The C++ Master Companion — Syntax, Insight & Practice

ZephyrAmmor

October 2025

Contents

Module 7: Concurrency and Parallelism

Concurrency is where modern C++ meets real-world performance. When multiple tasks run simultaneously or asynchronously, efficiency soars—but so do risks like race conditions and deadlocks. Modern C++ (since C++11) offers a rich, standardized multithreading API that makes concurrent programming both powerful and portable.

□ Concept Overview

- Concurrency: Multiple tasks make progress at the same time (not necessarily simultaneously).
- Parallelism: Tasks execute *simultaneously* on multiple cores.
- Goal: Utilize hardware efficiently while maintaining correctness, safety, and clarity.

□ Thread Management

Syntax Block

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

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

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

```
    t1.join(); // Wait for t1 to finish
    t2.join();
}
```

Pitfalls / Notes / Insights

Pitfall: Failing to call `join()` or `detach()` before program exit causes termination.

Insight: Threads start immediately upon creation; they are not paused waiting for `join()`.

Under the Hood: `std::thread` wraps a native OS thread handle; `join()` synchronizes and releases the thread resource.

Mutexes and Locks

Used to synchronize access to shared resources.

Syntax Block

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

std::mutex mtx;
int counter = 0;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx); // RAII lock
        ++counter;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();

    std::cout << "Final counter: " << counter << '\n';
}
```

Pitfalls / Notes / Insights

Pitfall: Forgetting to lock leads to race conditions.

Insight: `std::lock_guard` ensures the mutex unlocks automatically (RAII principle).

Under the Hood: Mutexes often map to low-level kernel primitives; contention can cause context switches.

Condition Variables

Used to notify threads of state changes.

Example

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

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

void producer() {
    for (int i = 0; i < 5; ++i) {
        std::unique_lock<std::mutex> lock(mtx);
        q.push(i);
        cv.notify_one();
    }
    {
        std::unique_lock<std::mutex> lock(mtx);
        done = true;
        cv.notify_all();
    }
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [] { return !q.empty() || done; });
        if (!q.empty()) {
            std::cout << "Consumed: " << q.front() << '\n';
            q.pop();
        } else if (done) break;
    }
}

int main() {
    std::thread p(producer);
    std::thread c(consumer);
    p.join();
    c.join();
}
```

 Insight: Always pair `wait()` with a predicate to avoid spurious wakeups.

Futures, Promises, and Async

High-level concurrency abstractions that simplify thread management.

Example

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

int compute_square(int x) {
    return x * x;
}

int main() {
    std::future<int> result = std::async(std::launch::async, compute_square, 10);
    std::cout << "Result: " << result.get() << '\n';
}
```

 Insight: `std::async` automatically handles thread creation and synchronization.

 Under the Hood: Futures store state in a shared object; `get()` blocks until the value is ready.

□ Atomics and Memory Ordering

For lightweight synchronization of simple data types.

Example

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

std::atomic<int> counter(0);

void increment() {
    for (int i = 0; i < 100000; ++i)
        counter.fetch_add(1, std::memory_order_relaxed);
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << counter.load() << '\n';
}
```

□ Insight: Atomics prevent data races *without* explicit locks.

□ Pitfall: Use `memory_order_relaxed` only when order of operations doesn't matter.

□ Parallel Algorithms (C++17)

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

int main() {
    std::vector<int> v(1000000);
    std::iota(v.begin(), v.end(), 0);

    long long sum = std::reduce(std::execution::par, v.begin(), v.end(), 0LL);
    std::cout << "Sum: " << sum << '\n';
}
```

□ Insight: `std::execution::par` enables parallel execution on multicore CPUs.

□ Under the Hood: Implementations use thread pools or work-stealing algorithms to optimize task distribution.

□ Best Practices for Thread Safety

- Prefer higher-level abstractions (`async`, `future`, `lock_guard`).
- Avoid shared mutable state when possible.
- Always design with RAII and exception safety in mind.
- Use atomic operations for simple counters.
- Never assume thread scheduling order.
- Document synchronization assumptions explicitly.

Summary

Concept	API	C++ Version
`std::thread`, `join`, `detach`	Thread creation & management	C++11
`std::mutex`, `std::lock_guard`	Synchronization	C++11
`std::condition_variable`	Coordination	C++11
`std::future`, `std::async`	Asynchronous tasks	C++11
`std::atomic`	Lock-free synchronization	C++11
`std::execution::par`	Parallel algorithms	C++17

Final Mentor Note: Concurrency isn't about making code faster—it's about structuring computation so it *can* be fast *safely*. Always start simple, reason about data ownership, and scale concurrency only when correctness is guara