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**

```
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
```

```
    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**

```
#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**

```
#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**

```
#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**

```
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**

```
#ifndef THREAD_GUARD_H
#define THREAD_GUARD_H

```

```
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*.

```
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**

```
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**

```
#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));
```

```
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>
```

```

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**

```
#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**

```
#ifndef SCOPED_THREAD_H
#define SCOPED_THREAD_H

#include <thread>

```

```
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 contructor and
13  * throws and 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 owns 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 succesfully moves t_ into a local copy. NOTE that the thread
29      * <t_> is MOVED. If you isntantiate 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>
```

```
#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(),
```

```
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

# 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