Back to Basics: Concurrency

I also do C++ training!

`arthur.j.odwyer@gmail.com`

Arthur O'Dwyer

2020-09-18

Outline

- What is a data race and how do we fix it? [3–12]
- C++11 `mutex` and RAII lock types [13–23]

Questions?

- `condition_variable` [24–28]
- Static initialization and `once_flag` [29–34]
- New C++17 and C++20 primitives [35–45]

Questions?

- The blue/green pattern [46–52]
- Bonus C++20 slides [53–58]

Questions?

What is concurrency?

- **Concurrency** means doing two things concurrently — “running together.” Maybe you’re switching back and forth between them.
    - Writing slides and answering email
- **Parallelism** means doing two things in parallel — simultaneously.
    - Writing slides and listening to music
- In extremely broad strokes, parallelism is a hardware problem (think multiple CPUs) and concurrency is a software problem (think time-sharing, but also Intel’s “hyperthreading”).

Why does C++ care about it?

Standard C++03 didn't have "threads." You'd just use some platform-specific library, such as pthreads. But then what could the Standard say about multithreaded programs?

Thread A	Thread B

`int x = 0;`
`start_thread_b();`
`x = 1;`
Will this write ever become "visible" to Thread B?
	`while (x != 1) {}`
	Will this loop ever terminate?

The compiler can rewrite accesses

Original code was...

```
int x = 0;
x = 1;
sleep(100ms);
x = 2;
sleep(100ms);
x = 3;
```

Effectively rewritten to...

```
int x = 3;

sleep(200ms);
```

Is this a legal optimization for a C++ compiler to perform?

Prior to C++11, the answer was “no idea.”

The C++11 answer is unambiguously “***yes***.”

No other thread is allowed to look at the variable x *while* this thread is modifying it; and without some kind of synchronization, there’s no way to ensure that this thread isn’t modifying it *right when* you happen to be looking at it.

The hardware can reorder accesses

Original code was...

```
char a[1000] = {};
a[0] = 1;
a[100] = 2;
a[1] = 3;
```

This is an ***extreme*** oversimplification and/or a flat-out lie — but it shows that what the code says is only loosely related to what the hardware does.

Effectively rewritten to...

```
cacheLine1 = a[0..63];
cacheLine[0] = 1;

cacheLine2 = a[64..127];
cacheLine2[36] = 2;
cacheLine1[1] = 3;

a[0..63] = cacheLine1;
a[64..127] = cacheLine2;
```

C++11 gave us a “memory model”

- Now a program consists of one or more ***threads of execution***
- Every write to a single memory location must ***synchronize-with*** all other reads or writes of that memory location, or else the program has undefined behavior
- *Synchronizes-with* relationships can be established by using various standard library facilities, such as `std::mutex` and `std::atomic<T>`

Starting a new thread

In C++03 pthreads, you'd create a new thread by calling a third-party library function.

In C++11, the standard library "owns" the notion of creating new threads. To create a thread, you create a `std::thread` object. The constructor argument is a callable that says what you want the thread to do.

```
std::thread threadB = std::thread([](){
    puts("Hello from threadB!");
});
```

Joining finished threads

The new thread starts executing “immediately.” When its job is done, the thread has nothing else to do: it becomes ***joinable***.

Call `.join()` on the `std::thread` object before destroying it. This call will ***block***, if necessary, until the other thread’s job is finished.

```
std::thread threadB = std::thread([](){
    puts("Hello from threadB!");
});
puts("Hello from threadA!");
threadB.join();
```

Getting the “result” of a thread

We don’t need any special way to return an “exit status” from a thread, because joining with a child thread is a synchronizing operation.

```
int result = 0;
std::thread threadB = std::thread([&](){
    puts("Hello from threadB!");
    result = 42;
});
puts("Hello from threadA!");
threadB.join();
printf("The result of threadB was %d\n", result);
```

This read synchronizes with this write, because the synchronizing operation `join()` returns *after* the write, but *before* the read.

Example of a data race on an `int`

```
using SC = std::chrono::steady_clock;
auto deadline = SC::now() + std::chrono::seconds(10);

int counter = 0;

std::thread threadB = std::thread([&](){
    while (SC::now() < deadline)
        printf("B: %d\n", ++counter);
});
while (SC::now() < deadline)
    printf("A: %d\n", ++counter);
threadB.join();
```

This is a data race.
No synchronization exists between these two accesses, and at least one of them is a write.
(In fact, both are.)

This program has UB.

Fixing the race via `std::atomic<T>`

```
using SC = std::chrono::steady_clock;
auto deadline = SC::now() + std::chrono::seconds(10);

std::atomic<int> counter = 0;

std::thread threadB = std::thread([&](){
    while (SC::now() < deadline)
        printf("B: %d\n", ++counter);
});
while (SC::now() < deadline)
    printf("A: %d\n", ++counter);
threadB.join();
```

This minor change completely fixes the ***physical*** data race! Every access to an `atomic` implicitly synchronizes with every other access to it.

(There's still a "semantic data race": different valid executions will produce different outputs. That might be considered a bug, but it's not UB.)

“Logical synchronization”

Problem statement:

```
std::thread threadB = std::thread([&](){
    waitUntilUnblocked();
    printf("Hello from B\n");
});
printf("Hello from A\n");
unblockThreadB();
threadB.join();
printf("Hello again from A\n");
```

Recall that threads start “running.” What if we need to set up a bit more state before letting the new thread run off on its own?

Can we tell thread B to wait until thread A unblocks it?

Yes, using any of several *synchronization primitives*.

First, a non-solution: busy-wait

This is ***not*** a solution.

```
std::atomic<bool> ready = false;
std::thread threadB = std::thread([&](){
    while (!ready) { }
    printf("Hello from B\n");
});
printf("Hello from A\n");
ready = true;
threadB.join();
printf("Hello again from A\n");
```

No data race, because `ready` is an `atomic`.

This is “spinning,” not “waiting.” Thread B never stops working; it just keeps checking over and over, never going to sleep. On a single-core system, this is a huge waste of time.

Also, the compiler can see that if this thread never sleeps then ready will never change, and (in theory) hoist it out of the loop. We still have UB here. Don’t do this, for many reasons.

A real solution: `std::mutex`

One way to solve the problem is `std::mutex`.

```
std::mutex mtx;
mtx.lock();
std::thread threadB = std::thread([&](){
    mtx.lock(); mtx.unlock();
    printf("Hello from B\n");
});
printf("Hello from A\n");
mtx.unlock();  // Now go!
threadB.join();
printf("Hello again from A\n");
```

mutex is a *mutual exclusion* mechanism.

I like to compare it to a coffeeshop bathroom key. When Alice holds the key, Bob can’t enter (and vice versa).

`mutex`’s `.lock()` method “acquires” the bathroom key (waiting in line for it, if necessary). The `.unlock()` method returns it so the next person can use it.

`std::mutex` frequently *protects* data

```
class TokenPool {
    std::mutex mtx_;
    std::vector<Token> tokens_;

    Token getToken() {
        mtx_.lock();
        if (tokens_.empty())
            tokens_.push_back(Token::create());
        Token t = std::move(tokens_.back());
        tokens_.pop_back();
        mtx_.unlock();
        return t;
    }
};
```

The coffeeshop bathroom key protects the facilities of the bathroom itself.

`TokenPool`'s `mtx_` protects its vector `tokens_`.

Every access (***read or write***) to `tokens_` must be done under a lock on `mtx_`. This is an invariant that must be preserved for correctness.

Protection must be complete

```
Token getToken() {
    mtx_.lock();
    if (tokens_.empty())
        tokens_.push_back(Token::create());
    Token t = std::move(tokens_.back());
    tokens_.pop_back();
    mtx_.unlock();
    return t;
}

size_t numTokensAvailable() const {
    return tokens_.size();
}
```

The code on this slide almost certainly has a thread-safety bug — a data race!

Suppose thread A calls `getToken()` while thread B calls `numTokensAvailable()`. Thread A takes the lock and starts popping from `tokens_`, while thread B (which didn't take any lock) is ***also*** reading `tokens_`. This is a data race and UB!

Protecting a variable with a mutex must be 100% or it's no good.

Plus, what about exception-safety?

```
Token getToken() {
    mtx_.lock();
    if (tokens_.empty())
        tokens_.push_back(Token::create());
    Token t = std::move(tokens_.back());
    tokens_.pop_back();
    mtx_.unlock();
    return t;
}
```

The code on this slide still has a potential bug. What happens if `Token::create()` or `push_back()` throws an exception?

We've locked the mutex, but the exception aborts execution of this function, so we never execute the line that would have unlocked it.

We should look for a way to follow RAII principles: Every "cleanup" action, including unlocking mutexes, should be done inside a destructor.

RAII to the rescue!

```
Token getToken() {
    std::lock_guard<std::mutex> lk(mtx_);
    if (tokens_.empty())
        tokens_.push_back(Token::create());
    Token t = std::move(tokens_.back());
    tokens_.pop_back();
    return t;
}

size_t numTokensAvailable() const {
    std::lock_guard lk(mtx_);
    return tokens_.size();
}
```

The class template `std::lock_guard<T>` is defined in the `<mutex>` header.

Its constructor locks the given mutex, and stores a reference to it.

Its destructor unlocks the mutex.

In C++17 and later, `lock_guard` can be used with CTAD, as shown in `numTokensAvailable()`.

A “mutex lock” is a resource

- A `new`’ed `T*` is a *resource*, in the sense that you need to do something special with it when you’re done with it: call `delete`.
- A locked `std::mutex` is a resource, in the sense that you need to do something special with it when you’re done with it: call `.unlock()`.
- We have `std::unique_ptr` to help us manage unique ownership of heap allocations.
- Likewise, we have `std::unique_lock` to help us manage unique ownership of mutex locks.

A “mutex lock” is a resource

Just as functions can pass or return ownership of a pointer, functions can pass or return ***ownership of a mutex lock***.

```
unique_ptr<int> foo(unique_ptr<int> p) {
    if (rand())
        p = nullptr;  // prematurely clean up
    return p;         // the resource
}

unique_lock<mutex> foo(unique_lock<mutex> lk) {
    if (rand())
        lk.unlock();  // prematurely clean up
    return lk;        // the resource
}
```

`std::lock_guard` is just a special case that can’t be passed around or prematurely cleaned up.

You might compare `std::lock_guard` to `boost::scoped_ptr` in C++03.

In fact, `scoped_lock` also exists

C++17 introduced `std::scoped_lock<Ts...>` as a “new and improved” `std::lock_guard<T>`. It can take multiple mutexes “at once,” although naming the resulting type is quite ugly without CTAD.

```
size_t numTokensAvailable() const {
    std::scoped_lock lk(mtx_);
    return tokens_.size();
}

void mergeTokensFrom(TokenPool& rhs) {
    std::scoped_lock lk(mtx_, rhs.mtx_);
    tokens_.insert(rhs.tokens_.begin(),
                   rhs.tokens_.end());
    rhs.tokens_.clear();
}
```

i.e., `scoped_lock<mutex>`

i.e., `scoped_lock<mutex, mutex>`

Question Break

Metaphor time!

This is Pat.

Pat is going to deliver a letter.

This is Frosty.

Frosty is waiting for a letter.

Mailboxes, flags, and cymbals

- Frosty goes to sleep next to the mailbox
- Pat puts a letter in the mailbox
- Pat raises the flag
- Pat clashes her cymbals
- Frosty wakes up, sees the flag raised, and looks in the mailbox

`condition_variable` for "wait until"

If we have no `Token::create()`, then when `tokens_` is empty we should ***block and wait until*** some other thread returns a token to the pool.

```
struct TokenPool {
    std::vector<Token> tokens_;
    std::mutex mtx_;
    std::condition_variable cv_;

    void returnToken(Token t) {
        std::unique_lock lk(mtx_);
        tokens_.push_back(t);
        lk.unlock();
        cv_.notify_one();
    }
```

Remember, every access (read or write) to `tokens_` must still be done under a `mtx_` lock, so as to avoid physical data races (UB).

"Notifying" the condition variable will wake up any one thread that's blocked on it. This is Pat's cymbals. (Pushing back `t` is delivering the letter.)

condition_variable for “wait until”

Here is the code that blocks and waits, using `std::condition_variable cv_`.

```
    Token getToken() {
        std::unique_lock lk(mtx_);
        while (tokens_.empty()) {
            cv_.wait(lk);
        }
        Token t = std::move(tokens_.back());
        tokens_.pop_back();
        return t;
    }
};
```

The “mailbox flag is raised” whenever `!tokens_.empty()`. At this point we hold the mutex lock, and know the flag is raised.

Remember, every access (read or write) to `tokens_` must still be done under a `mtx_` lock, so as to avoid physical data races.

Internally, `cv_.wait(lk)` will relinquish the lock and go to sleep; then, once it wakes up, it’ll re-acquire the lock.

This is Frosty.

`mutex` + `condition_variable`

- Whenever you have a “producer” and a “consumer”...
  - ...where the consumer must ***wait*** for the producer...
  - ...and production and consumption happen over and over...
  - Such as our `TokenPool`
  - Such as a task queue, work queue, or Go-style *channel*
- Then you almost certainly want a `mutex` plus a `condition_variable`.

If produce/consume happen only *once*, consider `std::promise/std::future`, which we aren’t going to talk about in this presentation. It still uses mutex+cv internally.

Of course try to use higher-level frameworks where you can, especially if your program is fundamentally concerned with concurrency. This presentation is geared to one-off tasks.

Waiting for initialization

C++11 made the core language know about threads in order to explain how concurrent writes to `int` cause UB but concurrent writes to `atomic<int>` don't.

But C++11 did another cool thing with its core-language-threading idea!

```
int main() {
    std::thread t1(foo), t2(foo);
    t1.join(); t2.join();
}

void foo() {
    static ComplicatedObject obj("some", "data");
    std::cout << "Hello from foo! obj.x is " << obj.x << "\n";
}
```

t1 and t2 arrive at this line concurrently. Which one performs the initialization?

And what is the other one doing while that's happening?

Thread-safe static initialization

In C++03, to make a “singleton” thread-safe, you had to experiment with things like “double-checked locking,” and of course it was all UB anyway.

In C++11, it’s as easy as:

```
inline auto& SingletonFoo::getInstance() {
    static SingletonFoo instance;
    return instance;
}
```

The first thread to arrive will start initializing the static `instance`.

Any more that arrive will ***block and wait*** until the first thread either succeeds (unblocking them all) or fails with an exception (unblocking one of them).

How to initialize a data member

But suppose you want a singleton per instance of some other object!

```
class Logger {
    std::optional<NetworkConnection> conn_;

    NetworkConnection& getConn() {
        if (!conn_.has_value()) {
            conn_ = NetworkConnection(defaultHost);
        }
        return *conn_;
    }
};
```

This code is clearly unsafe if two threads call `getConn()` concurrently while `conn_.has_value()` is false. They might both try to modify `conn_` without synchronization.

How to initialize a data member

```
class Logger {
    std::mutex mtx_;
    std::optional<NetworkConnection> conn_;

    NetworkConnection& getConn() {
        std::lock_guard<std::mutex> lk(mtx_);
        if (!conn_.has_value()) {
            conn_ = NetworkConnection(defaultHost);
        }
        return *conn_;
    }
};
```

We could add a mutex *protecting* every access to conn_.

This code is safe, but perhaps slower than it could be.

Initialize a member with `once_flag`

```
class Logger {
    std::once_flag once_;
    std::optional<NetworkConnection> conn_;

    NetworkConnection& getConn() {
        std::call_once(once_, []() {
            conn_ = NetworkConnection(defaultHost);
        });
        return *conn_;
    }
};
```

Here, the ***first*** access to `conn_` is protected by a `once_flag`.

This mimics how C++ does static initialization, but for a ***non***-static. Each `Logger` has its own `conn_`, protected by its own `once_`.

This access to `conn_` doesn’t need to be protected because it is definitely not the first access. We know that `conn_` must be initialized by now.

Comparison of C++11's primitives

`mutex`:

- Many threads can queue up on `lock`.
- Calling `unlock` unblocks exactly one waiter: the new "owner."
- `lock` blocks only if somebody "owns" the mutex.

`condition_variable`:

- Many threads can queue up on `wait`.
- Calling `notify_one` unblocks exactly one waiter.
- Calling `notify_all` unblocks all waiters.
- `wait` always blocks.

`once_flag`:

- Many threads can queue up on `call_once`.
- Failing at the callback unblocks exactly one waiter: the new "owner."
- Succeeding at the callback unblocks all waiters and sets the "done" flag.
- `call_once` blocks only if the "done" flag isn't set.

C++17 shared_mutex (R/W lock)

```
class ThreadSafeConfig {
    std::map<std::string, int> settings_;
    mutable std::shared_mutex rw_;

    void set(const std::string& name, int value) {
        std::unique_lock<std::shared_mutex> lk(rw_);
        settings_.insert_or_assign(name, value);
    }

    int get(const std::string& name) const {
        std::shared_lock<std::shared_mutex> lk(rw_);
        return settings_.at(name);
    }
};
```

`unique_lock` calls `rw_.lock()` in its constructor and `rw_.unlock()` in its destructor.

`shared_lock` calls `rw_.lock_shared()` in its constructor and `rw_.unlock_shared()` in its destructor.

C++20 counting_semaphore

```
                                              max        initial
class AnonymousTokenPool {
    std::counting_semaphore<256> sem_{100};
    void getToken() {
        sem_.acquire();  // may block
    }
    void returnToken() {
        sem_.release();
    }
};
```

A semaphore is a “bag of poker chips.”

`.acquire()` removes a chip (perhaps blocking until a chip is available).

`.release()` returns a chip. We assume that you acquired a chip earlier. If you didn’t, such that the bag overflows, that’s UB.

Chips are indistinguishable, interchangeable, and (unlike mutex locks) *not tied to any particular thread.*

C++20 counting_semaphore

```
using Sem = std::counting_semaphore<256>;
struct SemReleaser {
    bool operator()(Sem *s) const { s->release(); }
};
class AnonymousTokenPool {
    Sem sem_{100};
    using Token = std::unique_ptr<Sem, SemReleaser>;
    Token borrowToken() {
        sem_.acquire();  // may block
        return Token(&sem_);
    }
};
```

This slight change makes our token pool safer to use.

Destroying a Token now automatically returns it to the pool.

See my CppCon 2019 talk on smart pointers for more on this pattern.

C++20 `std::latch`

- A `latch` is kind of like a semaphore, in that it has an integer counter that starts positive and counts down toward zero.
- `latch.wait()` blocks until the counter reaches zero.
- `latch.count_down()` decrements the counter.
  - If the counter reaches zero then this unblocks all the waiters.
- `latch.arrive_and_wait()` decrements *and* begins waiting.

Use a `std::latch` as a one-shot “starting gate” mechanism: “Wait for everyone to arrive at this point, then unblock everyone simultaneously.”

`latch` is like `once_flag` in that there is no way to “reset” its counter.

C++20 `std::barrier<>`

- A `barrier` is essentially a resettable `latch`.
- `barrier.wait()` blocks until the counter reaches zero, as before.
- `barrier.arrive()` decrements the counter.
    - If the counter reaches zero then this unblocks all the waiters...
    - ...and begins a new phase with the counter reset to its initial value.
- `barrier.arrive_and_wait()` decrements *and* waits, as before.

Use `std::barrier` as a “pace car” mechanism: “Stop everyone as they arrive at this point. Once everyone’s caught up, unblock everyone, and atomically refresh the barrier to stop them on their next trip around the loop.”

C++20 `std::barrier<>` arcana

There's a lot of subtleties to `std::barrier` which I am glossing over.

- `std::barrier`, unlike `std::latch`, is a class template!
  - The template parameter has a default, so CTAD ***permits*** you to say `barrier b;` in most places. But I recommend `barrier<>`, just like `less<>`.
  - It defines a "completion function" to be called right before everyone is unblocked. The default is "do nothing," which is usually fine.
- "Lapping the pace car" produces UB. Falling two laps behind produces UB.
- `myBarrier.arrive_and_drop()` lets your car drop out of the race forever.

Synchronization with `std::latch`

This gives us another solution to our thread-starting problem.

```
std::latch myLatch(2);
std::thread threadB = std::thread([&](){
    myLatch.arrive_and_wait();
    printf("Hello from B\n");
});
printf("Hello from A\n");
myLatch.arrive_and_wait();
threadB.join();
printf("Hello again from A\n");
```

Synchronization with `std::latch`

The main thread is going to wait in `join()` anyway, so in fact we can just do:

```
std::latch myLatch(1);
std::thread threadB = std::thread([&](){
    myLatch.wait();
    printf("Hello from B\n");
});
printf("Hello from A\n");
myLatch.arrive();
threadB.join();
printf("Hello again from A\n");
```

Synchronization with `std::barrier`

```
std::barrier b(2, []{ puts("Green flag, go!"); });
std::thread threadB = std::thread([&](){
    printf("B is setting up\n");
    b.arrive_and_wait();
    printf("B is running\n");
});
printf("A is setting up\n");
b.arrive_and_wait();
printf("A is running\n");
threadB.join();
```

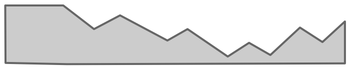
CTAD alert!

This code uses `std::barrier` with a `CompletionFunction` of lambda type. You should see A and B setting up (in some order), followed by "Green flag, go!", followed by A and B running (in some order).

Comparison of C++20's primitives

`counting_semaphore`:

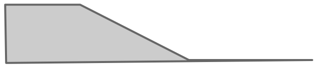
- The counter goes



- Many threads can queue up on `acquire`.
- `acquire` blocks only as long as the counter is zero.
- Calling `release` unblocks exactly one waiter.

`latch`:

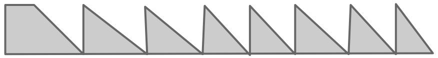
- The counter goes



- Many threads can queue up on `wait`.
- `wait` blocks only until the counter becomes zero.

`barrier`:

- The counter goes



- Many threads can queue up on `wait`.
- `wait` blocks only until the counter becomes zero.
- (Supports some extra complexity which we mostly glossed over.)

Question Break

One-slide intro to C++11 promise/future

```
std::future<int> f1 = std::async([]() {
    puts("Hello from thread A!");
    return 1;
});

std::future<int> f2 = std::async([]() {
    puts("Hello from thread B!");
    return 2;
});

int result = f1.get() + f2.get();
  // automatically blocks until the results
  // are available from threads A and B
```

This slide shows the `future` side (Frosty). Not shown: the `promise` side (Pat).

`std::async` *creates a new thread* on each call. The STL's `async` has serious perf caveats, but it's nice API design — this factory function saves the programmer from managing raw threads by hand.

If `std::async` sounds relevant to your use-case, don't base any architecture decisions on this talk!

That goes triple for C++20 coroutines, which I won't even attempt to demonstrate.

Patterns for sharing data

- Remember: Protect shared data with a mutex.
  - You must protect ***every*** access, both reads and writes, to avoid UB.
  - Maybe use a reader-writer lock (`std::shared_mutex`) for perf.
- Remember: Producer/consumer? Use `mutex` + `condition_variable`.
- Best of all, though: ***Avoid sharing mutable data between threads.***
  - Make the data immutable.
  - Clone a “working copy” for yourself, mutate that copy, and then quickly “merge” your changes back into the original when you’re done.

The “blue/green” pattern

- The name “blue/green” comes from devops.

  > ***Blue/green deployment*** *is an application release model that gradually transfers user traffic from a previous version of an app or microservice to a nearly identical new release — both of which are running in production.*
  >
  > *The old version can be called the* ***blue*** *environment while the new version is called the* ***green*** *environment. Once production traffic is fully transferred from blue to green, blue can ... be pulled from production.*

- We might use this to “publish” a new version of mutable global state, such as a global configuration object.

The “blue/green” pattern (write-side)

```
using ConfigMap = std::map<std::string, std::string>;

std::atomic<std::shared_ptr<const ConfigMap>> g_config;

void setDefaultHostname(const std::string& value) {
    std::shared_ptr<const ConfigMap> blue = g_config.load();
    do {
        std::shared_ptr<ConfigMap> green =
            std::make_shared<ConfigMap>(*blue);
        green->insert_or_assign("default.hostname", value);
    } while (g_config.compare_exchange_strong(blue, std::move(green)));
}
```

Clone the entire map to get a private copy we can write to!

“Publish” our changes: Expect `g_config` to still be `blue`. If so, store `green`. Otherwise, update `blue` and go around again.

The “blue/green” pattern (read-side)

```
using ConfigMap = std::map<std::string, std::string>;

std::atomic<std::shared_ptr<const ConfigMap>> g_config;

// ...

std::shared_ptr<const std::string> getDefaultHostname() {

    std::shared_ptr<const ConfigMap> blue = g_config.load();
    const std::string& value = blue.at("default.hostname");
    return std::shared_ptr<const std::string>(std::move(blue), &value);
}
```

“Aliasing constructor” alert!
The `blue` ConfigMap will stay alive for as long as it is the current `g_config` ***or*** anyone is still holding one of these `shared_ptrs`.

In conclusion

- Unprotected data races are UB
  - Use `std::mutex` to protect ***all*** accesses (both reads and writes)
- Thread-safe static initialization is your friend
  - Use `std::once_flag` only when the initializee is non-static
- `mutex` + `condition_variable` are best friends
- C++20 gives us “counting” primitives like semaphore and latch
- But if your program is ***fundamentally multithreaded***, look for higher-level facilities: promise/future, coroutines, ASIO, TBB

Questions?

followed by some bonus slides

Bonus: C++20 `std::atomic_ref<T>`

- Basically, `std::atomic<T>` protects your ***data*** against data races
    - Gives you a `T` that you *cannot access* non-atomically
    - Solves the problem at the typesystem level
- `std::atomic_ref<T>` protects your ***accesses*** against data races
    - You supply a plain old `T` object of your own, anywhere in memory
    - As long as you access it only through `atomic_ref`, no two protected accesses will race with each other
    - Similar to protecting the data with a mutex, but can be optimized better for small/trivial data

Unfortunately, `atomic_ref` works ***only*** for trivial data; no specialization for `shared_ptr`.

Bonus: C++20 `std::jthread`

- Recall that if you have a `std::thread`, you must call `.join()` on it before it is destroyed; otherwise your program terminates.
  - By the way, you can also discharge this responsibility via `t.detach()`
- A joinable `std::thread` is a ***resource*** requiring ***management***, just the same as a heap-allocated pointer or a locked mutex.
- So we should have an RAII type for it, right?
- C++20 gives us `std::jthread` (“joining thread”)...

Bonus: C++20 `std::jthread`

```
int main() {
    std::barrier<> b(2);
    std::jthread threadB = std::jthread([&](){
        printf("B is setting up\n");
        b.arrive_and_wait();
        printf("B is running\n");
    });
    may_throw("A is setting up\n");
    b.arrive_and_wait();
    printf("A is running\n");
} // threadB is joined automatically in its destructor
```

`std::jthread` is just like `std::thread`, except that it joins automatically in its destructor.

But do you see a problem here?

Bonus: C++20 `std::jthread`

```
int main() {
    std::barrier<> b(2);
    std::jthread threadB = std::jthread([&](){
        printf("B is setting up\n");
        b.arrive_and_wait();
        printf("B is running\n");
    });
    may_throw("A is setting up\n");
    b.arrive_and_wait();
    printf("A is running\n");
} // threadB is joined automatically in its destructor
```

If this line throws, then `std::jthread`'s destructor won't `std::terminate`; it will simply block forever, waiting for the `barrier`'s counter to reach zero.

C++20 `std::jthread` is cancellable

```
bool ready = false;
std::mutex m;  // m protects ready
std::condition_variable_any cv;

std::jthread threadB([&](std::stop_token token) {
    printf("B is setting up\n");
    std::unique_lock lk(m);
    cv.wait(lk, token, [&]{ return ready; });
    if (token.stop_requested()) return;
    printf("B is running\n");
});

may_throw("A is setting up\n");
{ std::scoped_lock lk(m); ready = true; }
cv.notify_one();
printf("A is running\n");
```

`jthread` can magically provide a `stop_token` to its job.

Both accesses to `ready` happen under the mutex lock.

Meanwhile, if this line throws, the `jthread` will notify its `stop_token` before it joins. The `stop_token` wakes up the `condition_variable`, allowing the task to "cancel" itself.

Questions?