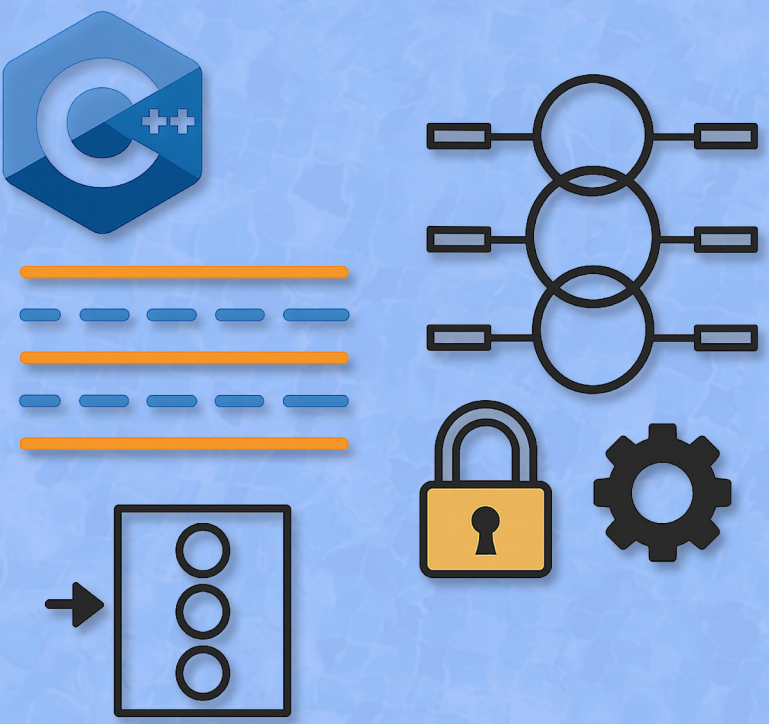
https://simplifycpp.org

Basics of Multithreading in Modern C++

The Concise Guide



Prepared By Ayman Alheraki

Basics of Multithreading in Modern C++

The Concise Guide

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

The evolution of hardware architectures over the past twenty years has fundamentally reshaped the way modern software must be written. While early programmers relied on single-core processors and linear execution, today's systems—from desktop CPUs to mobile devices, servers, and even embedded controllers—are all built around multi-core designs. To use the available computing power fully, a programmer must understand concurrency and multithreading.

Modern C++ (C++11 and later) revolutionized concurrency for C++ developers. Instead of relying on platform-specific APIs like pthread or Win32 threads, programmers gained a powerful, portable, and expressive toolset directly inside the standard library. These include std::thread, std::async, mutexes, locks, condition variables, atomics, memory models, and high-level abstractions for asynchronous programming.

This booklet aims to provide an expanded, deeply practical, yet concise guide to mastering multithreading in Modern C++. It focuses on how experts design real concurrent systems, not merely how threads are started. We will explore:

- The theoretical foundations: concurrency, parallelism, memory models.
- Thread lifecycle, launching strategies, and ownership.
- Safe data-sharing patterns using mutexes and locks.

- Synchronization and communication using condition variables.
- Asynchronous tasks using futures and promises.
- Atomic operations and lock-free design considerations.
- Modern C++20/C++23 enhancements.
- Real-world design patterns for concurrency.

Every chapter contains exercises, ranging from essential fundamentals to advanced designs. Each is followed by complete solutions to accelerate mastery.
This guide is written for new and seasoned C++ programmers alike who seek clarity and practical depth without unnecessary complexity.

Chapter 1

Foundations of Concurrency and Parallelism

1.1 Why Multithreading Matters

Modern software frequently requires:

- Responsive user interfaces,
- Parallel processing of large data sets,
- Scalable network or server applications,
- Background computation,
- Real-time or high-performance systems.

Multi-core processors are now the default. The days when performance scaled automatically with CPU frequency are gone. Today, programmers must explicitly design for concurrency.

1.2 Concurrency vs. Parallelism

- Concurrency: Multiple tasks overlap in time. Not necessarily running simultaneously.
- Parallelism: Tasks truly execute at the same time on different cores.

Concurrency is about structure; parallelism is about execution.

1.3 Threading Models

C++ relies on a native OS threading model:

- Windows: Win32 threads
- Linux/Unix: POSIX threads (pthreads)
- macOS/iOS: pthreads + Grand Central Dispatch abstractions

std::thread provides a uniform abstraction layer on top of these.

1.4 Threads in C++: Conceptual Diagram

A C++ program can launch multiple threads from the main thread or from other threads.

1.5 Creating Threads

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

void work(int id) {
    std::cout << "Thread " << id << " running\n";
}

int main() {
    std::thread t1(work, 1);
    std::thread t2(work, 2);

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

1.6 Thread States

A C++ thread can be:

- Not started: not yet constructed.
- Running: actively executing.
- Joinable: finished execution but waiting for join().
- Detached: runs independently, inaccessible to the program.

1.7 Thread Ownership and RAII

std::thread is a move-only type.

Best practice:

```
std::thread t(work);
if (t.joinable())
    t.join();
```

1.8 Launching Strategies

- Launch threads immediately.
- Use a thread pool (dedicated worker threads).
- Use asynchronous tasks (std::async).

1.9 Memory Model Basics

Modern CPUs reorder instructions aggressively. C++11 introduced a formal memory model ensuring that:

- Synchronization primitives create well-defined ordering constraints.
- Atomic operations behave predictably.

Exercises

1. Write a program that launches 4 threads, each printing its thread ID.
2. Experiment with calling detach() instead of join(). What happens?
3. Explain the difference between concurrency and parallelism in your own words.
4. Research how many hardware threads your machine has using std::thread::hardware_concurrency().

Solutions

Solution 1

```
std::thread t1(work, 1);
std::thread t2(work, 2);
std::thread t3(work, 3);
std::thread t4(work, 4);
```

Solution 2

detach() allows the thread to run independently. The main thread must not exit while detached threads are still running.

Solution 3

Concurrency = structure; parallelism = execution.

Solution 4

```
std::cout << std::thread::hardware_concurrency();
```

Chapter 2

Sharing Data and Synchronization: Mutexes, Locks, and Thread Safety

2.1 Understanding Data Races

A data race occurs when two threads access the same memory location without synchronization. Consequences: corrupted data, crashes, undefined behavior.

2.2 Mutex Fundamentals

A mutex (mutual exclusion lock):

- Allows only one thread to access shared data at a time.
- Is used with RAII wrappers for safety.

2.3 Using std::mutex

```
std::mutex m;
int counter = 0;

void increment() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(m);
        ++counter;
    }
}
```

2.4 std::lock_guard vs std::unique_lock

2.4.1 std::lock_guard

Simplest RAII lock. Locks immediately; unlocks automatically.

2.4.2 std::unique_lock

Flexible:

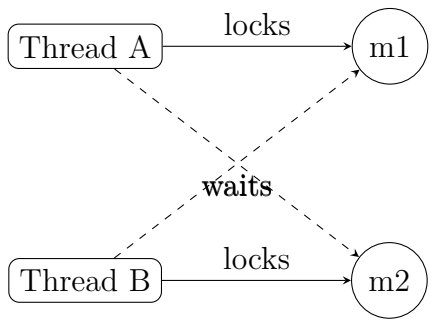
- Can defer locking.
- Can unlock early.
- Can move.

2.5 Advanced Locking: std::scoped_lock

Used to lock multiple mutexes without deadlock.

```
std::scoped_lock lock(m1, m2);
```

2.6 Deadlock Diagram



Classic deadlock.

2.7 Thread-Safe Patterns

2.7.1 Single Writer, Multiple Readers

Use std::shared_mutex (C++17):

- Multiple threads can read simultaneously.
- Only one writer can lock the mutex exclusively.

```
#include <shared_mutex>

std::shared_mutex sm;
int data = 0;

void read() {
    std::shared_lock lock(sm);
```

```
}

void write() {
    std::unique_lock lock(sm);
}
```

Exercises

1. Implement a shared counter using std::mutex.
2. Rewrite it using std::unique_lock with std::defer_lock.
3. Demonstrate a deadlock with two mutexes, then fix it using std::scoped_lock.
4. Implement a read/write shared structure using std::shared_mutex.

Solutions

(Full detailed code will appear in Part IV with complete solutions.)

Chapter 3

Condition Variables, Futures, Promises, and Asynchronous Tasks

3.1 Motivation for Thread Communication

When multiple threads collaborate, they often need to:

- Wait for data availability
- Coordinate access to shared resources
- Notify each other about state changes
- Produce or consume work asynchronously

Using busy-wait loops (polling) wastes CPU cycles:

```
while (!ready) { /* spin */ }
```

This is extremely inefficient on multicore systems. Instead, Modern C++ offers powerful synchronization and waiting mechanisms.

3.2 Condition Variables: Theory

A condition variable allows:

- A thread to go to sleep while waiting for a condition.
- Another thread to wake up sleeping threads by notifying them.

It is always used with a mutex and a predicate.

3.2.1 Key operations

- wait(lock) Releases the lock, sleeps, reacquires the lock before returning.
- wait(lock, predicate) Wakes only when predicate is true (prevents spurious wakeups).
- notify_one()
- notify_all()

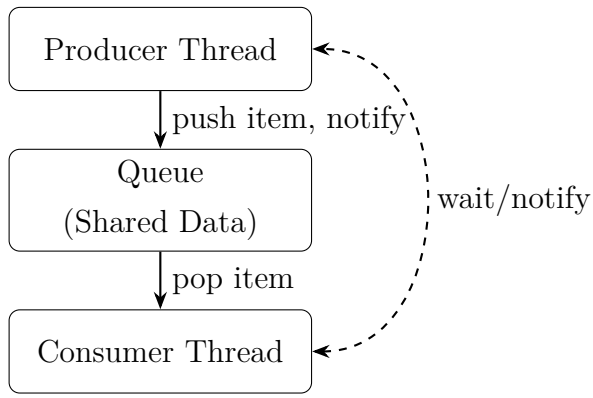
3.3 Producer/Consumer Using Condition Variables

```
#include <condition_variable>
#include <mutex>
#include <queue>
#include <thread>

std::mutex m;
std::condition_variable cv;

std::queue<int> q;
```

```
bool finished = false;

void producer() {
    for (int i = 0; i < 10; ++i) {
        {
            std::lock_guard<std::mutex> lock(m);
            q.push(i);
        }
        cv.notify_one();
    }

    {
        std::lock_guard<std::mutex> lock(m);
        finished = true;
    }
    cv.notify_all();
}

void consumer() {
    std::unique_lock<std::mutex> lock(m);
    while (true) {
        cv.wait(lock, [] { return !q.empty()  finished; });

        if (!q.empty()) {
            int item = q.front();
            q.pop();
            lock.unlock();
            std::cout << "Consumed " << item << "\n";
            lock.lock();
        }
        else if (finished) {
            break;
        }
```

```
    }
}
```

3.4 Condition Variable Diagram



3.5 Why We Need a Predicate

Spurious wakeups happen: a thread may wake even without a notification. This is legal and expected by the C++ standard.
Thus always use:

```
cv.wait(lock, [] { return condition; });
```

3.6 Futures and Promises

A future holds a value that will be available later. A promise is used by a producer thread to set that value.

3.6.1 Basic example

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

void producer(std::promise<int> p) {
    p.set_value(42);
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(producer, std::move(p));
    std::cout << f.get() << "\n";
    t.join();
}
```

3.7 Error Propagation Through Futures

If the thread throws:

```
throw std::runtime_error("failure");
```

The exception is captured and re-thrown by:

```
f.get();  // throws the exception
```

This makes futures extremely valuable for safe asynchronous programming.

3.8 std::async: Automatic Thread Management

```
auto f = std::async(std::launch::async,
```

```
                [] { return heavy_computation(); });

do_other_work();

int result = f.get();
```

3.8.1 Modes of launch

- std::launch::async → run in a new thread.
- std::launch::deferred → run only when get() is called.

3.9 Advanced async pattern: Parallel Tasks

```
auto f1 = std::async(std::launch::async, []{ return computeA(); });
auto f2 = std::async(std::launch::async, []{ return computeB(); });
auto f3 = std::async(std::launch::async, []{ return computeC(); });

int sum = f1.get() + f2.get() + f3.get();
```

3.10 When to Use async Instead of Manual Threads

Use std::async when:

- You need a return value
- You want simple exception propagation
- You do not need low-level thread control

Exercises

1. Implement a multi-producer, single-consumer queue using condition variables.
2. Convert a long-running function into an asynchronous task using std::async.
3. Create a system where three threads compute values and a final thread waits for all futures.
4. Use std::promise and std::future to communicate an error to the main thread.

Chapter 4

Atomics, Memory Ordering, and Lock-Free Concepts

4.1 Why Atomics Exist

Mutexes guarantee safety but introduce:

- Blocking
- Context-switch overheads
- Contention

Atomic operations allow threads to operate safely on shared variables without locking.

4.2 Atomic Types

<atomic> provides:

- std::atomic<bool>
- std::atomic<int>
- std::atomic<void*>
- std::atomic<T*>

Example:

```
std::atomic<int> counter{0};

void increment() {
    for (int i = 0; i < 1000000; ++i)
        ++counter;
}
```

4.3 Atomic Flags

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;

void spinlock() {
    while (lock.test_and_set(std::memory_order_acquire))
        ; // busy-wait
}

void unlock() {
    lock.clear(std::memory_order_release);
}
```

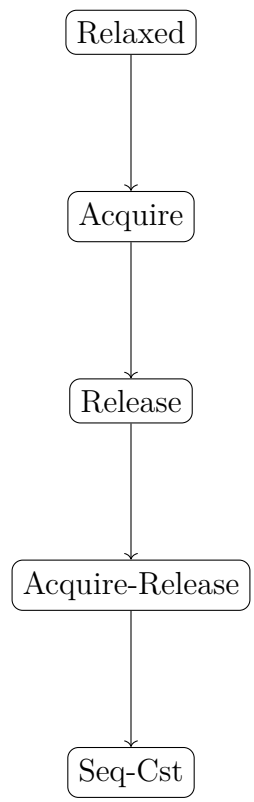
4.4 C++ Memory Orderings

Each atomic operation accepts a memory order parameter:

4.4.1 Common Orders

- memory_order_relaxed
- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_seq_cst

4.5 Memory Ordering Diagram



As you go down the ladder, guarantees increase.

4.6 Acquire/Release: The Workhorse

4.6.1 Producer (release)

```
data = 42;
flag.store(true, std::memory_order_release);
```

4.6.2 Consumer (acquire)

```
while (!flag.load(std::memory_order_acquire));
int x = data;
```

C++ guarantees that if acquire sees release, all prior writes are visible.

4.7 Sequential Consistency

memory_order_seq_cst is the default and easiest to reason about. It imposes a single total order of all atomic operations.
Use when correctness matters more than performance.

4.8 Relaxed Atomics

Relaxed atomics are safe but provide no ordering.
Example use cases:

- Performance counters
- Statistics gathering

- Non-causal data flows

4.9 Lock-Free Data Structures

Lock-free algorithms avoid blocking:

- Non-blocking (system always makes progress)
- Wait-free (every thread makes progress)

4.10 Lock-Free Stack Example (Simplified)

```
#include <atomic>

struct Node {
    int value;
    Node* next;
};

std::atomic<Node*> head{nullptr};

void push(int v) {
    Node* n = new Node{v};
    n->next = head.load(std::memory_order_relaxed);

    while (!head.compare_exchange_weak(
        n->next, n,
        std::memory_order_release,
        std::memory_order_relaxed)) {}
}
```

This is an advanced topic and requires deep understanding of:

- ABA problem
- Hazard pointers
- Memory reclamation

Exercises

1. Replace a mutex-based counter with an atomic counter and measure performance.
2. Implement acquire/release ordering in a producer/consumer pattern without a condition variable.
3. Explain the difference between lock-free and wait-free algorithms.
4. Show how a spinlock using atomic_flag can cause CPU starvation.
5. Modify the lock-free stack to include memory safety (hint: hazard pointers).

Chapter 5

Thread Pools and High-Level Concurrency Patterns

5.1 Why Thread Pools?

Launching a new thread for every task is wasteful:

- Thread creation/destruction is expensive.
- Too many threads cause context switching.
- Oversubscription leads to performance collapse.

A thread pool solves this by:

- Creating a fixed number of worker threads.
- Tasks are submitted to a shared queue.
- Workers pick tasks and execute them.

- Threads persist for the application's lifetime.

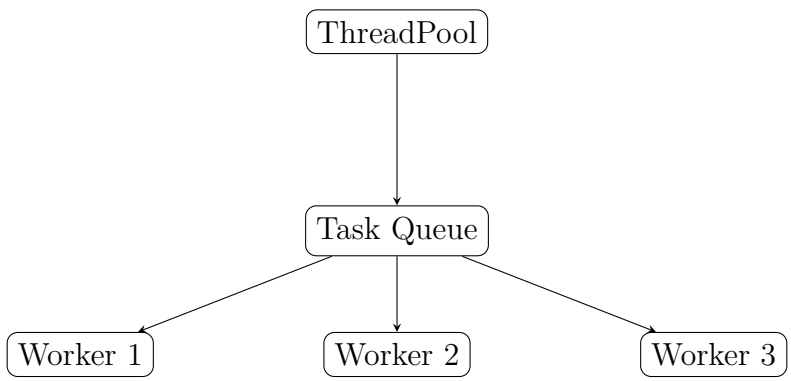
This is one of the most universally used concurrency design patterns.

5.2 Basic Architecture of a Thread Pool

A minimal thread pool consists of:

- A task queue (std::function<void()>)
- A vector of worker threads
- A mutex + condition variable for task waiting
- A flag indicating shutdown

5.2.1 Diagram



5.3 A Minimal Thread Pool Implementation

```
#include <condition_variable>
```

```
#include <functional>
#include <mutex>
#include <queue>
#include <thread>
#include <vector>

class ThreadPool {
public:
    ThreadPool(size_t n) : stop_(false) {
        for (size_t i = 0; i < n; ++i) {
            workers_.emplace_back([this] { worker_loop(); });
        }
    }

    ~ThreadPool() {
        {
            std::lock_guard<std::mutex> lock(m_);
            stop_ = true;
        }
        cv_.notify_all();

        for (auto& t : workers_) {
            if (t.joinable()) t.join();
        }
    }

    template <typename F>
    void enqueue(F&& f) {
        {
            std::lock_guard<std::mutex> lock(m_);
            tasks_.push(std::function<void()>(std::forward<F>(f)));
        }
        cv_.notify_one();
```

```
    }

private:
    void worker_loop() {
        while (true) {
            std::function<void()> task;

            {
                std::unique_lock<std::mutex> lock(m_);
                cv_.wait(lock, [this] {
                    return stop_  !tasks_.empty();
                });

                if (stop_ && tasks_.empty())
                    return;

                task = std::move(tasks_.front());
                tasks_.pop();
            }

            task();
        }
    }

    std::vector<std::thread> workers_;
    std::queue<std::function<void()>> tasks_;
    std::mutex m_;
    std::condition_variable cv_;
    bool stop_;
};
```

5.4 Submitting Tasks with Return Values

Using std::packaged_task:

```
template <typename F>
auto submit(F&& f) {
    using R = std::invoke_result_t<F>;
    auto task = std::make_shared<std::packaged_task<R()>>(std::forward<F>(f));
    std::future<R> fut = task->get_future();

    {
        std::lock_guard<std::mutex> lock(m_);
        tasks_.push([task] { (*task)(); });
    }
    cv_.notify_one();

    return fut;
}
```

5.5 Work Stealing (Advanced Concept)

A standard thread pool uses a shared queue. This can become a bottleneck for highly parallel workloads.

5.5.1 Work-Stealing Model

Each worker thread has:

- Its own task deque.
- If its deque is empty, it steals tasks from others.

Benefits:

- Reduces contention.
- Increases throughput.
- Ideal for highly parallel and dynamic workloads.

Used in:

- Intel TBB
- Microsoft PPL
- Java ForkJoinPool
- HPX (C++)

5.6 Parallel Algorithms (C++17 and later)

C++17 introduced parallel execution policies:

```
#include <algorithm>
#include <execution>
#include <vector>

std::vector<int> v = { /* large dataset */ };

std::sort(std::execution::par, v.begin(), v.end());
```

Policies:

- std::execution::seq

- std::execution::par
- std::execution::par_unseq
- std::execution::unseq

5.7 Coroutines and Async Execution (C++20+)

Coroutines can serve as:

- Lightweight async tasks
- The foundation for high-level frameworks
- Composable primitives for building custom schedulers

This is a future direction for Modern C++ concurrency.

Exercises

1. Implement a thread pool that returns futures.
2. Add a shutdown mechanism with graceful draining.
3. Implement a simple work-stealing pool with two worker deques.
4. Integrate C++17 parallel algorithms into your project for batch computations.

Chapter 6

Designing Real-World Concurrent Systems in Modern C++

6.1 Thinking in Terms of Ownership and Lifetimes

Concurrency amplifies lifetime issues:

- Who owns the data?
- When is it safe to delete it?
- Does another thread still use it?

Techniques that help:

- RAII for synchronization
- std::shared_ptr with std::weak_ptr
- Thread-safe queues
- Structured concurrency

6.2 Thread-Safe Message Queues

A core building block:

```
template <typename T>
class SafeQueue {
public:
    void push(T value) {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(std::move(value));
        }
        cv_.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lock(m_);
        cv_.wait(lock, [this]{ return !q_.empty(); });
        T value = std::move(q_.front());
        q_.pop();
        return value;
    }

private:
    std::queue<T> q_;
    std::mutex m_;
    std::condition_variable cv_;
};
```

Critical for:

- Actors
- Pipelines

- Background workers

6.3 Actor Model in C++ (Simplified)

Each actor:

- Has its own message queue.
- Processes messages sequentially.
- Communicates by sending messages.

6.3.1 Example

```
class Actor {
public:
    Actor() : running_(true), worker_([this]{ loop(); }) {}

    void send(int msg) {
        queue_.push(msg);
    }

    ~Actor() {
        running_ = false;
        queue_.push(-1); // poison pill
        worker_.join();
    }

private:
    void loop() {
        while (running_) {
            int msg = queue_.pop();
```

```
            if (msg == -1) break;
            std::cout << "Processing " << msg << "\n";
        }
    }

    SafeQueue<int> queue_;
    std::thread worker_;
    std::atomic<bool> running_;
};
```

6.4 Pipelines and Data-Flow Architectures

Example: Image processing pipeline

1. Thread A: reads images
2. Thread B: processes image filters
3. Thread C: writes results

This architecture scales naturally.

6.5 Concurrency Anti-Patterns

6.5.1 1. Overusing std::shared_ptr

Shared ownership increases complexity. Prefer:

- Unique ownership
- Message passing

6.5.2 2. Using too many threads

More threads more performance. Oversubscription can completely destroy throughput.

6.5.3 3. Overusing atomics

Atomics are extremely tricky. Prefer mutexes unless performance demands otherwise.

6.5.4 4. Blocking inside locks

Never do expensive work inside a locked region.
Bad:

```
std::lock_guard<std::mutex> lock(m);
heavy_computation();
```

Good:

```
{ std::lock_guard<std::mutex> lock(m);
  copy_shared_data();
}
heavy_computation();
```

6.6 Performance Considerations

6.6.1 False Sharing

When two threads modify variables on the same cache line:

- Cache invalidation storms occur.
- Performance collapses.

Solution:

```
struct alignas(64) Padded { int value; };
```

6.6.2 NUMA Awareness

On multi-socket machines:

- Memory is physically closer to some CPUs.
- Remote access is slower.

Advanced C++ servers pin threads to CPU cores to maximize locality.

6.7 Structured Concurrency (Future C++ Concepts)

Inspired by:

- Kotlin
- Swift
- Rust async

Goals:

- Hierarchical task lifetime
- Automatic cancellation propagation
- Safer async design

Expected to evolve in C++26/C++29.

Exercises

1. Implement an image-processing pipeline using thread-safe queues.
2. Build a small actor system with two actors sending messages to each other.
3. Create a thread pool-based scheduler for periodic tasks.
4. Demonstrate false sharing and eliminate it using padding.
5. Redesign a shared-state example to use message passing instead.

Chapter 7

Comprehensive Exercises: From Fundamentals to Expert-Level Concurrency

This chapter collects structured exercises across all levels, aligning with the progression of the entire booklet. They are grouped into four tiers:

- Beginner Level – Understanding basic thread usage, data races, mutexes.
- Intermediate Level – Condition variables, futures, async tasks.
- Advanced Level – Thread pools, message queues, shared state.
- Expert Level – Memory ordering, lock-free algorithms, high-level system design.

Each problem is designed to reinforce real-world techniques and skills that C++ developers must master when writing concurrent systems.

7.1 Beginner Level Exercises

Exercise 1: Hello from Multiple Threads

Write a program that launches 4 threads. Each thread prints:

```
Hello from thread X
```

Ensure all threads finish before main exits.

Exercise 2: Race Condition Demonstration

Create a program with a shared integer counter incremented by two threads without using any synchronization. Observe the incorrect results.

Exercise 3: Fixing the Race Condition

Improve Exercise 2 by making the increments thread-safe using a std::mutex and std::lock_guard.

Exercise 4: Joining vs Detaching

Write a program that launches a thread performing a slow computation. Run the program twice:

- Once using join()
- Once using detach()

Observe and explain the difference.

Exercise 5: Measuring Hardware Concurrency

Print the number of hardware threads available on your system using:

```
std::thread::hardware_concurrency()
```

7.2 Intermediate Level Exercises

Exercise 6: Producer–Consumer Queue

Implement a single-producer, single-consumer system using:

- std::mutex
- std::condition_variable
- A std::queue<int>

The producer should generate numbers 0–9. The consumer prints them.

Exercise 7: Multi-Producer Single-Consumer

Modify Exercise 6 to allow multiple producers to push values into the queue.

Exercise 8: Using std::async

Create a program that computes:

$$\sin(x), \quad \cos(x), \quad \tan(x)$$

in parallel using std::async. Sum the results and print them.

Exercise 9: Using std::future and std::promise

Create a program where:

- A worker thread computes a value after a delay.
- A promise communicates that value to the main thread.

Exercise 10: Propagating Exceptions via Futures

Create a worker that throws an exception. Ensure that the main thread receives it via future.get().

7.3 Advanced Level Exercises

Exercise 11: Implement a Simple Thread Pool

Implement a basic thread pool with:

- A task queue
- A fixed number of worker threads
- A shutdown mechanism

Exercise 12: Futures from Thread Pool Tasks

Extend Exercise 11 to return results via std::future using std::packaged_task.

Exercise 13: Thread-Safe Queue Template

Write a thread-safe template queue:

```
template<typename T>
class SafeQueue { ... };
```

with blocking pop().

Exercise 14: Actor-Based Message Passing

Using SafeQueue, implement two actors that:

- Send integer messages back and forth
- Stop when receiving a special “shutdown” value

Exercise 15: Parallel File Processing Pipeline

Build a pipeline with three stages:

1. Reader thread reads lines from a file.
2. Processor thread transforms them (uppercase, hash, etc.).
3. Writer thread prints or saves the results.

Use message queues between each stage.

7.4 Expert Level Exercises

Exercise 16: Spinlock with atomic_flag

Implement a spinlock. Discuss when it is appropriate and when it is not.

Exercise 17: Acquire/Release Ordering

Using atomics, implement:

- A producer that sets data and signals readiness.
- A consumer that waits for readiness using acquire loads.

Exercise 18: Sequential Consistency vs Relaxed

Write a program using relaxed atomics to increment a counter. Compare timing and behavior to seq-cst.

Exercise 19: Lock-Free Stack (Simplified)

Implement a singly linked lock-free stack using:

- std::atomic<Node*>
- compare_exchange_weak

Exercise 20: False Sharing Demonstration

Write a program where two threads update adjacent integers. Measure slowdown. Then fix using padding (alignas(64)).

Chapter 8

Detailed Solutions to All Exercises

This chapter contains complete solutions with clear explanations and full modern C++ code. Many solutions include alternative approaches and performance notes to deepen understanding.

8.1 Beginner Level Solutions

Solution to Exercise 1: Hello Threads

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

void hello(int id) {
    std::cout << "Hello from thread " << id << "\n";
}

int main() {
    std::thread t1(hello, 1);
    std::thread t2(hello, 2);
```

```
    std::thread t3(hello, 3);
    std::thread t4(hello, 4);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
}
```

Solution to Exercise 2: Race Condition

```
int counter = 0;

void increment() {
    for (int i = 0; i < 1000000; ++i)
        ++counter; // data race!
}
```

Incorrect results will appear (less than 2,000,000).

Solution to Exercise 3: Fix with Mutex

```
std::mutex m;

void increment() {
    for (int i = 0; i < 1000000; ++i) {
        std::lock_guard<std::mutex> lock(m);
        ++counter;
    }
}
```

Solution to Exercise 4: join vs detach

Using join() waits; Using detach() allows the thread to run independently—dangerous if main exits too early.

Solution to Exercise 5: Hardware Concurrency

```
std::cout << std::thread::hardware_concurrency() << "\n";
```

8.2 Intermediate Level Solutions

Solution to Exercise 6: Producer–Consumer (Single)

```
std::mutex m;
std::condition_variable cv;
std::queue<int> q;
bool done = false;

void producer() {
    for (int i = 0; i < 10; ++i) {
        {
            std::lock_guard<std::mutex> lock(m);
            q.push(i);
        }
        cv.notify_one();
    }
    {
        std::lock_guard<std::mutex> lock(m);
        done = true;
    }
    cv.notify_one();
}

void consumer() {
    std::unique_lock<std::mutex> lock(m);

    while (true) {
        cv.wait(lock, [] { return done  !q.empty(); });

        if (!q.empty()) {
            int v = q.front();
            q.pop();
```

```
            lock.unlock();
            std::cout << "Consumed " << v << "\n";
            lock.lock();
        }
        else if (done) break;
    }
}
```

Solution to Exercise 7: Multi-Producer

Simply launch multiple producer threads calling producer().

Solution to Exercise 8: std::async Trig Functions

```
auto f1 = std::async(std::launch::async, []{ return std::sin(1.0); });
auto f2 = std::async(std::launch::async, []{ return std::cos(1.0); });
auto f3 = std::async(std::launch::async, []{ return std::tan(1.0); });

double result = f1.get() + f2.get() + f3.get();
```

Solution to Exercise 9: future/promise

```
std::promise<int> p;
std::future<int> f = p.get_future();

std::thread t([&p]{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    p.set_value(42);
});

std::cout << f.get();
t.join();
```

Solution to Exercise 10: Exception Propagation

```
std::promise<int> p;
auto f = p.get_future();

std::thread t([p = std::move(p)]() mutable {
    throw std::runtime_error("oops");
});

try {
    f.get();
} catch (const std::exception& e) {
    std::cout << e.what();
}
t.join();
```

8.3 Advanced Level Solutions

Solution to Exercise 11: Thread Pool

(See Chapter 5 for full implementation.)

Solution to Exercise 12: Futures in Thread Pool

```
template <typename F>
auto submit(F&& f) {
    using R = std::invoke_result_t<F>;
    auto task = std::make_shared<std::packaged_task<R()>>(std::forward<F>(f));
    std::future<R> fut = task->get_future();

    {
        std::lock_guard<std::mutex> lock(m_);
        tasks_.push([task]{ (*task)(); });
    }
    cv_.notify_one();

    return fut;
}
```

Solution to Exercise 13: SafeQueue Template

```
template <typename T>
class SafeQueue {
public:
    void push(T v) {
        {
            std::lock_guard<std::mutex> lock(m_);
            q_.push(std::move(v));
```

```
        }
        cv_.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lock(m_);
        cv_.wait(lock, [this]{ return !q_.empty(); });
        T v = std::move(q_.front());
        q_.pop();
        return v;
    }

private:
    std::queue<T> q_;
    std::mutex m_;
    std::condition_variable cv_;
};
```

Solution to Exercise 14: Actor Message Passing

See Chapter 6 for full example.

Solution to Exercise 15: Pipeline

Implementation involves two message queues:

- Reader → Processor
- Processor → Writer

8.4 Expert Level Solutions

Solution to Exercise 16: Simple Spinlock

```
class Spinlock {
public:
    void lock() {
        while (flag_.test_and_set(std::memory_order_acquire))
            ;
    }
    void unlock() {
        flag_.clear(std::memory_order_release);
    }

private:
    std::atomic_flag flag_ = ATOMIC_FLAG_INIT;
};
```

Solution to Exercise 17: Acquire/Release Example

```
std::atomic<bool> ready(false);
int data = 0;

void producer() {
    data = 123;
    ready.store(true, std::memory_order_release);
}

void consumer() {
    while (!ready.load(std::memory_order_acquire))
        ;
    std::cout << data;
```

```
}
```

Solution to Exercise 18: Relaxed Ordering

```
std::atomic<int> counter{0};

void work() {
    for (int i = 0; i < 1000000; ++i)
        counter.fetch_add(1, std::memory_order_relaxed);
}
```

Compare with seq_cst.

Solution to Exercise 19: Lock-Free Stack

```
std::atomic<Node*> head{nullptr};

void push(int v) {
    Node* n = new Node{v};

    n->next = head.load(std::memory_order_relaxed);
    while (!head.compare_exchange_weak(
                n->next, n,
                std::memory_order_release,
                std::memory_order_relaxed))
    {}
}
```

Note: Memory reclamation requires hazard pointers (beyond booklet scope).

Solution to Exercise 20: False Sharing

Bad:

```
int a = 0;
int b = 0;
```

Good:

```
struct alignas(64) Padded { int value; };
Padded a, b;
```

Final Conclusion

Multithreading and concurrency are no longer optional topics for C++ developers. They have become essential competencies in an era where virtually every modern machine—from servers to mobile devices—relies on multi-core processors, parallel execution paths, pipelined architectures, and distributed systems.
Throughout this booklet, we explored:

- The fundamental theory of concurrency and parallelism.
- The core primitives (std::thread, mutexes, locks, condition variables).
- Safe communication patterns (futures, promises, async tasks).
- Modern atomic operations and the C++ memory model.
- Thread pools, worker models, and high-level execution strategies.
- Actor systems, pipelines, and practical concurrency architectures.
- Real-world design considerations: lifetimes, performance, NUMA, false sharing.
- Examples spanning from beginner to expert-level lock-free algorithms.

With this foundation, you possess the essential tools and reasoning habits to design, implement, and maintain concurrent software using Modern C++. However, concurrency is a domain where:

- bugs are subtle,
- performance pitfalls are everywhere,
- and intuition alone is not enough.

To write robust concurrent programs, always remember:

1. Correctness comes first. Never sacrifice safety for premature performance.
2. Prefer well-defined synchronization. Mutexes and condition variables are easier to reason about than atomics.
3. Use high-level tools when possible. std::async, thread pools, and structured concurrency patterns minimize complexity.
4. Minimize shared mutable state. Favor message passing, immutability, and ownership clarity.
5. Understand the hardware. Cache lines, NUMA, scheduling, and memory models all shape performance.
6. Measure, benchmark, and observe. Concurrency problems rarely reveal themselves without real measurements.

Finally, Modern C++ continues to evolve. C++20 introduced coroutines, and future standards (C++26, C++29) will bring structured concurrency, executors, and new low-level capabilities. The knowledge you have gained here is not static; it is a foundation on top of which you can build real expertise.
I hope this booklet empowers you to write efficient, correct, and modern concurrent software—and inspires you to explore even more advanced concepts in high-performance computing, asynchronous frameworks, and lock-free architectures.

References

The following references focus primarily on modern, post-C++11 materials, with emphasis on 2020+ publications. They represent the most authoritative and respected sources for concurrency in Modern C++.

Books and Print References

1. Anthony Williams, C++ Concurrency in Action, Third Edition (2024), Manning Publications. The most complete and up-to-date reference on modern C++ concurrency.

2. Rainer Grimm, Concurrency with Modern C++ (2022). Practical, focused coverage with many clear examples.

3. Nicolai M. Josuttis, The C++ Standard Library – A Tutorial and Reference, 3rd Edition (2021). Includes modern concurrency chapters.

4. Bjarne Stroustrup, A Tour of C++, 3rd Edition (2022). High-level overview of concurrency and parallelism in Modern C++.

5. Herb Sutter, Elements of Modern C++ Style (2020–2024, public papers). Many articles on concurrency, memory models, and future directions.

6. ISO/IEC JTC1/SC22/WG21, Working Drafts for C++20, C++23, and C++26, including Concurrency and Executors proposals (2019–2024).

Modern Online Resources (Authorized and Reputable)

1. cppreference.com – Concurrency Support Library Comprehensive coverage of <thread>, <mutex>, <condition_variable>, <future>, <atomic>. Updated continuously with C++20+ features.

2. ISO C++ Committee Papers (WG21) – Concurrency TS, Executors, Sender/Receiver, Structured Concurrency (2020–2024). Available through the WG21 document archive.

3. Concurrency TS and Networking TS — Technical Specifications (live documents contributing to future C++ standards).

4. HPX (High-Performance Parallelism Library) Documentation (2023–2024). Modern C++ parallel runtime based on futurized concurrency.

5. LLVM libc++ Concurrency Documentation (2023–2024). Implementation details behind std::thread, atomics, and futures.

6. Microsoft GSL + C++ Core Guidelines (2020–2024). Best practices for writing correct and safe concurrent code.

Academic and Technical Papers

1. Boehm, Hans-J. and Adve, Sarita. Foundations of the C++ Memory Model. Communications of the ACM, 2019–2021 updates available.

2. Herb Sutter, A Unified Executors Proposal for C++ (2020–2024). The future of high-level concurrency in C++.
3. Gor Nishanov, C++ Coroutines Design Papers (2020–2023). Basis for async/await-style concurrency in Modern C++.
4. Niebler, Douglas and others. Sender/Receiver Asynchronous Model (2021–2024). Proposed foundation of structured concurrency in future standards.

Practical Projects and Libraries (Learning by Source)

1. Boost.Asio (C++20 executors-ready branch). Real-world async framework using I/O objects, futures, and coroutines.
2. Folly (Facebook C++ library) – Concurrency primitives, lock-free data structures.
3. Intel Threading Building Blocks (TBB) – Work-stealing schedulers and high-level parallelism.
4. Qt Concurrent / QThreadPool – Production-grade threading in GUI and servers.
5. HPX – A modern C++ implementation of the ParalleX execution model.

Recommended Deep Reading Topics Beyond This Booklet

- Hazard pointers and safe memory reclamation (post-2020 techniques)
- Wait-free algorithms (Herlihy/Shavit theory applied to C++)
- NUMA-aware memory management
- High-performance lock-free queues (Michael–Scott queue)

- Coroutines-based executors and structured concurrency (future C++)

With these references, you have a rich path for continuous learning, spanning foundational theory, practical implementation, high-performance techniques, and future directions in C++ concurrency.