Gorgo Continuing C++

April 11, 2026

Contents

10. Concurrency

Modern CPUs have multiple cores, but the programs you have written so far use only one. **Concurrency** lets your program do several things at the same time — processing data on one thread while waiting for I/O on another, or splitting a large computation across cores to finish faster. Concurrency is powerful but treacherous: shared data accessed from multiple threads without coordination leads to **data races** — subtle, non-deterministic bugs that are among the hardest to diagnose. In this chapter you will learn to create threads, protect shared data with mutexes, coordinate threads with condition variables, use `std::async` for higher-level concurrency, and avoid common pitfalls.

`std::thread`

`std::thread` (`#include <thread>`) creates a new thread of execution:

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

void hello()
{
    std::cout << "Hola from a thread!\n";
}

int main()
{
    std::thread t(hello);  // start a new thread running hello()
    t.join();              // wait for it to finish

    return 0;
}
```

```
Hola from a thread!
```

`t.join()` blocks the calling thread until `t` finishes. You **must** either `join()` or `detach()` every thread before it is destroyed, or the program calls `std::terminate`.

Passing Arguments

You can pass arguments to the thread function:

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

void play(const std::string& song, int track_num)
{
    std::cout << "Track " << track_num << ": " << song << "\n";
}

int main()
{
    std::thread t(play, "Hung Up", 1);
    t.join();

    return 0;
}
```

```
Track 1: Hung Up
```

Trap: Arguments are **copied** into the thread by default. If you need to pass by reference, use `std::ref()`:

```
int count = 0;
std::thread t([](int& c) { c++; }, std::ref(count));
t.join();
// count is now 1
```

Without `std::ref`, the thread gets its own copy and `count` stays 0.

Detaching Threads

`t.detach()` lets the thread run independently. The thread continues even after the `std::thread` object is destroyed:

```
std::thread t(background_task);
t.detach();  // thread runs on its own
// Be careful: if t accesses local variables, they may be destroyed!
```

Trap: A detached thread has no way to report back. If it accesses variables from the creating scope, those variables may already be destroyed. Prefer `join()` unless you have a specific reason to detach.

Lambdas as Thread Functions

Lambdas are the most common way to write thread functions:

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

int main()
{
    int result = 0;

    std::thread t([&result]() {
        result = 42;
    });
    t.join();

    std::cout << "Result: " << result << "\n";

    return 0;
}
```

```
Result: 42
```

`std::jthread` (C++20)

`std::jthread` automatically joins in its destructor, so you never forget:

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

int main()
{
    std::jthread t([]() {
        std::cout << "Auto-joining thread\n";
```

```
    });
    // No need to call t.join() — destructor handles it

    return 0;
}
```

Mutexes and Locks

When two threads access the same data and at least one writes, you have a **data race** — undefined behavior. A **mutex** (mutual exclusion) prevents this by ensuring only one thread accesses the protected data at a time.

`std::mutex`

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

std::mutex mtx;
int shared_count = 0;

void increment(int times)
{
    for (int i = 0; i < times; ++i) {
        mtx.lock();
        shared_count++;
        mtx.unlock();
    }
}

int main()
{
    std::thread t1(increment, 100000);
    std::thread t2(increment, 100000);
    t1.join();
    t2.join();

    std::cout << "Count: " << shared_count << "\n";  // 200000

    return 0;
}
```

```
Count: 200000
```

Without the mutex, the count would be unpredictable — both threads could read the same value, increment it, and write back, losing an increment.

`std::lock_guard`

Calling `lock()` and `unlock()` manually is error-prone — if an exception is thrown between them, the mutex stays locked forever. `std::lock_guard` uses RAII (Chapter 9) to lock on construction and unlock on destruction:

```
void increment(int times)
{
    for (int i = 0; i < times; ++i) {
```

```
        std::lock_guard<std::mutex> guard(mtx);
        shared_count++;
        // mutex unlocked when guard goes out of scope
    }
}
```

Tip: Always use `lock_guard` (or `unique_lock`, `scoped_lock`) instead of calling `lock()`/`unlock()` directly. This is RAII applied to mutex locking.

`std::unique_lock`

`std::unique_lock` is like `lock_guard` but more flexible — you can unlock and relock it, and it is movable:

```
std::unique_lock<std::mutex> lock(mtx);
// ... critical section ...
lock.unlock();
// ... non-critical work ...
lock.lock();
// ... another critical section ...
```

`unique_lock` is required for condition variables (covered next).

`std::scoped_lock` (C++17)

`std::scoped_lock` can lock multiple mutexes at once without risking deadlock:

```
std::mutex mtx1, mtx2;

void transfer()
{
    std::scoped_lock lock(mtx1, mtx2);  // locks both, deadlock-free
    // ... modify data protected by both mutexes ...
}
```

If you tried to lock them separately, two threads could each lock one mutex and wait for the other — a **deadlock**.

Condition Variables

A **condition variable** lets one thread wait until another thread signals that something has happened. The classic use is the **producer-consumer** pattern:

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <queue>
#include <string>
#include <thread>
#include <vector>

std::queue<std::string> work_queue;
std::mutex mtx;
std::condition_variable cv;
bool done = false;
```

```cpp
void producer()
{
    std::vector<std::string> songs = {"Toxic", "Crazy", "Rehab"};
    for (const auto& song : songs) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            work_queue.push(song);
        }
        cv.notify_one();
    }
    {
        std::lock_guard<std::mutex> lock(mtx);
        done = true;
    }
    cv.notify_one();
}

void consumer()
{
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, []() { return !work_queue.empty() || done; });

        while (!work_queue.empty()) {
            std::cout << "Processing: " << work_queue.front() << "\n";
            work_queue.pop();
        }

        if (done) break;
    }
}

int main()
{
    std::thread prod(producer);
    std::thread cons(consumer);

    prod.join();
    cons.join();

    return 0;
}
```

```
Processing: Toxic
Processing: Crazy
Processing: Rehab
```

Key points about condition variables: - `cv.wait(lock, predicate)` unlocks the mutex, sleeps until notified, relocks, and checks the predicate. If the predicate is false, it goes back to sleep. - Always use the predicate form of `wait` to handle **spurious wakeups** — the OS can wake a thread without a notification. - `notify_one()` wakes one waiting thread; `notify_all()` wakes all of them.

Wut: Condition variables can experience **spurious wakeups** — the thread wakes up even though no one called `notify`. This is why you must always use the predicate form: `cv.wait(lock, predicate)`. Without it, your thread may proceed when it should still be waiting.

`std::async` and `std::future`

Threads are low-level. `std::async` (`#include <future>`) provides a higher-level way to run a task asynchronously and get its result:

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

std::string fetch_lyrics(const std::string& song)
{
    // Simulate slow operation
    return "Lyrics for: " + song;
}

int main()
{
    // Start async task
    std::future<std::string> result = std::async(std::launch::async,
        fetch_lyrics, "Somewhere Only We Know");

    std::cout << "Doing other work...\n";

    // Get the result (blocks if not ready yet)
    std::cout << result.get() << "\n";

    return 0;
}
```

```
Doing other work...
Lyrics for: Somewhere Only We Know
```

`std::async` returns a `std::future` that holds the result. Calling `future.get()` blocks until the result is ready and returns it. If the async task threw an exception, `get()` rethrows it.

Launch Policies

Policy	Behavior
`std::launch::async`	Guaranteed new thread
`std::launch::deferred`	Runs when `get()` is called (lazy)
Default (both)	Implementation chooses

Tip: `std::async` is the easiest way to parallelize independent tasks. Use it when you need a result from a background computation. Use `std::thread` when you need more control over the thread's lifetime.

Atomics

For simple operations on shared variables (counters, flags), mutexes are overkill. **Atomic operations** guarantee that reads and writes happen indivisibly, without locks:

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

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

void count_up(int times)
{
    for (int i = 0; i < times; ++i) {
        counter++;  // atomic increment — no mutex needed
    }
}

int main()
{
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(count_up, 100000);
    }
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Counter: " << counter << "\n";  // 400000

    return 0;
}
```

```
Counter: 400000
```

Common atomic operations:

```cpp
std::atomic<int> a(0);
a.store(5);             // write
int x = a.load();       // read
int old = a.exchange(10);  // swap and return old value
a.fetch_add(1);         // atomic increment
a.fetch_sub(1);         // atomic decrement
```

Tip: Use `std::atomic` for simple shared counters and flags. For anything more complex (protecting multiple variables or a data structure), use a mutex.

Thread Safety Pitfalls

Data Races

A data race occurs when two threads access the same variable, at least one writes, and there is no synchronization. Data races are **undefined behavior** — anything can happen:

```cpp
// BUG: data race on 'count'
int count = 0;

void bad_increment()
{
    for (int i = 0; i < 100000; ++i) {
        count++;  // not atomic, not protected
    }
}
```

Fix with a mutex, `lock_guard`, or `std::atomic<int>`.

Deadlock

A deadlock occurs when two threads each hold a lock the other needs:

```cpp
// Thread 1: lock A, then lock B
// Thread 2: lock B, then lock A
// Both wait forever!
```

Prevention strategies: - Always lock mutexes in the same order. - Use `std::scoped_lock` to lock multiple mutexes simultaneously. - Minimize the time you hold locks.

False Sharing

When two threads modify different variables that happen to share a cache line, the CPU invalidates the cache on every write, destroying performance. This is **false sharing**:

```cpp
// Both on the same cache line — slow!
struct Counters {
    int a;  // thread 1 writes this
    int b;  // thread 2 writes this
};
```

Fix by padding:

```cpp
struct alignas(64) Counters {
    alignas(64) int a;
    alignas(64) int b;
};
```

Try It: Concurrent Counter

Here is a program that demonstrates threads, mutexes, and atomics. Type it in, compile with `g++ -std=c++23 -pthread`, and experiment:

```cpp
#include <atomic>
#include <chrono>
#include <future>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

// Shared state
std::mutex mtx;
int mutex_count = 0;
std::atomic<int> atomic_count(0);
```

```cpp
void mutex_increment(int n)
{
    for (int i = 0; i < n; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        mutex_count++;
    }
}

void atomic_increment(int n)
{
    for (int i = 0; i < n; ++i) {
        atomic_count++;
    }
}

template<typename Func>
long long benchmark(Func f, int threads, int per_thread)
{
    auto start = std::chrono::high_resolution_clock::now();

    std::vector<std::thread> workers;
    for (int i = 0; i < threads; ++i) {
        workers.emplace_back(f, per_thread);
    }
    for (auto& t : workers) {
        t.join();
    }

    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
}

int main()
{
    const int threads = 4;
    const int per_thread = 1000000;

    auto ms1 = benchmark(mutex_increment, threads, per_thread);
    std::cout << "Mutex count:  " << mutex_count
              << " (" << ms1 << " ms)\n";

    auto ms2 = benchmark(atomic_increment, threads, per_thread);
    std::cout << "Atomic count: " << atomic_count
              << " (" << ms2 << " ms)\n";

    // std::async example
    auto fut = std::async(std::launch::async, []() {
        return std::string("Async result ready");
    });

    std::cout << fut.get() << "\n";

    return 0;
```

```
}
```

Compare the performance of mutex-based and atomic-based counting. Try removing the synchronization entirely and see how the count becomes incorrect.

Key Points

- **`std::thread`** creates a new thread of execution. Every thread must be `join()`ed or `detach()`ed before destruction.
- **`std::jthread`** (C++20) automatically joins on destruction.
- Arguments are copied into threads by default; use `std::ref()` for references.
- **`std::mutex`** provides mutual exclusion. Use **`std::lock_guard`** or **`std::scoped_lock`** (RAII) instead of manual `lock()`/`unlock()`.
- **`std::unique_lock`** is more flexible than `lock_guard` — it can be unlocked, relocked, and moved.
- **`std::scoped_lock`** (C++17) locks multiple mutexes simultaneously to prevent deadlock.
- **Condition variables** let threads wait for a signal. Always use the predicate form of `wait` to handle spurious wakeups.
- **`std::async`** and **`std::future`** provide high-level asynchronous computation. `future.get()` blocks until the result is ready.
- **`std::atomic`** provides lock-free operations for simple types (counters, flags).
- **Data races** (unsynchronized access) are undefined behavior. Use mutexes, atomics, or other synchronization primitives.
- **Deadlocks** occur when threads wait for each other's locks. Lock in consistent order or use `scoped_lock`.

Exercises

1. **Think about it:** Why must every `std::thread` be either joined or detached? What happens if you destroy a joinable thread?
2. **Where is the bug?**

   ```
   int total = 0;

   void add(int n) { total += n; }

   std::thread t1(add, 100);
   std::thread t2(add, 200);
   t1.join();
   t2.join();

   std::cout << total << "\n";
   ```

3. **Think about it:** Why does `std::lock_guard` not have `unlock()` and `lock()` methods, while `std::unique_lock` does? When would you need the extra flexibility?
4. **What does this program print?** (Approximately — exact output depends on scheduling.)

   ```
   std::atomic<int> x(0);

   std::thread t1([&]() { x++; x++; x++; });
   std::thread t2([&]() { x++; x++; x++; });
   t1.join();
   t2.join();

   std::cout << x << "\n";
   ```

5. **Where is the deadlock?**

```cpp
std::mutex m1, m2;

void thread_a()
{
    std::lock_guard<std::mutex> lock1(m1);
    std::lock_guard<std::mutex> lock2(m2);
    // ...
}

void thread_b()
{
    std::lock_guard<std::mutex> lock1(m2);
    std::lock_guard<std::mutex> lock2(m1);
    // ...
}
```

How would you fix it?

6. **Think about it:** When should you use `std::async` instead of creating a `std::thread` manually? What are the advantages?

7. **What value does `result` hold?**

```cpp
auto fut = std::async(std::launch::deferred, []() { return 6 * 7; });
// ... other work ...
int result = fut.get();
```

8. **Where is the bug?**

```cpp
std::mutex mtx;

void process()
{
    mtx.lock();
    if (some_condition()) {
        return;  // oops
    }
    // ... more work ...
    mtx.unlock();
}
```

9. **Think about it:** Why is `std::atomic` faster than using a mutex for simple counters? When would you still prefer a mutex over an atomic?

10. **Write a program** that uses four threads to compute the sum of a large vector (1 million elements). Each thread should sum one quarter of the vector. Use `std::async` and `std::future` to collect the partial sums, then print the total.