

# **Multithreading is the answer. What is the question?**

(part 2)

Ansel Sermersheim  
CppNow - May 2016

# Introduction

- Why is multithreading complicated
- Example of multithreading
- A better way
- Future plans

# Why is Multithreading Complicated

- Abstraction, Coding, Communication
  - pair programming
  - multiple developers working on the same code
  - you read your own code a few months later
- Mutexes and Locks
  - multi-user programming relies on record locks
  - multithreading locks are not the same, however it uses the same word

# Why is Multithreading Complicated

- When data must be read and modified from different threads
- Worst case scenario
  - two classes, each one has a struct data member
  - during the destruction of either class both structures must be updated
  - how do you handle this with minimal blocking

# Why is Multithreading Complicated

- myMutex1 is a random variable name which everyone must agree to use

```
std::mutex myMutex1;  
std::shared_timed_mutex myMutex2;  
std::recursive_mutex myMutex3;
```

```
std::lock_guard<std::mutex> lock1(myMutex1);  
std::unique_lock<std::mutex> lock2(myMutex1, std::defer_lock);  
std::shared_lock<shared_timed_mutex> lock3(myMutex2, std::try_lock);
```

# Threading 101

- When a person enters a phone booth to make a call
  - the person must hold the door handle of the booth to prevent another person from using the phone
  - when finished with the call, let go of the door handle
  - now another person is allowed to use the phone

thread	a person
mutex	the door handle
lock	the person's hand
resource	the phone

# Review Multithreading Example

# Multithreading Analogy (E)

- Stage: A kitchen
  - one oven, one brick pizza oven, one ice cream maker
    - shared resources
- Requirement:
  - anyone can randomly order pizza, garlic knots, apple pie, or ice cream
- Solution: pandemonium



# Multithreading Analogy (E)

```
Oven vikingOven;  
std::mutex vikingOven_mutex;  
  
Oven brickOven;  
std::mutex brickOven_mutex;  
  
IceCreamMaker iceCreamMaker;  
std::mutex iceCream_maker_mutex;  
  
class Food { ... };  
  
class Pizza { ... };  
class GarlicKnots { ... };  
class ApplePie { ... };  
class IceCream { ... };
```

# Multithreading Analogy (E)

```
void eat(Food && food) {  
    std::cout << "Patron was served: " << food.name();  
};  
  
using PatronTicket = std::future<std::unique_ptr<Food>>;  
using ChefTicket   = std::promise<std::unique_ptr<Food>>;
```

# Multithreading Analogy (E)

```
std::thread patron1( []() {  
    PatronTicket knots      = orderGarlicKnots();  
    PatronTicket pizza      = orderPizza();  
    PatronTicket iceCream   = orderIceCream();  
  
    eat(knots.get());  
    eat(pizza.get());  
    eat(icecream.get());  
});
```

```
std::thread patron2( []() {  
    PatronTicket iceCream   = orderIceCream();  
    PatronTicket applePie   = orderApplePie();  
  
    eat(iceCream.get());  
    eat(applePie.get());  
});
```

# Multithreading Analogy (E)

```
class Order { ... };  
std::atomic<bool> restaurantOpen;  
threadsafe_queue<Order> orderQueue;  
  
std::thread chef1( [&]() {  
    while(restaurantOpen) {  
        Order nextOrder = orderQueue.dequeue();  
        nextOrder.process();  
    }  
});  
  
std::thread chef2( [&]() {  
    while(restaurantOpen) {  
        Order nextOrder = orderQueue.dequeue();  
        nextOrder.process();  
    }  
});
```

# Multithreading Analogy (E)

```
PatronTicket orderPizza() {
    std::shared_ptr<ChefTicket> chefTicket =
        std::make_shared<ChefTicket>();
    PatronTicket patronTicket = chefTicket->get_future();

    Order order{ [chefTicket]() {
        std::unique_ptr<Pizza> pizza = std::make_unique<Pizza>();
        pizza.addSauce();
        pizza.addCheese();
        std::lock_guard<std::mutex> lock(brickOven_mutex);
        pizza = brickOven.bake(std::move(pizza));
        chefTicket->set_value(std::move(pizza));
    }};

    orderQueue.queue(std::move(order));
    return patronTicket;
}
```

# Code review

# Multithreading

// example 1 - any issues?

```
ComplicatedObject * createObject(int param1, double param2) {  
    ComplicatedObject * retval;  
  
    retval = new ComplicatedObject();  
    retval->doSomething(param1);  
    retval->somethingElse(param2);  
  
    return retval;  
}
```

# Multithreading

```
// example 2 - any issues?
```

```
class MyCache {  
public:  
    void insert(std::string key, ComplicatedObject * element);  
    ComplicatedObject * lookup(std::string key) const;  
  
private:  
    std::map<std::string, ComplicatedObject *> m_cache;  
    std::shared_timed_mutex m_cacheMutex;  
};  
  
ComplicatedObject * MyCache::lookup(std::string key) {  
    std::shared_lock<std::shared_timed_mutex> lock(m_cacheMutex);  
  
    return m_cache[key];  
}
```



- Problems with example 2
  - returns a raw ptr, who is responsible for deleting it
  - what if someone else deletes the object
  - what if I delete the object and but I do not remove it from the `std::map`
  - if the key is not found in the map, a reference to the mapped value with a `nullptr` is inserted in the map
    - undefined behavior since the lock is a “read” lock

## A Better Way . . .

- `class guarded<T>`

# class guarded<T>

```
template <typename T, typename M = std::mutex>
class guarded {
public:
    using handle = std::unique_ptr<T, deleter>;

    template <typename... Us>
    guarded(Us &&... data);

    handle lock();
    handle try_lock();

    template <class Duration>
    handle try_lock_for(const Duration & duration);
    template <class TimePoint>
    handle try_lock_until(const TimePoint & timepoint);

private:
    T m_obj;
    M m_mutex;
};
```

# class guarded<T>

```
template <typename T, typename M>
template <typename... Us>
guarded<T, M>::guarded(Us &&... data) : m_obj(std::forward<Us>(data)...)
{
}
```

```
template <typename T, typename M>
auto guarded<T, M>::lock() -> handle
{
    std::unique_lock<M> lock(m_mutex);
    return handle(&m_obj, deleter(std::move(lock)));
}
```

# class guarded<T>

```
template <typename T, typename M>
auto guarded<T, M>::try_lock() -> handle
{
    std::unique_lock<M> lock(m_mutex, std::try_to_lock);

    if (lock.owns_lock()) {
        return handle(&m_obj, deleter(std::move(lock)));
    } else {
        return handle(nullptr, deleter(std::move(lock)));
    }
}
```

# class guarded<T>

```
class deleter
{
    public:
        using pointer = T *;

        deleter(std::unique_lock<M> lock) : m_lock(std::move(lock))
        {
        }

        void operator()(T * ptr) {
            if (m_lock.owns_lock()) {
                m_lock.unlock();
            }
        }

    private:
        std::unique_lock<M> m_lock;
};
```

## A Better Way . . .

- `class guarded<T>`
- `class shared_guarded<T>`

# class shared\_guarded<T>

```
template <typename T, typename M = std::shared_timed_mutex>
class shared_guarded {
public:
    using handle = std::unique_ptr<T, deleter>;

    template <typename... Us>
    shared_guarded(Us &&... data);

    handle lock();
    handle try_lock();

    template <class Duration>
    handle try_lock_for(const Duration & duration);

    template <class TimePoint>
    handle try_lock_until(const TimePoint & timepoint);
```



# class shared\_guarded<T>

```
public:
```

```
    using shared_handle = std::unique_ptr<const T, shared_deleter>;
```

```
    shared_handle lock_shared() const;
```

```
    shared_handle try_lock_shared() const;
```

```
    template <class Duration>
```

```
    shared_handle try_lock_shared_for(const Duration & duration) const;
```

```
    template <class TimePoint>
```

```
    shared_handle try_lock_shared_until(const TimePoint & timepoint) const;
```

```
private:
```

```
    T m_obj;
```

```
    mutable M m_mutex;
```

```
};
```

## Example 2 Revisited -- Using shared\_guarded<T>

```
class MyCache {
public:
    void insert(std::string key, std::shared_ptr<ComplicatedObject> element);
    std::shared_ptr<ComplicatedObject> lookup(std::string key) const;

private:
    shared_guarded<std::map<std::string,
                        std::shared_ptr<ComplicatedObject>>> m_cache;
};

std::shared_ptr<ComplicatedObject> MyCache::lookup(std::string key) {
    auto handle = m_cache.lock_shared();
    auto iter   = handle->find(key);

    if(iter != handle->end()) {
        return iter->second;
    }
    return nullptr;
}
```

## Example 2 Revisited -- Using shared\_guarded<T>

```
// any issues?
```

```
void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    auto handle = m_cache->lock();
    handle->emplace(key, element);
}
```

## A Better Way . . .

- `class guarded<T>`
- `class shared_guarded<T>`
- `class ordered_guarded<T>`

# class ordered\_guarded<T>

```
template <typename T, typename M = std::shared_timed_mutex>
class ordered_guarded
{
public:
    using shared_handle = std::unique_ptr<const T, shared_deleter>;

    template <typename... Us>
    ordered_guarded(Us &&... data);

    template <typename Func>
    void modify(Func && func);
```

# class ordered\_guarded<T>

```
shared_handle lock_shared() const;  
shared_handle try_lock_shared() const;
```

```
template <class Duration>  
shared_handle try_lock_shared_for(const Duration & duration) const;
```

```
template <class TimePoint>  
shared_handle try_lock_shared_until(const TimePoint & timepoint) const;
```

```
private:
```

```
    T          m_obj;  
    mutable M m_mutex;
```

```
};
```

# class ordered\_guarded<T>

```
template <typename T, typename M>
template <typename Func>
void ordered_guarded<T, M>::modify(Func && func)
{
    std::lock_guard<M> lock(m_mutex);

    func(m_obj);
}
```

## Example 2 Revisited -- Using ordered\_guarded<T>

```
// any issues?
```

```
void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    m_cache.modify(
        [&key, &element]
        (std::map<std::string, std::shared_ptr<ComplicatedObject>> & map)
        {
            map.emplace(key, element);
        });
}
```



## A Better Way . . .

- `class guarded<T>`
- `class shared_guarded<T>`
- `class ordered_guarded<T>`
- `class deferred_guarded<T>`

# class deferred\_guarded<T>

```
template <class T>
typename std::add_lvalue_reference<T>::type declref();

template <typename T, typename M = std::shared_timed_mutex>
class deferred_guarded
{
public:
    using shared_handle = std::unique_ptr<const T, shared_deleter>;

    template <typename... Us>
    deferred_guarded(Us &&... data);

    template <typename Func>
    void modify_detach(Func && func);

    template <typename Func>
    auto modify_async(Func && func) ->
        typename std::future<decltype(std::declval<Func>()(declref<T>()))>;
```

# class deferred\_guarded<T>

```
shared_handle lock_shared() const;  
shared_handle try_lock_shared() const;
```

```
template <class Duration>  
shared_handle try_lock_shared_for(const Duration & duration) const;
```

```
template <class TimePoint>  
shared_handle try_lock_shared_until(const TimePoint & timepoint) const;
```

```
private:
```

```
    T          m_obj;  
    mutable M m_mutex;  
    mutable std::atomic<bool> m_pendingWrites;  
    mutable guarded<std::vector<std::packaged_task<void(T &)>>> m_pendingList;  
};
```

## Example 2 Revisited -- Using deferred\_guarded<T>

```
// any issues?
```

```
void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    m_cache.modify_detach(
        [key, element]
        (std::map<std::string, std::shared_ptr<ComplicatedObject>> & map)
        {
            map.emplace(key, element);
        });
}
```

## A Better Way . . .

- `class guarded<T>`
- `class shared_guarded<T>`
- `class ordered_guarded<T>`
- `class deferred_guarded<T>`
- `class lr_guarded<T>`

# class lr\_guarded<T>

```
template <typename T, typename Mutex = std::mutex>
class lr_guarded
{
public:
    using shared_handle = std::unique_ptr<const T, shared_deleter>;

    template <typename... Us>
    lr_guarded(Us &&... data);

    template <typename Func>
    void modify(Func && f);
```

# class lr\_guarded<T>

```
// shared access
```

```
shared_handle lock_shared() const;
```

```
shared_handle try_lock_shared() const;
```

```
template <class Duration>
```

```
shared_handle try_lock_shared_for(const Duration & duration) const;
```

```
template <class TimePoint>
```

```
shared_handle try_lock_shared_until(const TimePoint & timepoint) const;
```

# class lr\_guarded<T>

private:

```
T          m_left;  
T          m_right;  
std::atomic<bool> m_readingLeft;  
std::atomic<bool> m_countingLeft;  
mutable std::atomic<int> m_leftReadCount;  
mutable std::atomic<int> m_rightReadCount;  
mutable Mutex          m_writeMutex;
```

```
};
```



# class lr\_guarded<T>

```
template <typename T, typename M>
template <typename Func>
void lr_guarded<T, M>::modify(Func && func)
{
    std::lock_guard<M> lock(m_writeMutex);
    T * firstWriteLocation;
    T * secondWriteLocation;

    bool local_readingLeft = m_readingLeft.load();

    if (local_readingLeft) {
        firstWriteLocation = &m_right;
        secondWriteLocation = &m_left;
    } else {
        firstWriteLocation = &m_left;
        secondWriteLocation = &m_right;
    }
}
```

# class lr\_guarded<T>

```
try {  
    func(*firstWriteLocation);  
} catch (...) {  
    *firstWriteLocation = *secondWriteLocation;  
    throw;  
}  
m_readingLeft.store(! local_readingLeft);  
bool local_countingLeft = m_countingLeft.load();  
  
if (local_countingLeft) {  
    while (m_rightReadCount.load() != 0) {  
        std::this_thread::yield(); }  
} else {  
    while (m_leftReadCount.load() != 0) {  
        std::this_thread::yield(); }  
}
```

# class lr\_guarded<T>

```
m_countingLeft.store(! local_countingLeft);

if (local_countingLeft) {
    while (m_leftReadCount.load() != 0) {
        std::this_thread::yield();
    }
} else {
    while (m_rightReadCount.load() != 0) {
        std::this_thread::yield();
    }
}
try {
    func(*secondWriteLocation);
} catch (...) {
    *secondWriteLocation = *firstWriteLocation;
    throw;
}
}
```

## Example 2 Revisited -- Using `lr_guarded<T>`

```
// any issues?
```

```
void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    m_cache.modify(
        [&key, &element]
        (std::map<std::string, std::shared_ptr<ComplicatedObject>> & map)
        {
            map.emplace(key, element);
        });
}
```

## A Better Way . . .

- `class guarded<T>`
- `class shared_guarded<T>`
- `class ordered_guarded<T>`
- `class deferred_guarded<T>`
- `class lr_guarded<T>`
- `class cow_guarded<T>`

# class cow\_guarded<T>

```
template <typename T, typename Mutex = std::mutex>
class cow_guarded
{
public:
    template <typename... Us>
    cow_guarded(Us &&... data);

    handle lock();
    handle try_lock();

    template <class Duration>
    handle try_lock_for(const Duration & duration);

    template <class TimePoint>
    handle try_lock_until(const TimePoint & timepoint);
```

# class cow\_guarded<T>

```
shared_handle lock_shared() const;  
shared_handle try_lock_shared() const;
```

```
template <class Duration>  
shared_handle try_lock_shared_for(const Duration & duration) const;
```

```
template <class TimePoint>  
shared_handle try_lock_shared_until(const TimePoint & timepoint) const;
```

```
private:
```

```
    lr_guarded<std::shared_ptr<const T>> m_data;  
    Mutex m_writeMutex;
```

```
};
```

# class cow\_guarded<T>

```
class deleter
{
public:
    using pointer = T *;

    deleter(std::unique_lock<Mutex> && lock, cow_guarded & guarded)
        : m_lock(std::move(lock)), m_guarded(guarded), m_cancelled(false)
    {
    }

    void cancel()
    {
        m_cancelled = true;
    }
}
```



# class cow\_guarded<T>

```
void operator()(T * ptr)
{
    if (m_cancelled) {
        delete ptr;
    } else if (ptr) {
        std::shared_ptr<const T> newPtr(ptr);

        m_guarded.m_data.modify([newPtr]
            (std::shared_ptr<const T> & ptr)
            { ptr = newPtr; } );
    }

    if (m_lock.owns_lock()) {
        m_lock.unlock();
    }
}
```

# class cow\_guarded<T>

```
private:
    std::unique_lock<Mutex> m_lock;
    cow_guarded &          m_guarded;
    bool                  m_cancelled;
};
```

## Example 2 Revisited -- Using `cow_guarded<T>`

```
// any issues?
```

```
void MyCache::insert(std::string key, std::shared_ptr<ComplicatedObject> element)
{
    auto handle = m_cache->lock();
    handle->emplace(key, element);
}
```

## A Better Way . . . Recap

- `class guarded<T>`
  - exclusive locks
  - C++11
- `class shared_guarded<T>`
  - exclusive locks
  - shared locks
  - C++14 or `boost::thread`
- `class ordered_guarded<T>`
  - shared locks
  - blocking modifications to shared data (via lambda)
  - C++14 or `boost::thread`

## A Better Way . . . Recap

- `class deferred_guarded<T>`
  - shared locks
  - nonblocking modifications to data (via lambda)
  - deadlock free eventual consistency
  - C++14 or `boost::thread`
- `class lr_guarded<T>`
  - shared access without locks
  - blocking modifications to data (via lambda)
  - readers block writers
  - readers never see data older than the previous write
  - C++11

## A Better Way . . . Recap

- `class cow_guarded<T>`
  - shared access without locks
  - blocking modifications to data (via lambda)
  - only other writers can block writers
  - readers see a snapshot of data
  - unwanted modifications can be discarded
  - C++11

## Example 3

```
class MyCache {
public:
    std::shared_ptr<ComplicatedObject> lookup(std::string key) const;

    void insert(std::string key, std::shared_ptr<ComplicatedObject> element);
    void insert_batch(std::map<std::string, std::shared_ptr<ComplicatedObject>>);

private:

    // must be called with m_cacheMutex held
    void internal_insert(std::string key, std::shared_ptr<ComplicatedObject> e);

    std::map<std::string, ComplicatedObject *> m_cache;
    std::shared_timed_mutex m_cacheMutex;
};
```

## Example 3 Revisited -- using deferred\_guarded<T>

```
class MyCache {
public:
    std::shared_ptr<ComplicatedObject> lookup(std::string key) const;

    void insert(std::string key, std::shared_ptr<ComplicatedObject> element);
    void insert_batch(std::map<std::string, std::shared_ptr<ComplicatedObject>>);

private:

    using shared_handle = deferred_guarded<std::map<std::string,
        std::shared_ptr<ComplicatedObject>>>::shared_handle;

    void internal_insert(std::string key, std::shared_ptr<ComplicatedObject> e,
        shared_handle& cache);

    deferred_guarded<std::map<std::string,
        std::shared_ptr<ComplicatedObject>>> m_cache;
};
```



## Example 3 Revisited -- using deferred\_guarded<T>

```
using MyCache = deferred_guarded<std::map<std::string,  
                                     std::shared_ptr<ComplicatedObject>>>>;
```

# Multithreading Analogy -- using guarded<T>

```
PatronTicket orderPizza() {
    std::shared_ptr<ChefTicket> chefTicket =
        std::make_shared<ChefTicket>();
    PatronTicket patronTicket = chefTicket->get_future();

    Order order{ [chefTicket]() {
        std::unique_ptr<Pizza> pizza = std::make_unique<Pizza>();
        pizza.addSauce();
        pizza.addCheese();
        ovenHandle = brickOven.lock();
        pizza = ovenHandle->bake(std::move(pizza));
        chefTicket->set_value(std::move(pizza));
    }};

    orderQueue.queue(std::move(order));
    return ticket;
}
```

# Future plans

- Future enhancements
  - per-element locking for guarded containers
  - integration with condition variables
  - locking multiple guarded objects simultaneously
  - use LibGuarded in CsSignal
  - integration with a work queue

# Wrap Up

# Libraries & Applications

- CopperSpice
  - libraries for developing GUI applications
- PepperMill
  - converts Qt headers to CS standard C++ header files
- CsSignal Library
  - new standalone thread aware signal / slot library
- LibGuarded
  - new standalone multithreading library for shared data

# Libraries & Applications

- KitchenSink
  - one program which contains 30 demos
  - links with almost every CopperSpice library
- Diamond
  - programmers editor which uses the CS libraries
- DoxyPress & DoxyPressApp
  - application for generating documentation

## Where to find these Projects

- [github.com/copperspice/libguarded/](https://github.com/copperspice/libguarded/)
- [www.copperspice.com](http://www.copperspice.com)
- [download.copperspice.com](http://download.copperspice.com)
- [forum.copperspice.com](http://forum.copperspice.com)
- [ansel@copperspice.com](mailto:ansel@copperspice.com)
- Questions? Comments?