()) {
3     const int value = s.top();
4     s.pop();
5     do_something(value);
6 }

```

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.

```

1 std::vector<int> result;
2 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

```

1  #ifndef CONC_STACK_H
2  #define CONC_STACK_H
3
4  #include <exception>
5  #include <mutex>
6  #include <stack>
7  #include <memory>
8
9  namespace conc {
10
11  /*****
12   *
13   *****/
14  struct empty_stack : std::exception {
15      const char *what() const throw();
16  };
17
18  /*****
19   *
20   *****/
21  template <typename T>
22  class stack {
23  public:
24      /*****
25       *
26       *****/
27      stack();
28
29      /*****
30       *
31       *****/

```

```

32     stack(const stack& other);
33
34     /*****
35      *
36      *****/
37     stack& operator=(const stack&) = delete;
38
39     /*****
40      *
41      *****/
42     void push(T val);
43
44     /*****
45      *
46      *****/
47     std::shared_ptr<T> pop();
48
49     /*****
50      *
51      *****/
52     void pop(T& value);
53
54     /*****
55      *
56      *****/
57     bool empty() const;
58
59 private:
60     /*****
61      *
62      *****/
63     std::stack<T> data;
64
65     /*****
66      *
67      *****/
68     mutable std::mutex m;
69 };
70
71 } // namespace conc
72
73 #include "conc_stack.hpp"
74
75 #endif // CONC_STACK_H

```

Listing 36: Concurrent stack “source”

```

1 namespace conc {
2
3     const char *empty_stack::what() const throw() {
4         return "empty stack";
5     }
6
7     template <typename T> stack<T>::stack() {}

```



```

8
9  template <typename T>
10 stack<T>::stack(const stack& other) {
11     std::lock_guard<std::mutex> lock(other.m);
12     data = other.data;
13 }
14
15 template <typename T>
16 void stack<T>::push(T val) {
17     std::lock_guard<std::mutex> lock(m);
18     data.push(val);
19 }
20
21 template <typename T>
22 std::shared_ptr<T> stack<T>::pop() {
23     std::lock_guard<std::mutex> lock(m);
24
25     if (data.empty()) {
26         throw empty_stack();
27     }
28
29     const std::shared_ptr<T> res(std::make_shared<T>(data.top()));
30     data.pop();
31     return res;
32 }
33
34 template <typename T>
35 void stack<T>::pop(T& val) {
36     std::lock_guard<std::mutex> lock(m);
37
38     if (data.empty()) {
39         throw empty_stack();
40     }
41
42     val = data.top();
43     data.pop();
44 }
45
46 template <typename T>
47 bool stack<T>::empty() const {
48     std::lock_guard<std::mutex> lock(m);
49     return data.empty();
50 }
51
52 } // namespace conc

```

#### Listing 37: Example with concurrent stack

```

1 #include "conc_stack.h"
2 #include "scoped_thread.h"
3 #include <iostream>
4 #include <thread>
5
6 int main() {

```

```

7     conc::stack<int> s;
8     for (int i = 0; i <= 1000000; ++i) {
9         s.push(i);
10    }
11
12    auto f = [](conc::stack<int>& s) -> void {
13        while (!s.empty()) {
14            s.pop(); // an exception can still be thrown
15        }
16    };
17
18    conc::scoped_thread st1{std::thread(f, std::ref(s))};
19    conc::scoped_thread st2{std::thread(f, std::ref(s))};
20 }

```

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

```

1 #include <iostream>
2 #include <ostream>
3 #include <mutex>
4
5 class X {
6 public:
7     X(const int& i) : detail(i) {}
8
9     friend void swap(X& lhs, X& rhs) {
10         if (&lhs == &rhs) {
11             return;
12         }
13     }

```

```

14         std::lock(lhs.m, rhs.m);
15         std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
16         std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
17         std::swap(lhs.detail, rhs.detail);
18     }
19
20     friend std::ostream& operator<<(std::ostream& stream, const X& x) {
21         stream << x.detail;
22         return stream;
23     }
24
25 private:
26     int detail;
27     std::mutex m;
28 };
29
30 int main() {
31     int i = 0;
32     int j = 42;
33     X a(i);
34     X b(j);
35
36     std::cout << a << " " << b << std::endl;
37     swap(a, b);
38     std::cout << a << " " << b << std::endl;
39 }

```

### 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

```

1  #ifndef HIERARCHICAL_MUTEX_H
2  #define HIERARCHICAL_MUTEX_H
3
4  #include <mutex>
5
6  namespace conc {
7
8  /*****
9   * A hierarchical mutex.
10  *
11  * Each mutex is assigned a priority. If a thread attempts to lock a
12  * hierarchical mutex with a priority lower than the lowest priority mutex it
13  * currently has locked, an std::logic_error is thrown. Otherwise, all lockable
14  * operations are the same.
15  *****/
16  class hierarchical_mutex {
17  public:
18      /*****
19       * Constructs a hierarchical_mutex with the given priority.
20       *
21       * \param val Priority. Lower priorities mutex must be locked after higher
22       * priority mutexes.
23       *****/
24      explicit hierarchical_mutex(ulong val);
25
26      /*****
27       * If the priority of this mutex is lower than the lowest priority mutex
28       * owned by the thread, then lock occurs just as std::mutex::lock() occurs.
29       * Otherwise, an std::logic_error is thrown.
30       *
31       * \see std::mutex::lock()
32       *****/
33      void lock();
34
35      /*****
36       * Same as std::mutex::unlock()
37       *

```

```

38     * \see std::mutex::unlock()
39     *****/
40 void unlock();
41
42 /*****/
43 * \see conc::hierarchical_mutex::lock()
44 * \see std::mutex::unlock()
45     *****/
46 bool try_lock();
47
48 private:
49 /*****/
50 * The internal mutex managed by this instance.
51     *****/
52 std::mutex m;
53
54 /*****/
55 * The priority associated with this instance.
56     *****/
57 const ulong hier_val;
58
59 /*****/
60 * The lowest priority associated with any mutex locked at the time this
61 * mutex was locked.
62     *****/
63 ulong prev_hier_val;
64
65 /*****/
66 * The lowest priority of a locked mutex in this thread.
67     *****/
68 static thread_local ulong this_thread_hier_val;
69
70 /*****/
71 * If the mutex being locked is a lower priority than the current thread's
72 * priority, an std::logic_error is thrown. Otherwise, nothing happens.
73     *****/
74 void check_hier_violation();
75
76 /*****/
77 * Updates prev_hier_val and this_thread_hier_val.
78     *****/
79 void update_hier_val();
80 };
81
82 } // namespace conc
83
84 #endif // HIERARCHICAL_MUTEX_H

```

#### Listing 40: Hierarchical mutex source

```

1 #include "hierarchical_mutex.h"
2 #include <climits>
3
4 namespace conc {

```

```

5
6 thread_local ulong hierarchical_mutex::this_thread_hier_val(ULONG_MAX);
7
8 hierarchical_mutex::hierarchical_mutex(ulong val) :
9     hier_val(val),
10    prev_hier_val(0)
11 {}
12
13 void hierarchical_mutex::lock() {
14     check_hier_violation();
15     m.lock();
16     update_hier_val();
17 }
18
19 void hierarchical_mutex::unlock() {
20     this_thread_hier_val = prev_hier_val;
21     m.unlock();
22 }
23
24 bool hierarchical_mutex::try_lock() {
25     check_hier_violation();
26     if (!m.try_lock()) {
27         return false;
28     }
29     update_hier_val();
30     return true;
31 }
32
33 void hierarchical_mutex::check_hier_violation() {
34     if (this_thread_hier_val <= hier_val) {
35         throw std::logic_error("mutex hierarchy violated");
36     }
37 }
38
39 void hierarchical_mutex::update_hier_val() {
40     prev_hier_val = this_thread_hier_val;
41     this_thread_hier_val = hier_val;
42 }
43
44 } // namespace conc

```

Listing 41: Hierarchical mutex conforming example

```

1 #include "hierarchical_mutex.h"
2 #include "scoped_thread.h"
3 #include <iostream>
4 #include <thread>
5 #include <mutex>
6
7 conc::hierarchical_mutex high_m(1000000);
8 conc::hierarchical_mutex medium_m(1000);
9 conc::hierarchical_mutex low_m(1);
10
11 void high_f();

```

```

12 void medium_f();
13 void low_f();
14
15 void high_f() {
16     std::lock_guard<conc::hierarchical_mutex> lock(high_m);
17     std::cout << "high" << std::endl;
18     medium_f();
19 }
20
21 void medium_f() {
22     std::lock_guard<conc::hierarchical_mutex> lock(medium_m);
23     std::cout << "medium" << std::endl;
24     low_f();
25 }
26
27 void low_f() {
28     std::lock_guard<conc::hierarchical_mutex> lock(low_m);
29     std::cout << "low" << std::endl;
30 }
31
32 int main() {
33     conc::scoped_thread high_t{std::thread(high_f)};
34 }

```

#### Listing 42: Hierarchical mutex violating example

```

1 #include "hierarchical_mutex.h"
2 #include "scoped_thread.h"
3 #include <iostream>
4 #include <thread>
5 #include <mutex>
6
7 conc::hierarchical_mutex high_m(20000000);
8 conc::hierarchical_mutex medium_m(2000);
9 conc::hierarchical_mutex low_m(2);
10 conc::hierarchical_mutex oops_m(1);
11
12 void high_f();
13 void medium_f();
14 void low_f();
15 void oops_f();
16
17 void high_f() {
18     std::lock_guard<conc::hierarchical_mutex> lock(high_m);
19     std::cout << "high" << std::endl;
20     medium_f();
21 }
22
23 void medium_f() {
24     std::lock_guard<conc::hierarchical_mutex> lock(medium_m);
25     std::cout << "medium" << std::endl;
26     low_f();
27 }
28

```

```

29 void low_f() {
30     std::lock_guard<conc::hierarchical_mutex> lock(low_m);
31     std::cout << "low" << std::endl;
32 }
33
34 void oops_f() {
35     std::lock_guard<conc::hierarchical_mutex> oops_lock(oops_m);
36     high_f();
37 }
38
39 int main() {
40     conc::scoped_thread oops_t{std::thread(oops_f)};
41 }

```

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

```

1  #include <iostream>
2  #include <ostream>
3  #include <mutex>
4
5  class X {
6  public:
7      X(const int& i) : detail(i) {}
8
9      friend void swap(X& lhs, X& rhs) {
10         if (&lhs == &rhs) {
11             return;
12         }
13
14         std::unique_lock<std::mutex> lock_a(lhs.m, std::defer_lock);
15         std::unique_lock<std::mutex> lock_b(rhs.m, std::defer_lock);
16         std::cout << "lock_a owns lock? " << lock_a.owns_lock() << std::endl;
17         std::cout << "lock_b owns lock? " << lock_b.owns_lock() << std::endl;

```



```

18
19     std::lock(lock_a, lock_b);
20     std::cout << "lock_a owns lock? " << lock_a.owns_lock() << std::endl;
21     std::cout << "lock_b owns lock? " << lock_b.owns_lock() << std::endl;
22
23     std::swap(lhs.detail, rhs.detail);
24 }
25
26 friend std::ostream& operator<<(std::ostream& stream, const X& x) {
27     stream << x.detail;
28     return stream;
29 }
30
31 private:
32     int detail;
33     std::mutex m;
34 };
35
36 int main() {
37     int i = 0;
38     int j = 42;
39     X a(i);
40     X b(j);
41
42     std::cout << a << " " << b << std::endl;
43     swap(a, b);
44     std::cout << a << " " << b << std::endl;
45 }

```

### 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 be done by moving, but not copying, the unique lock as shown in Listing 44.

Listing 44: Moving a unique lock

```

1  #include <iostream>
2  #include <mutex>
3
4  std::mutex m;
5
6  std::unique_lock<std::mutex> get_lock() {
7      std::unique_lock<std::mutex> lock(m);
8      std::cout << "got lock" << std::endl;
9      return lock;
10 }
11
12 int main() {
13     std::unique_lock<std::mutex> lock(get_lock());
14     std::cout << "main" << std::endl;
15 }

```

### 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`**

```

1 void get_and_process_data() {
2     std::unique_lock<std::mutex> lock(m);
3     some_class data_to_process = get_next_data_chunk();
4     lock.unlock();
5     result_type result = process(data_to_process);
6     lock.lock();
7     write_result(data_to_process, result);
8 }
```

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.**

```

1 #include <iostream>
2 #include <mutex>
3
4 class Integer {
5 public:
6     Integer(int data_) : data(data_) {}
7
8     friend bool operator==(const Integer& lhs, const Integer& rhs) {
9         return &lhs == &rhs || lhs.get_data() == rhs.get_data();
10    }
11
12 private:
13     int data;
14     mutable std::mutex m;
15
16     int get_data() const {
17         std::lock_guard<std::mutex> lg(m);
18         return data;
19     }
20 };
21
22 int main() {
23     Integer i1(1);
24     Integer i2(2);
25     Integer i3(1);
26 }
```

```

27     std::cout << std::boolalpha << "i1 == i2? " << (i1 == i2) << std::endl;
28     std::cout << std::boolalpha << "i1 == i1? " << (i1 == i1) << std::endl;
29     std::cout << std::boolalpha << "i1 == i3? " << (i1 == i3) << std::endl;
30 }

```

### 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

```

1  #include <iostream>
2  #include <vector>
3  #include <memory>
4  #include <algorithm>
5
6  std::shared_ptr<std::vector<int>> p;
7
8  void square() {
9      if (!p) {
10         p.reset(new std::vector<int>(10, 42));
11     }
12
13     std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
14 }
15
16 std::ostream& operator<<(std::ostream& os, std::vector<int> v) {
17     os << "[";
18     for (auto p = v.begin(); p != v.end(); ++p) {
19         os << *p << (p == v.end() - 1 ? "" : ",");
20     }
21     os << "]";
22
23     return os;
24 }
25
26 int main() {

```

```

27     std::cout << "p is " << p << std::endl;
28     square();
29     std::cout << "p is " << p << std::endl;
30     std::cout << "*p is " << *p << std::endl;
31 }

```

**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**

```

1  #include <iostream>
2  #include <vector>
3  #include <memory>
4  #include <algorithm>
5  #include <mutex>
6
7  std::shared_ptr<std::vector<int>> p;
8  std::mutex p_m;
9
10 void square() {
11     std::unique_lock<std::mutex> lock(p_m);
12     if (!p) {
13         p.reset(new std::vector<int>(10, 42));
14     }
15     lock.unlock();
16
17     std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
18 }
19
20 std::ostream& operator<< (std::ostream& os, std::vector<int> v) {
21     os << "[";
22     for (auto p = v.begin(); p != v.end(); ++p) {
23         os << *p << (p == v.end() - 1 ? "" : ",");
24     }
25     os << "]";
26
27     return os;
28 }
29
30 int main() {
31     std::cout << "p is " << p << std::endl;
32     square();
33     std::cout << "p is " << p << std::endl;
34     std::cout << "*p is " << *p << std::endl;
35 }

```

**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**

```

1  #include <iostream>
2  #include <vector>
3  #include <memory>
4  #include <algorithm>
5  #include <thread>
6  #include <mutex>
7
8  std::shared_ptr<std::vector<int>>> p;
9  std::mutex p_m;
10
11 void square() {
12     if (!p) {
13         std::unique_lock<std::mutex> lock(p_m);
14         if (!p) {
15             p.reset(new std::vector<int>(100, 42));
16         }
17     }
18
19     std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
20 }
21
22 std::ostream& operator<<(std::ostream& os, std::vector<int> v) {
23     os << "[";
24     for (auto p = v.begin(); p != v.end(); ++p) {
25         os << *p << (p == v.end() - 1 ? "" : ",");
26     }
27     os << "]";
28
29     return os;
30 }
31
32 int main() {
33     std::cout << "p is " << p << std::endl;
34
35     std::thread t1(square);
36     std::thread t2(square);
37     t1.join();
38     t2.join();
39
40     std::cout << "p is " << p << std::endl;
41     std::cout << "*p is " << *p << std::endl;
42 }

```

**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`**

```

1  #include <iostream>
2  #include <vector>
3  #include <memory>
4  #include <algorithm>
5  #include <thread>
6  #include <mutex>
7
8  std::shared_ptr<std::vector<int>>> p;
9  std::once_flag p_flag;
10
11
12 void square() {
13     std::call_once(p_flag, [](){p.reset(new std::vector<int>(100, 42));});
14     std::for_each(p->begin(), p->end(), [](int& i) {i *= i;});
15 }
16
17 std::ostream& operator<< (std::ostream& os, std::vector<int> v) {
18     os << "[";
19     for (auto p = v.begin(); p != v.end(); ++p) {
20         os << *p << (p == v.end() - 1 ? "" : ",");
21     }
22     os << "]";
23
24     return os;
25 }
26
27 int main() {
28     std::cout << "p is " << p << std::endl;
29
30     std::thread t1(square);
31     std::thread t2(square);
32     t1.join();
33     t2.join();
34
35     std::cout << "p is " << p << std::endl;
36     std::cout << "*p is " << *p << std::endl;
37 }

```

`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**

```

1  #include "scoped_thread.h"
2  #include <iostream>
3  #include <vector>
4  #include <memory>

```

```

5 #include <algorithm>
6 #include <thread>
7 #include <mutex>
8
9 using namespace std;
10
11 class lunchline {
12 public:
13     lunchline(): entre("salisbury steak") {}
14
15     lunchline(const std::string& entre_): entre(entre_) {}
16
17     std::string front() {
18         std::call_once(flag, &lunchline::init, this);
19         return menu.front();
20     }
21
22     std::string back() {
23         std::call_once(flag, &lunchline::init, this);
24         return menu.back();
25     }
26
27 private:
28     std::vector<std::string> menu;
29     std::string entre;
30     std::once_flag flag;
31
32     void init() {
33         menu = std::vector<std::string>(100, entre);
34     }
35 };
36
37 int main() {
38     lunchline ll;
39     conc::scoped_thread t1{thread([&]() {cout << ll.front() << endl;}});
40     conc::scoped_thread t2{thread([&]() {cout << ll.back() << endl;}});
41 }

```

**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`**

```

1  #include <iostream>
2  #include <map>
3  #include <string>
4  #include <mutex>
5  #include <boost/thread/shared_mutex.hpp>
6
7  class dns_entry {
8  public:
9      dns_entry(int ip=-1) : ip(ip_) {}
10
11     friend bool operator==(const dns_entry& lhs, const dns_entry& rhs) {
12         return lhs.ip == rhs.ip;
13     }
14
15 private:
16     int ip;
17 };
18
19 class dns_cache {
20 public:
21     dns_entry find_entry(const std::string& domain) const {
22         // multiple threads will share this lock
23         boost::shared_lock<boost::shared_mutex> lk(entry_mutex);
24         const auto it = entries.find(domain);
25         return (it == entries.end()) ? dns_entry() : it->second;
26     }
27
28     void updated_or_add_entry(const std::string& domain, const dns_entry &dns) {
29         // only one thread will own this lock
30         std::lock_guard<boost::shared_mutex> lk(entry_mutex);
31         entries[domain] = dns;
32     }
33
34 private:
35     std::map<std::string, dns_entry> entries;
36     mutable boost::shared_mutex entry_mutex;
37 };
38
39
40 int main() {
41     dns_cache cache;
42 }

```



### 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

## 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