# Lock-free programming with modern C++

Timur Doumler

ACCU Conference, 26 May 2017

# overview

- motivation
- useful definitions
- `std::atomic` interface
- exchanging values between threads
- lock-free queue implementation

multiple threads exchanging data

# Standard approach: locks

## C++11

- `std::mutex, std::recursive_mutex, std::timed_mutex`
- `std::lock_guard, std::unique_lock`
- `std::condition_variable`

## C++14

- `std::shared_lock`

## C++17

- `std::shared_mutex`
- `std::scoped_lock`

# Lock-free programming: why bother?

- Hard to write & maintain
- Often, overall performance is not better

BUS
SIGHTSEEING
BUS
50

# Real-time environment

- cannot block and wait due to strict time constraints

  → no locks

  → no memory allocations/deallocations

  → no calls into 3rd party code

- why?

  → no guarantee how long you will be blocking

  → minimise dependence on thread scheduler

  → avoid priority inversion

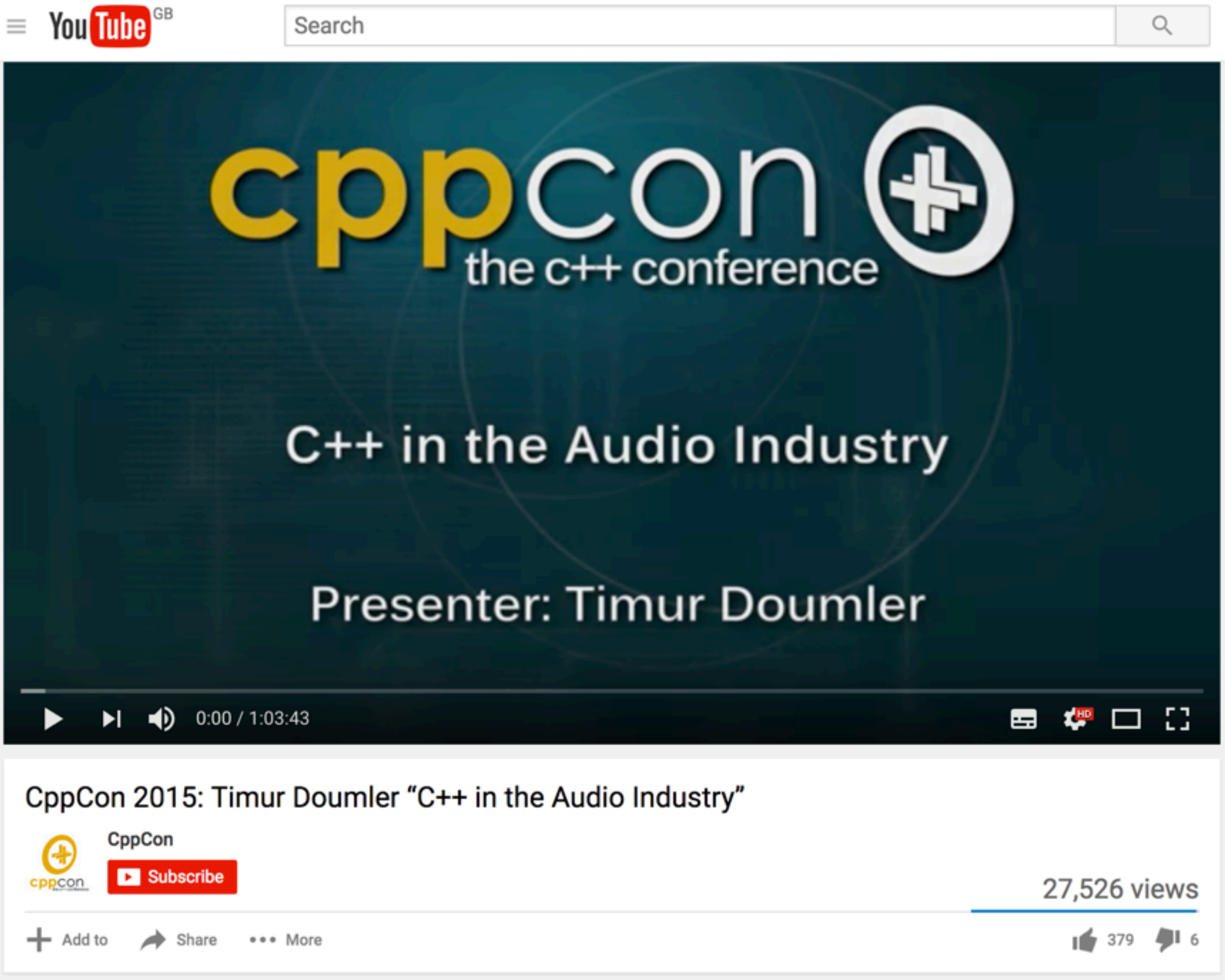
# Real-time environment

- audio processing
- finance
- embedded
- science & engineering
- …

# Example: audio processing

real-time audio callback

```
void audioCallback (float** channelData,
                              int numChannels, int numSamples)
{
    // write some data into channelData!
}
```



cppcon

the c++ conference

C++ in the Audio Industry

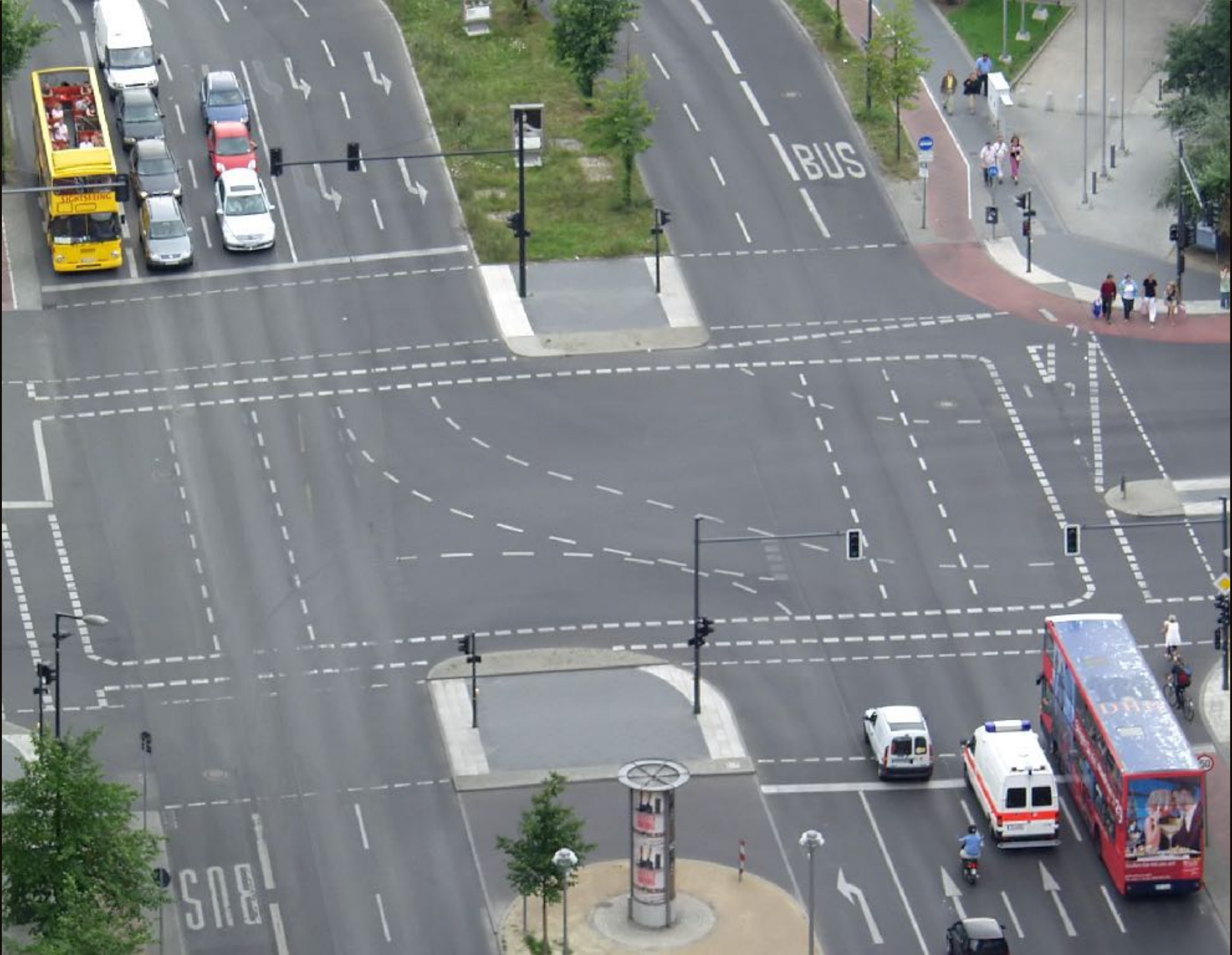
Presenter: Timur Doumler

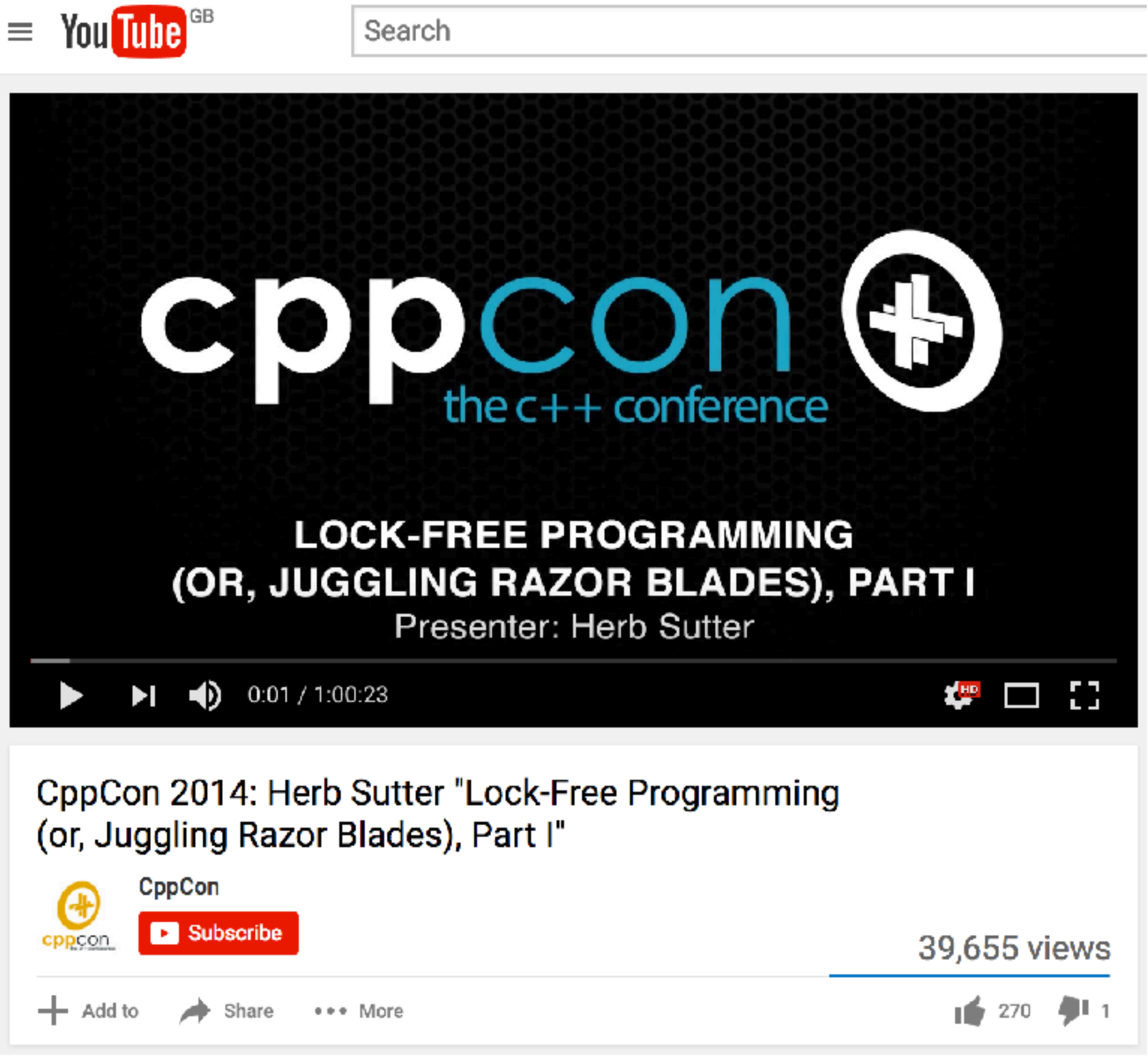
CppCon 2015: Timur Doumler “C++ in the Audio Industry”

CppCon

27,526 views

for explanation
Victoria's Secret
Bath & Body Works
La Senza


BUS
SIGHTSEEING
BUS
50



CppCon 2014: Herb Sutter "Lock-Free Programming (or, Juggling Razor Blades), Part I"

CppCon

39,655 views

## lock-free

at least one thread will always make progress

## wait-free

all threads will always make progress

# the C++ memory model

“A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. Two or more threads of execution can update and access separate memory locations without interfering with each other.”

# the C++ memory model

“A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. Two or more threads of execution can update and access separate memory locations without interfering with each other.”

# the C++ memory model

“A memory location is either an object of scalar type* or a maximal sequence of adjacent bit-fields all having non-zero width. Two or more threads of execution can update and access separate memory locations without interfering with each other.”

*built-in arithmetic (`int`, `float`, `bool`…) / pointer / enum

# the C++ memory model

If two or more threads can update and access the same memory location:

data race = undefined behaviour

```
{
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
}
```

```
{
    std::lock_guard lock (mutex);

    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
    ...............
}
```

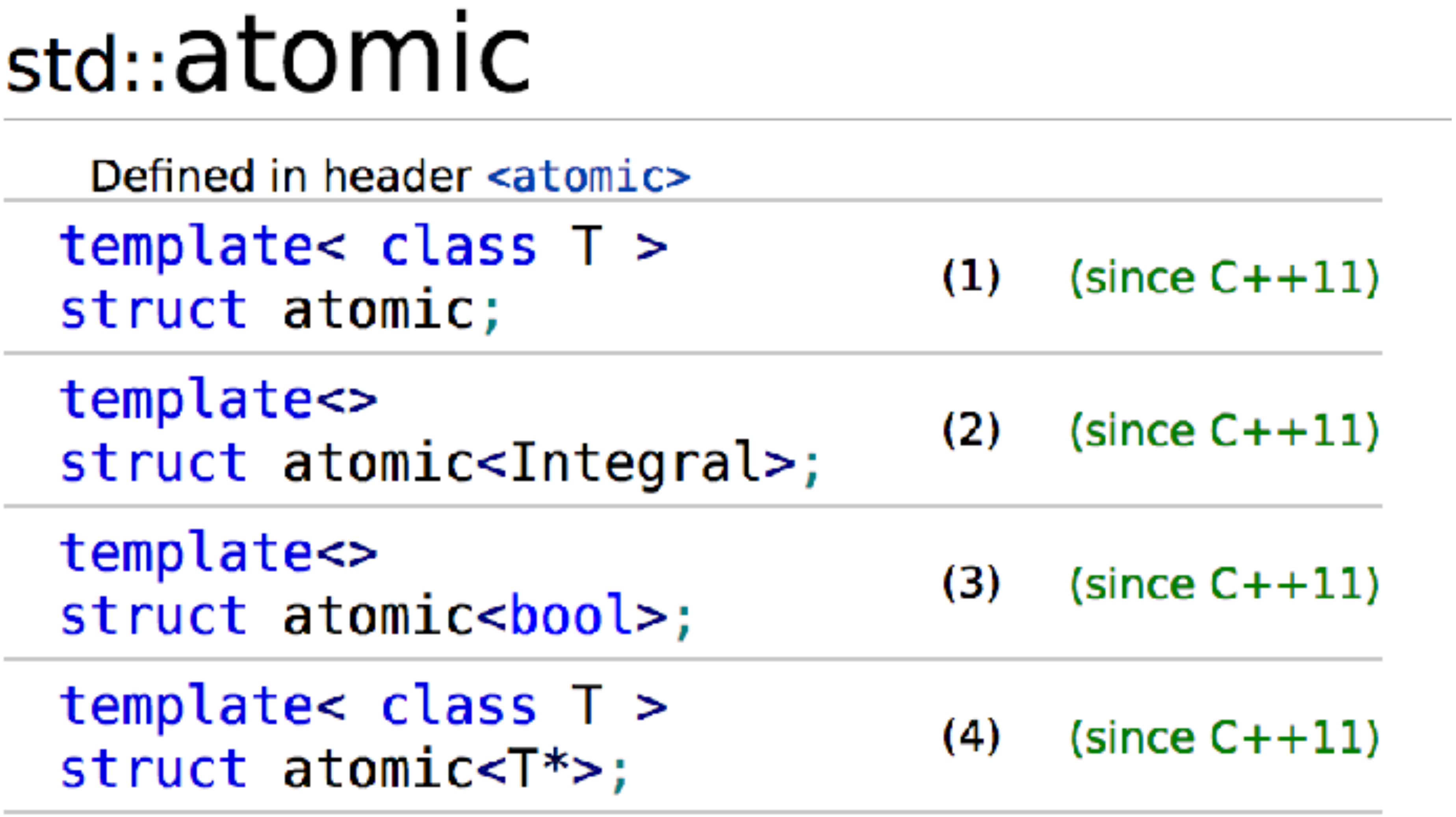
```
{
    ...............
    ...............
    ...............

    {
        std::lock_guard lock (mutex);
        ...............
        ...............
        ...............
    }

    ...............
    ...............
    ...............
    ...............
}
```

```
{
    ...............
    ...............
    ...............

    {
        std::lock_guard lock (mutex);
        ...............
    }

    ...............
    ...............
    ...............
    ...............
}
```

```
{
    ...............
    ...............
    ...............

    /* atomic instruction */

    ...............
    ...............
    ...............
    ...............
}
```

`std::atomic`

inherently race-free type.



# std::atomic

Defined in header `<atomic>`

| | | |
|---|---|---|
| `template< class T >`<br>`struct atomic;` | (1) | (since C++11) |
| `template<>`<br>`struct atomic<Integral>;` | (2) | (since C++11) |
| `template<>`<br>`struct atomic<bool>;` | (3) | (since C++11) |
| `template< class T >`<br>`struct atomic<T*>;` | (4) | (since C++11) |

```
std::atomic<int> pos;
```

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

```cpp
std::atomic<int> pos;

// write value 3 into pos
pos.store (3);

// read current value from pos
int currentPos = pos.load();

// write value 42 into pos and retrieve previous value
int previousPos = pos.exchange (42);

// if pos == expected, sets pos to desired and returns true.
// otherwise does nothing and returns false.
if (pos.compare_exchange_strong (expected, desired))
    return;

// if pos == expected, sets pos to desired and returns true.
// otherwise does nothing and returns false. (use in loops)
while (! pos.compare_exchange_weak (expected, desired))
    ;
```

```
pos.store (3);
```

~~`// same as:`~~

~~`pos = 3;`~~

```
int currentPos = pos.load();
```

~~`// same as:`~~

~~`int currentPos = pos;`~~

```
std::atomic<int> pos;

// write value 3 into pos
pos.store (3);

// read current value from pos
int currentPos = pos.load();

// write value 42 into pos and retrieve previous value
int previousPos = pos.exchange (42);

// if pos == expected, sets pos to desired and returns true.
// otherwise does nothing and returns false.
if (pos.compare_exchange_strong (expected, desired))
    return;

// if pos == expected, sets pos to desired and returns true.
// otherwise does nothing and returns false. (use in loops)
while (! pos.compare_exchange_weak (expected, desired))
    ;
```

# the problem with lock-free code

```
{
    if (readPos != data.end())
        ++readPos;
}
```

# the problem with lock-free code

```
{
    if (readPos != data.end())

        ++readPos;
}
```

# the problem with lock-free code

```
{
    if (readPos != data.end())

        ++readPos;
}
```

# the solution

```
{
    auto oldReadPos = readPos.load();
    if (oldReadPos == data.end())
        return;

    auto newReadPos = oldReadPos + 1;
    readPos.compare_exchange_strong (oldReadPos, newReadPos);
}
```

# atomic integer arithmetic

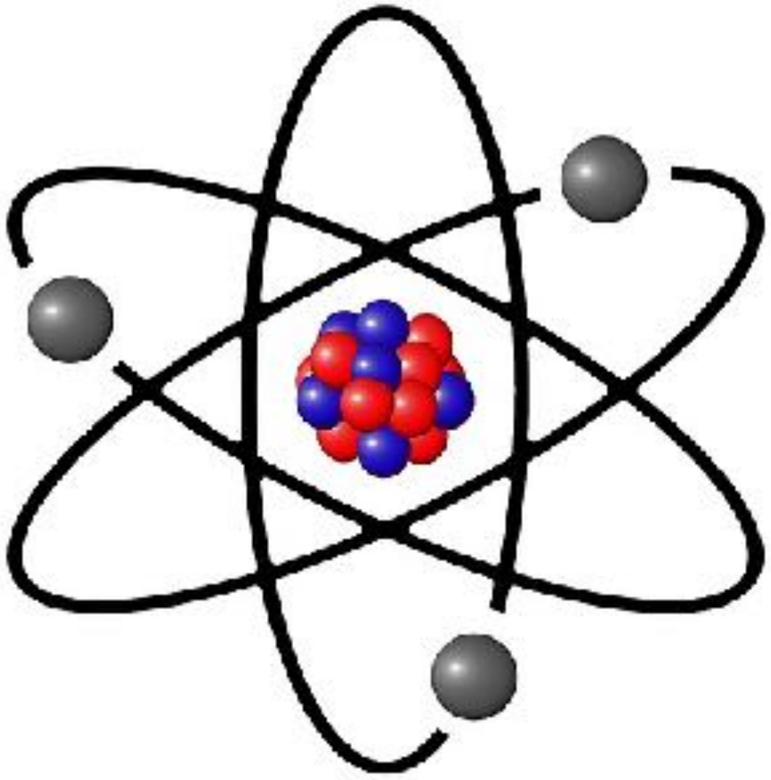
```
std::atomic<int> a;

++a;
--a;
a++;
a--;

a += 3;
a -= 3;
a &= 3;
a |= 3;
a ^= 3;

a.fetch_add (3);
a.fetch_sub (3);
a.fetch_and (3);
a.fetch_or (3);
a.fetch_xor (3);
```

# atomic integer arithmetic

```
std::atomic<int> a;

++a;
--a;
a++;
a--;

a += 3;
a -= 3;
a &= 3;
a |= 3;
a ^= 3;

a.fetch_add (3);  // useful: returns previous value
a.fetch_sub (3);
a.fetch_and (3);
a.fetch_or (3);
a.fetch_xor (3);
```

# floating point atomic

- No template specialisation for `float` and `double`
  (Proposal P0020: Floating Point Atomic, H. Carter Edwards et al.)
- `operator+=`, `operator-=` etc. do not exist
- `store` and load is fine
- `compare_exchange` is there, but not meaningful



atomic

lock-free

```
a.is_lock_free();
```

| T | std::atomic<T>::is_lock_free() ? |
|---|---|
| bool | ✓ |
| int | ✓ |
| double | ✓ |
| Widget* | ✓ |
| std::complex<double> | ? |
| Widget | ✗ |

```
a.is_lock_free();    // per instance!
```

# since C++17

```
std::atomic<T>::is_always_lock_free();
```

# memory order

```
std::memory_order_relaxed
std::memory_order_consume
std::memory_order_acquire
std::memory_order_release
std::memory_order_acq_rel
std::memory_order_seq_cst
```

# memory order

```
std::memory_order_relaxed
std::memory_order_consume
std::memory_order_acquire
std::memory_order_release
std::memory_order_acq_rel
std::memory_order_seq_cst   // default
```

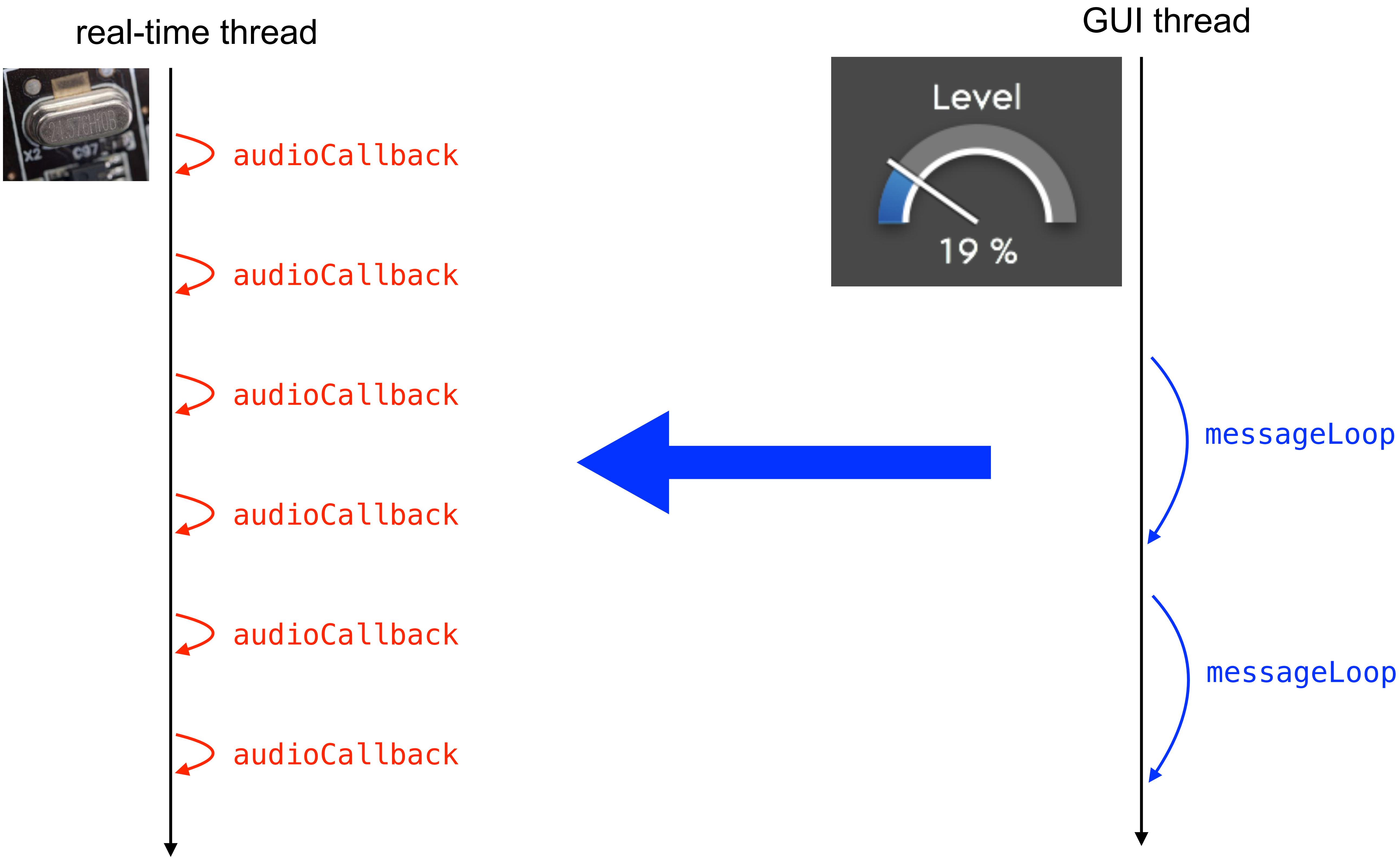
tbs

talk following this one:

“Atomic’s memory orders, what for?”

Frank Birbacher

# exchanging a `float` between threads



real-time thread

GUI thread

audioCallback

audioCallback

audioCallback

audioCallback

audioCallback

audioCallback

messageLoop

messageLoop

```
struct Synthesiser
{
    float level;


    // GUI thread:
    void levelChanged (float newValue)
    {
        level = newValue;
    }


    // real-time thread:
    void audioCallback (float* buffer, int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }
};
```

```
struct Synthesiser
{
    float level;


    // GUI thread:
    void levelChanged (float newValue)
    {
        level = newValue;
    }


    // real-time thread:
    void audioCallback (float* buffer, int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level * getNextAudioSample();
    }
};
```

**data race = undefined behaviour**

```
struct Synthesiser
{
    std::atomic<float> level;


    // GUI thread:
    void levelChanged (float newValue)
    {
        level.store (newValue);
    }


    // real-time thread:
    void audioCallback (float* buffer, int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level.load() * getNextAudioSample();
    }
};
```

```
struct Synthesiser
{
    std::atomic<float> level;


    // GUI thread:
    void levelChanged (float newValue)
    {
        level.store (newValue);
    }


    // real-time thread:
    void audioCallback (float* buffer, int numSamples) noexcept
    {
        for (int i = 0; i < numSamples; ++i)
            buffer[i] = level.load() * getNextAudioSample();
    }
};
```

**inefficient, and perhaps different result!**

```
struct Synthesiser
{
    std::atomic<float> level;


    // GUI thread:
    void levelChanged (float newValue)
    {
        level.store (newValue);
    }


    // real-time thread:
    void audioCallback (float* buffer, int numSamples) noexcept
    {
        const float currentLevel = level.load();

        for (int i = 0; i < numSamples; ++i)
            buffer[i] = currentLevel * getNextAudioSample();
    }
};
```

# exchanging an object between threads

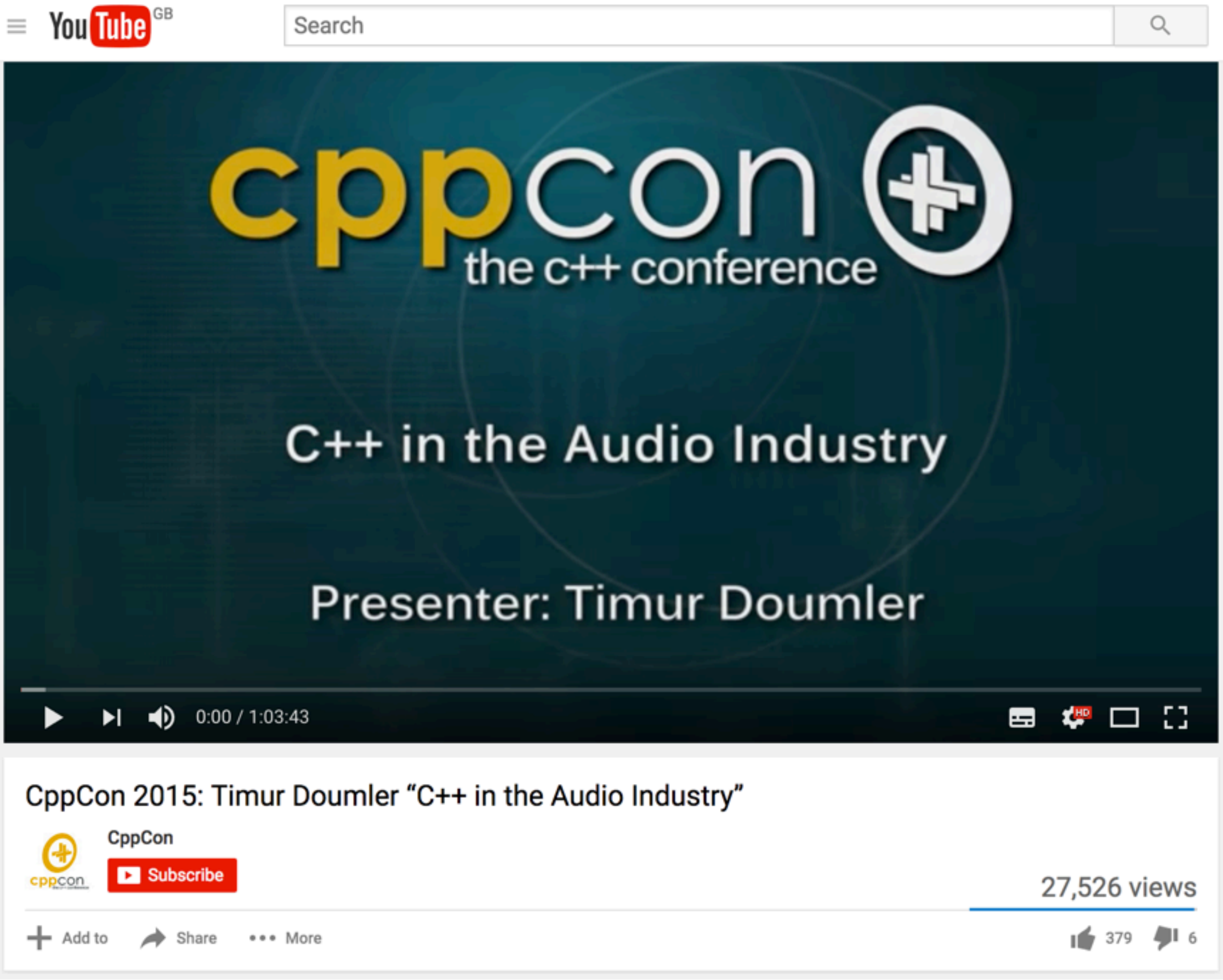
```
struct Foo
{
    ~~std::atomic<Widget> widget;~~
};
```

```
struct Foo
{
    std::atomic<Widget*> widget;
};
```

```cpp
struct Foo
{
    std::atomic<Widget*> widget {nullptr};
};
```

```
struct Foo
{
    std::atomic<Widget*> widget {nullptr};

    // thread 1:
    void modifyWidget()
    {
        auto* newWidget = new Widget (/* setup */);

        auto* oldWidget = widget.exchange (newWidget);

        // dispose of oldWidget
    }
};
```

```
struct Foo
{
    std::atomic<Widget*> widget {nullptr};

    // thread 1:
    void modifyWidget()
    {
        auto* newWidget = new Widget (/* setup */);

        auto* oldWidget = widget.exchange (newWidget);

        // dispose of oldWidget
    }

    // thread 2:
    void useWidget()
    {
        auto* currentWidget = widget.exchange (nullptr);

        // do work with currentWidget
        // dispose of oldWidget
    }
};
```



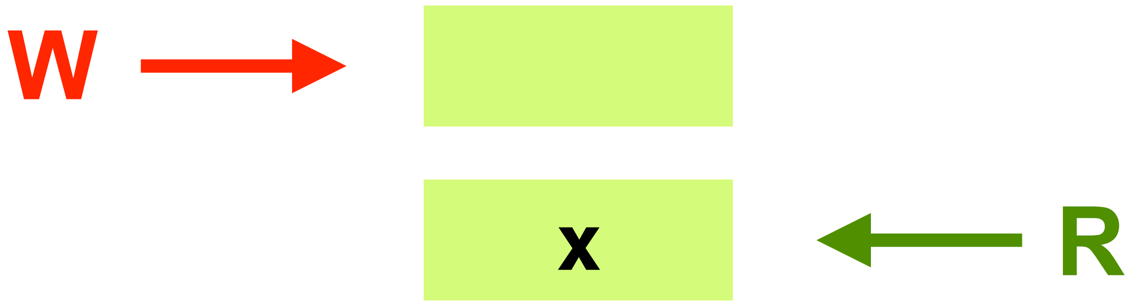
CppCon 2015: Timur Doumler “C++ in the Audio Industry”

```cpp
struct Foo
{
    std::atomic<Widget*> widget {nullptr};

    // thread 1:
    void modifyWidget()
    {
        auto* newWidget = new Widget (/* setup */);

        auto* oldWidget = widget.exchange (newWidget);

        // dispose of oldWidget
    }

    // thread 2:
    void useWidget()
    {
        auto* currentWidget = widget.exchange (nullptr);

        // do work with currentWidget
        // dispose of oldWidget
    }
};
```

# lock-free queue / fifo

W

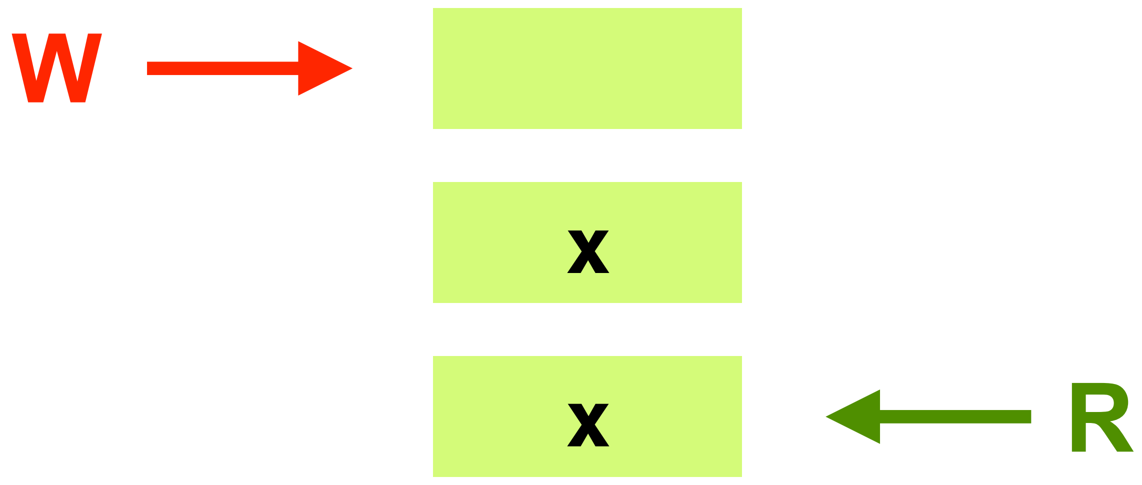
R

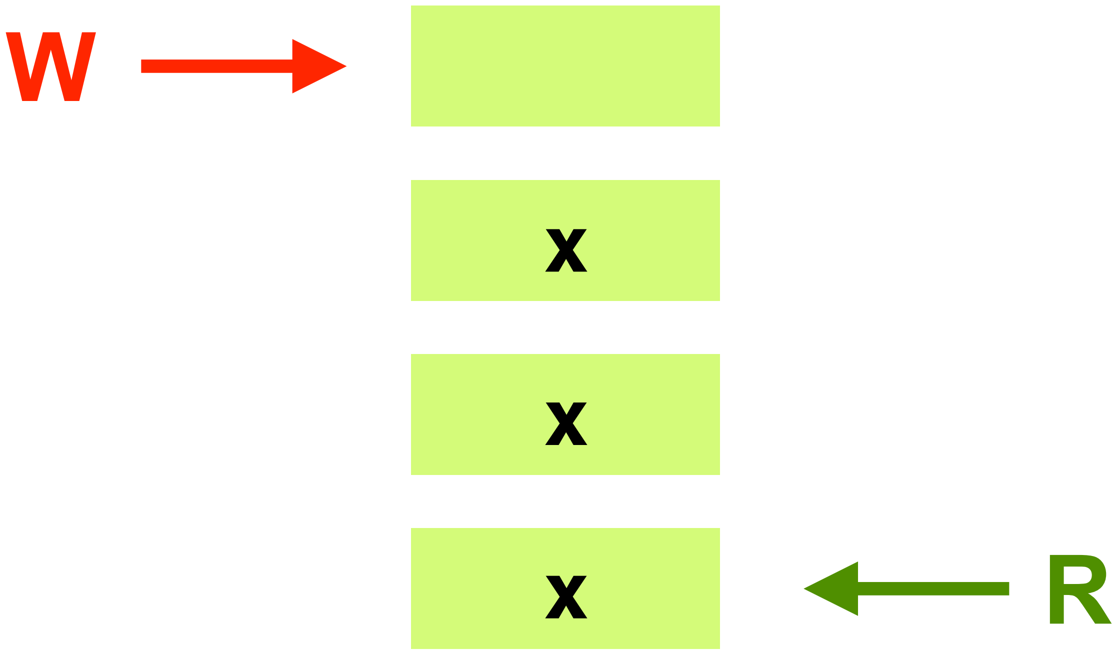


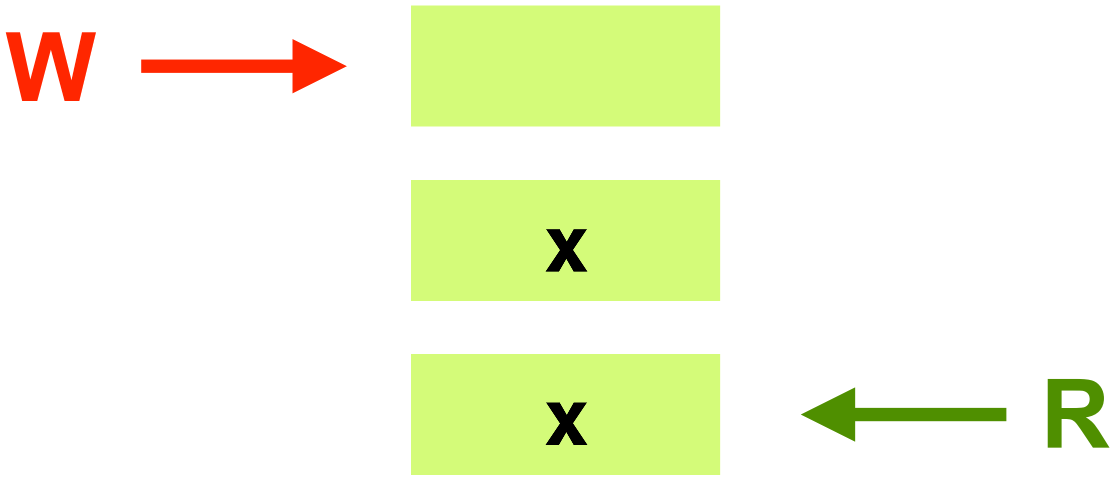
W

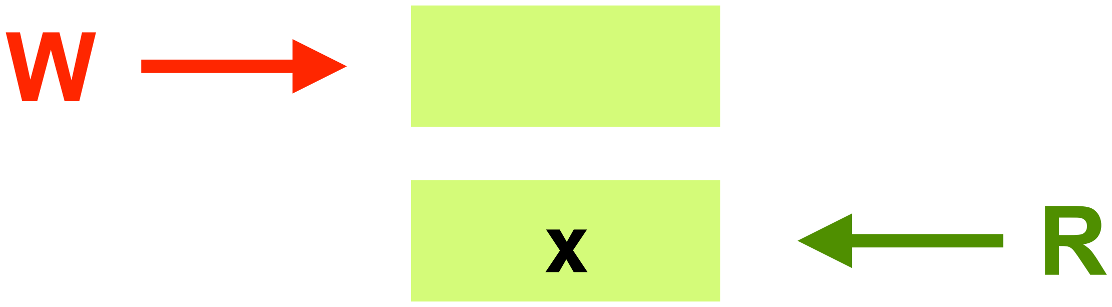
x

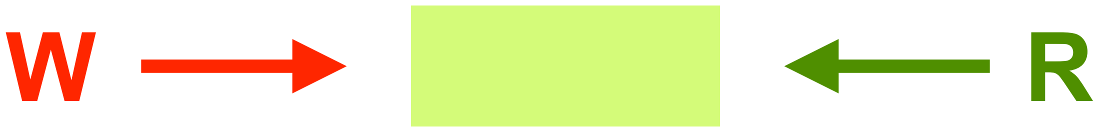
R


W
x
x
R


W
x
x
x
R


W
x
x
R


W
x
R



single producer

single consumer

```
template <typename T>
class Queue
{
    /** Adds an element to the queue. */
    void push (const T& newElement);

    /** Removes the front element from the queue, copies it into returnedElement
        and returns true. If the queue is empty, does nothing and returns false.
    */
    bool pop (T& elementReturned);
};
```

# requirements

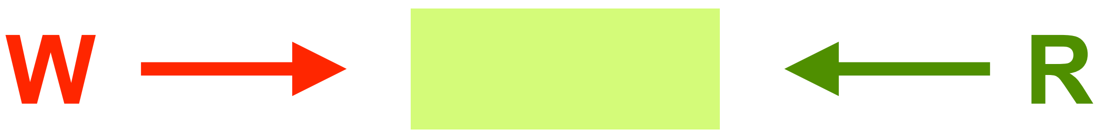
- no data races
- no allocations
- no locks

# no data races

queue is empty

`W == R`

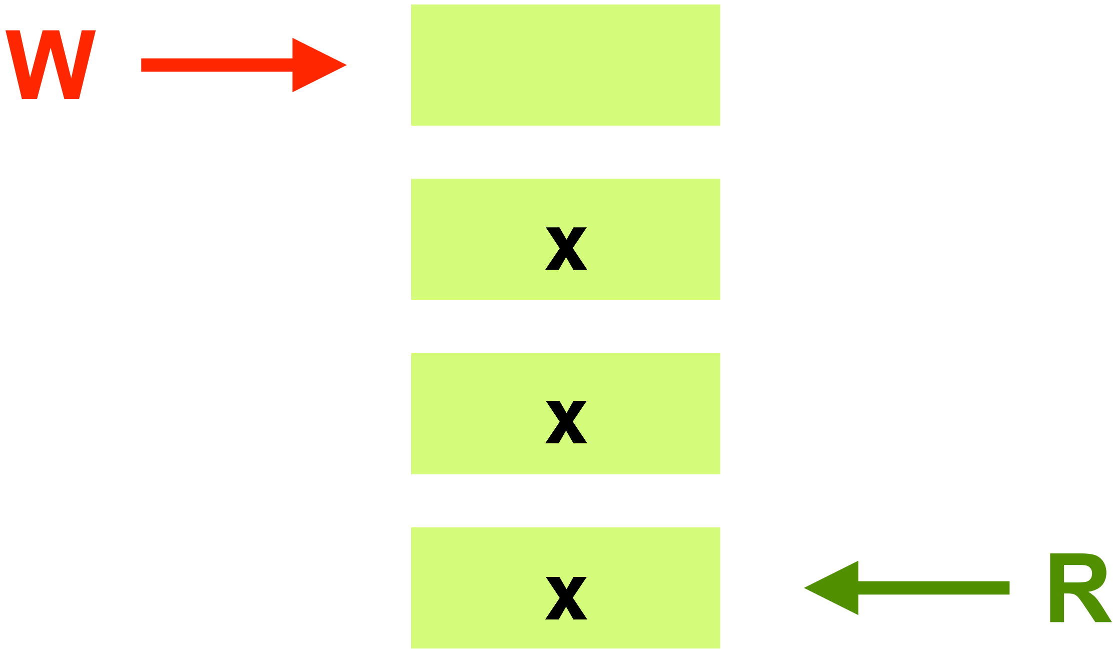
nothing to read!



queue is non-empty

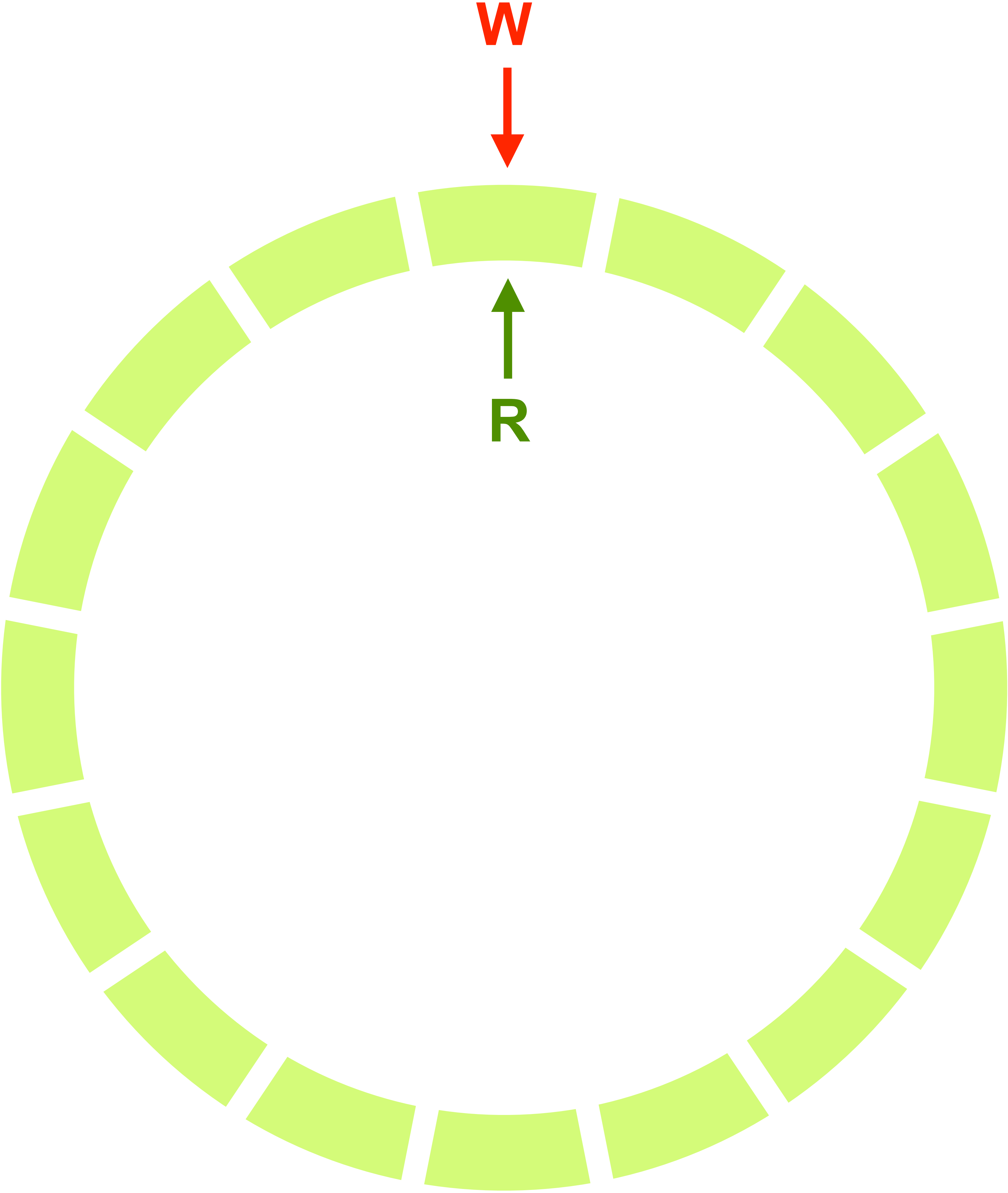
`W != R`

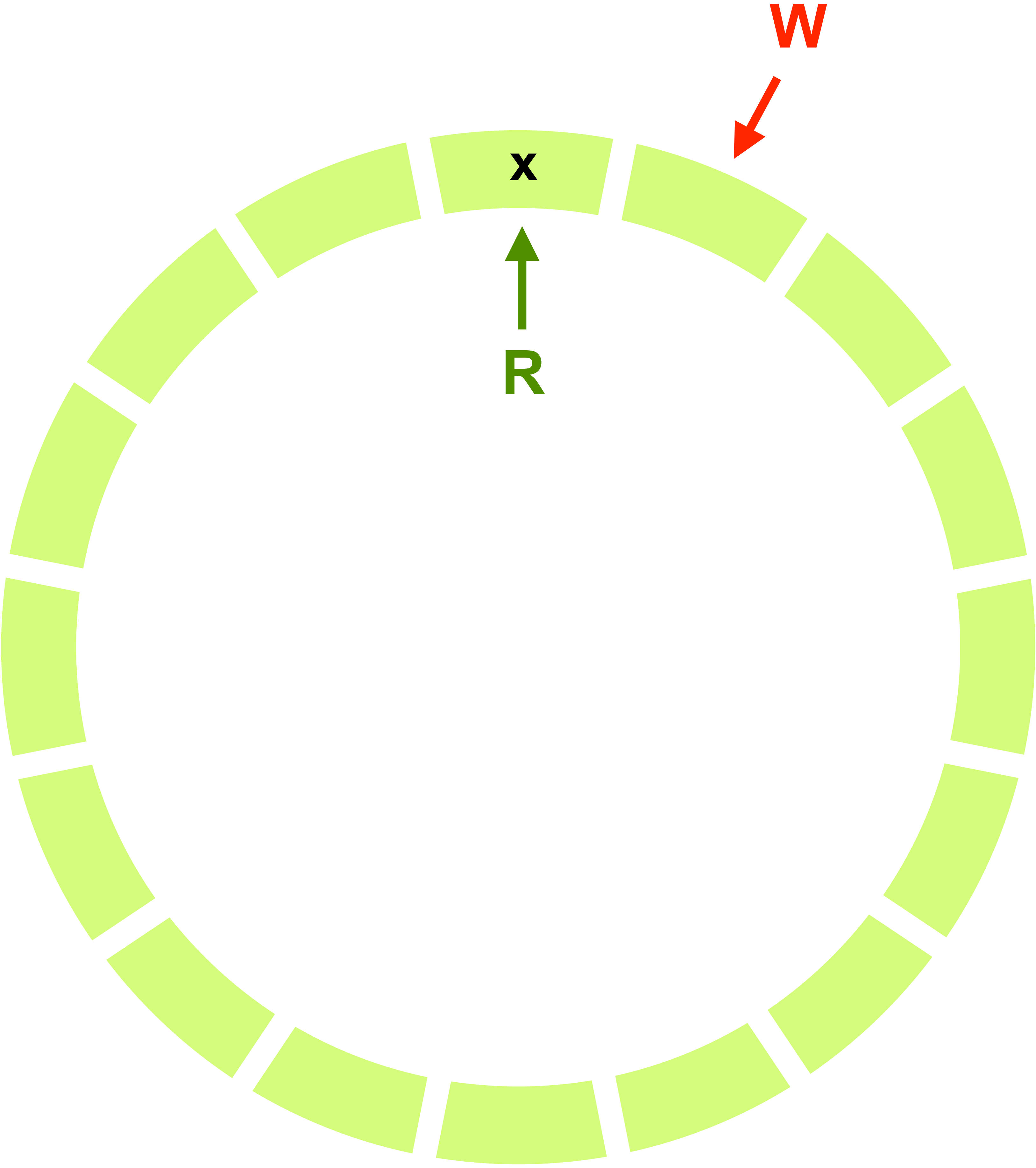
no data race!

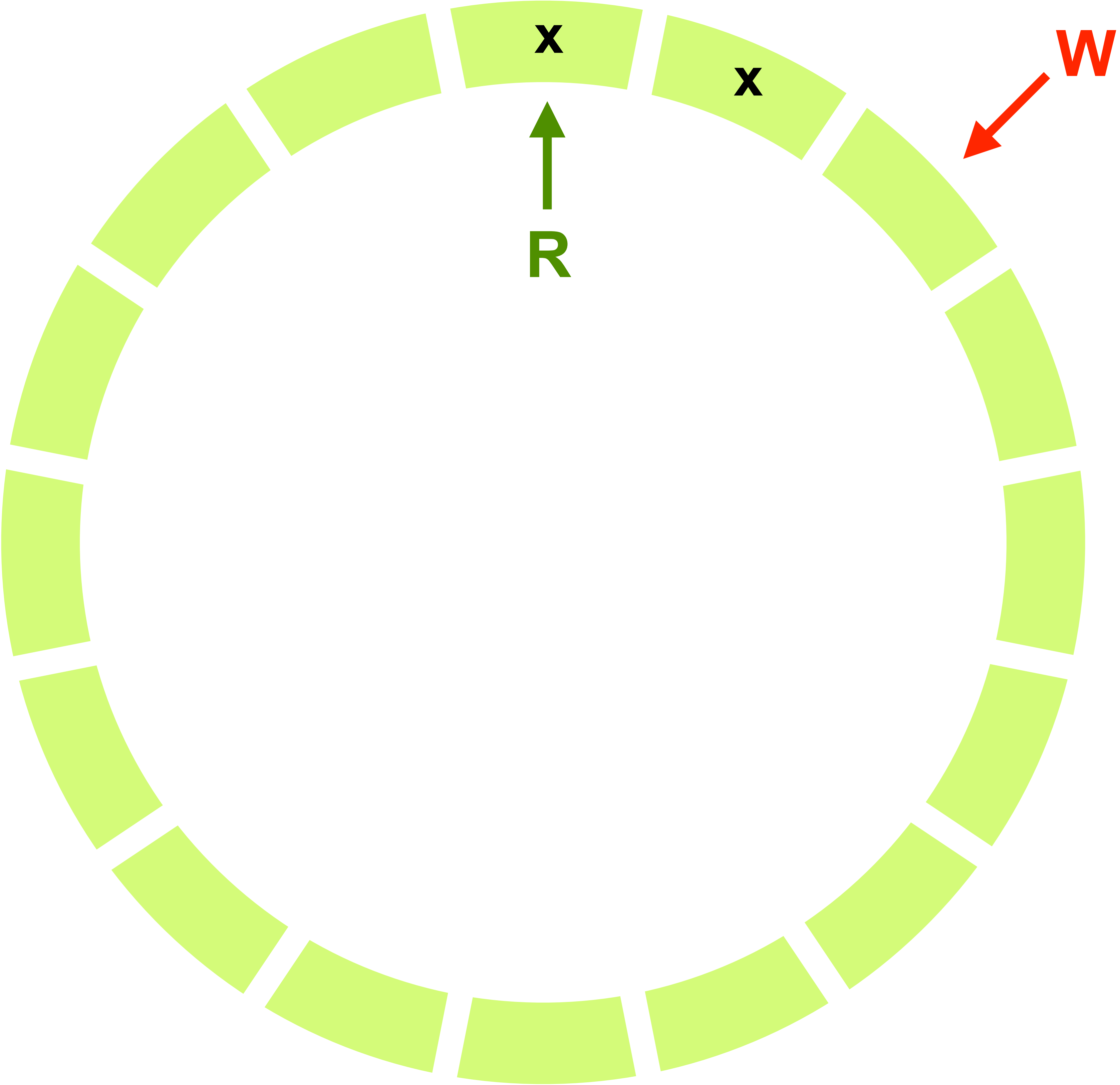


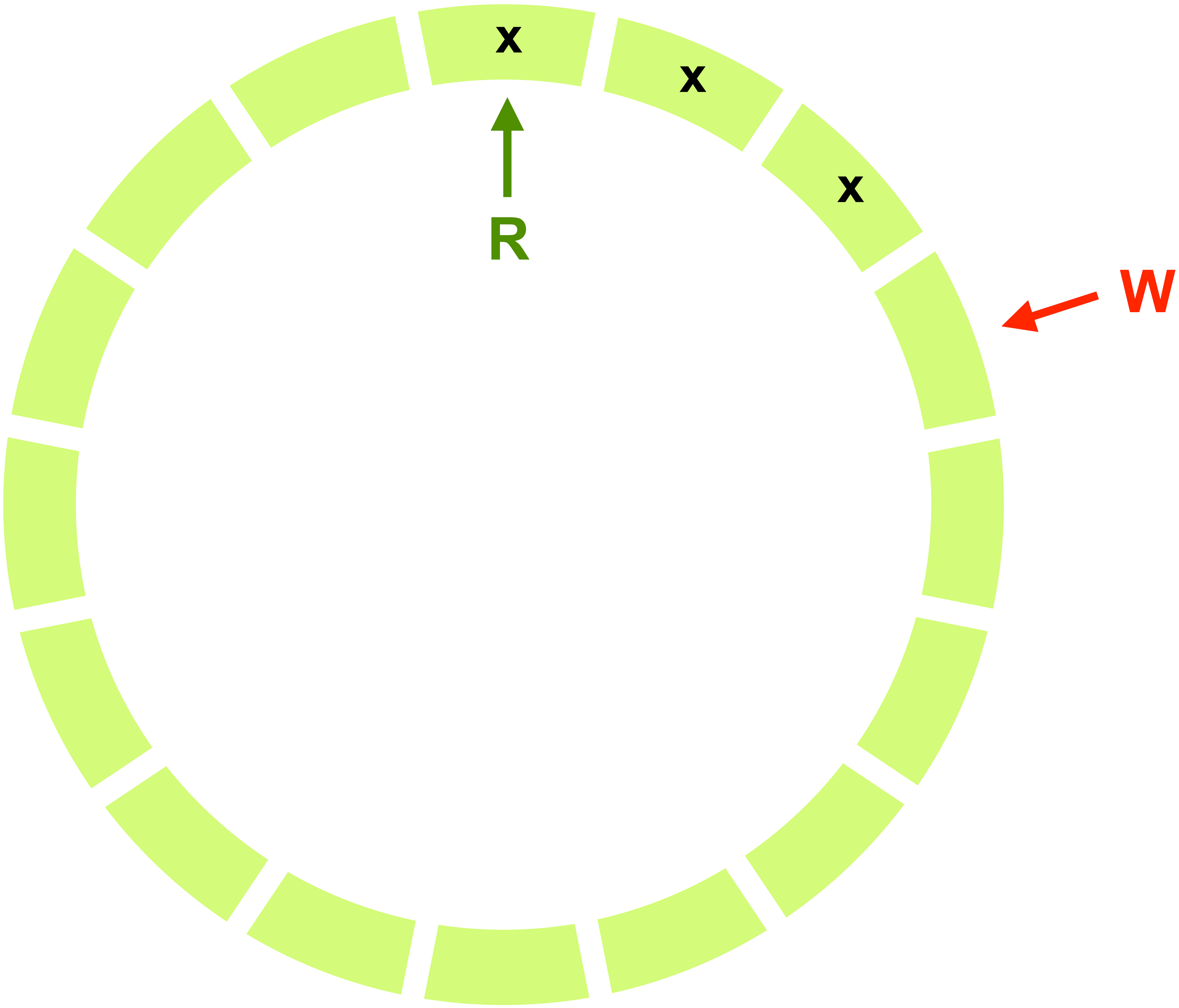
# no allocations

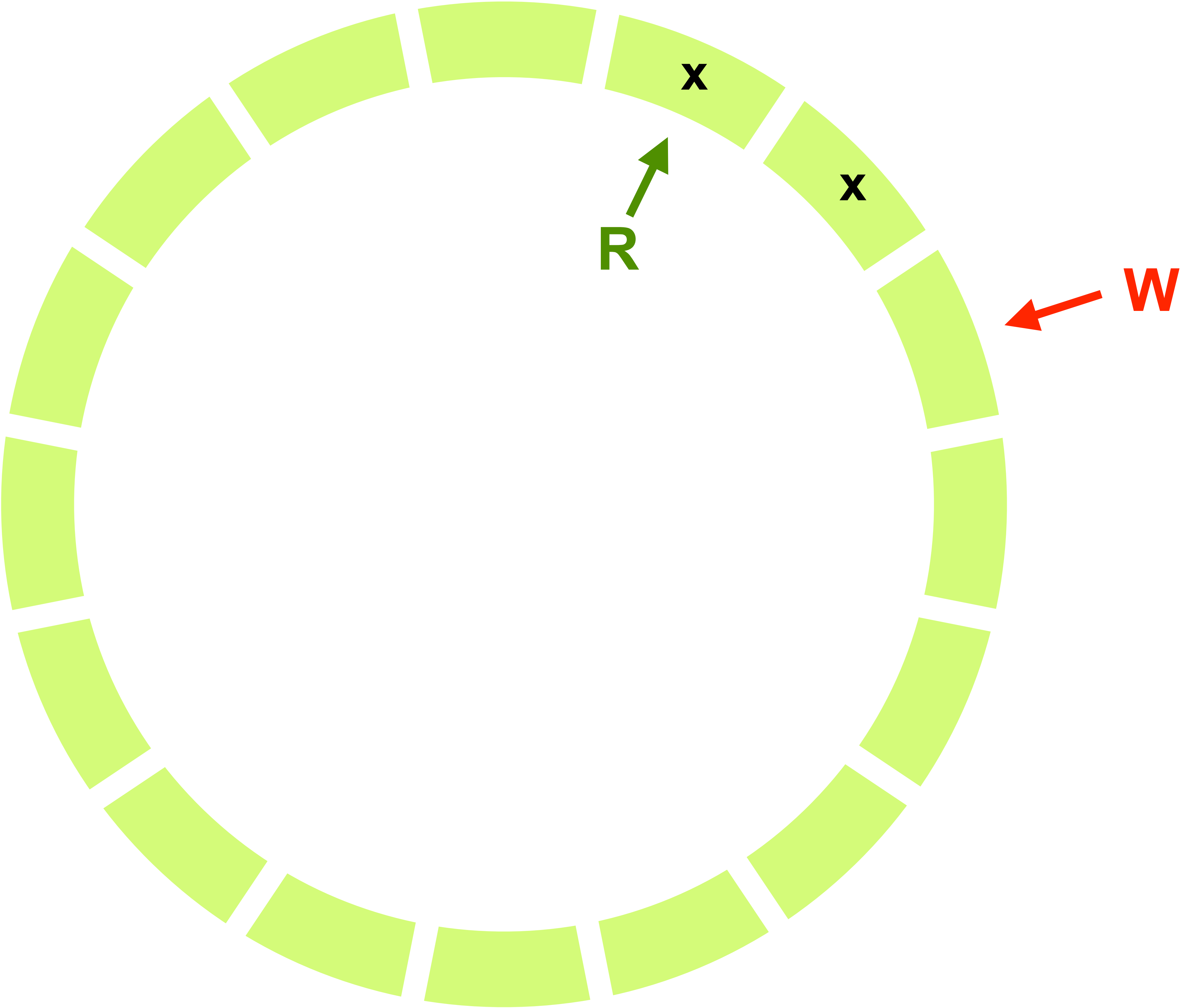
- limited capacity
- pre-allocated, fixed-size ring buffer for storage

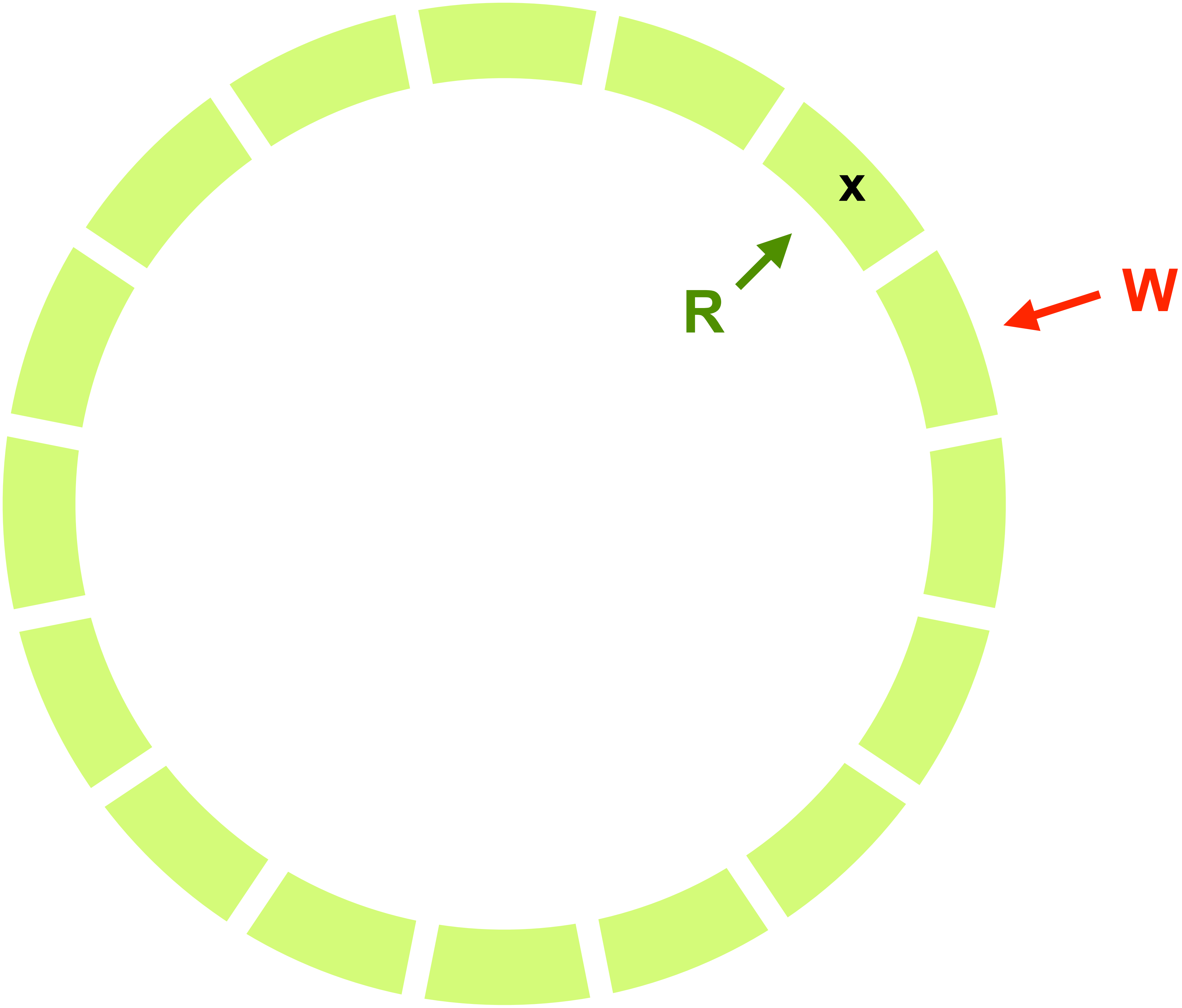

W
R


W
x
R

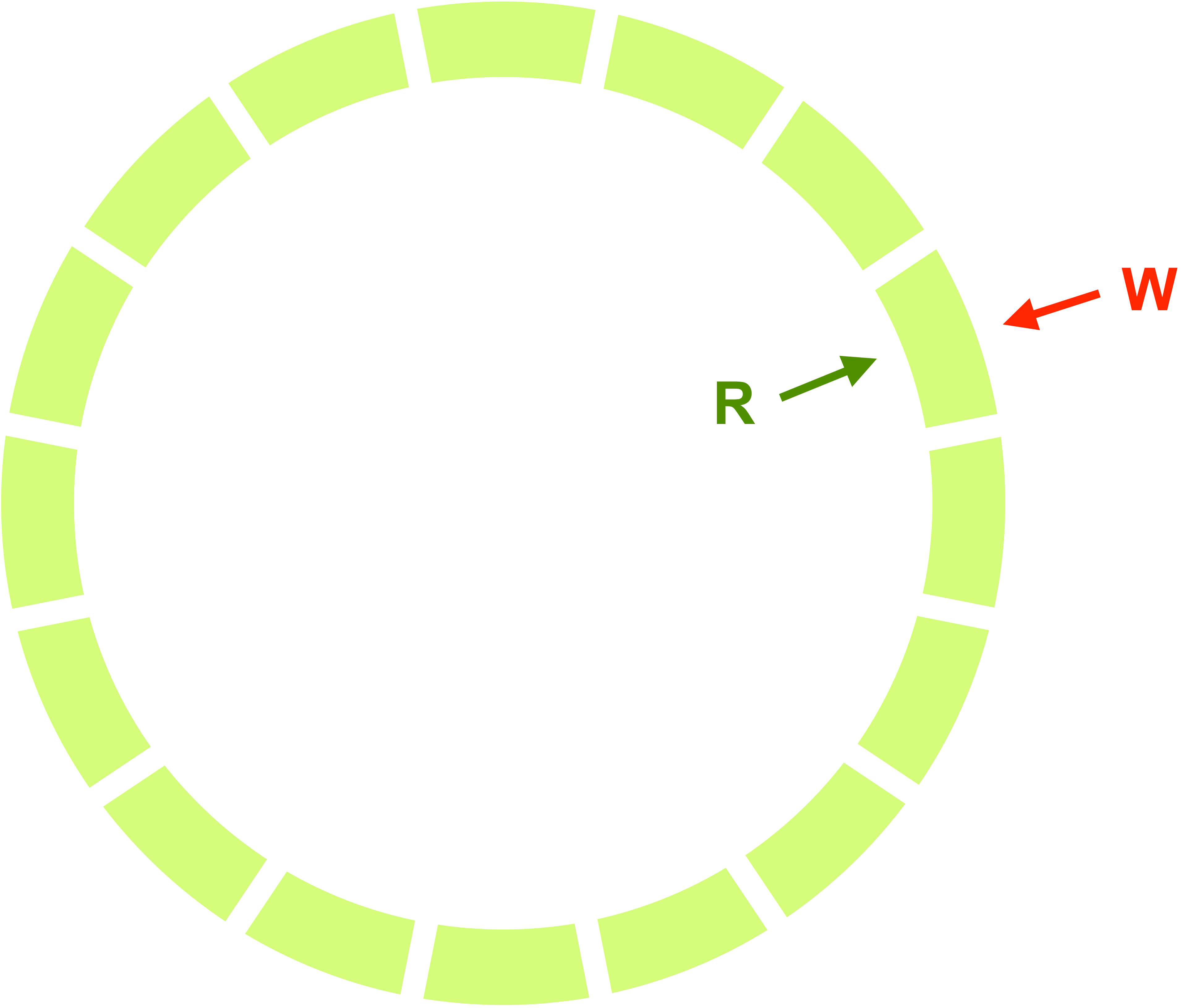

X
X
W
R


x
x
x
R
W

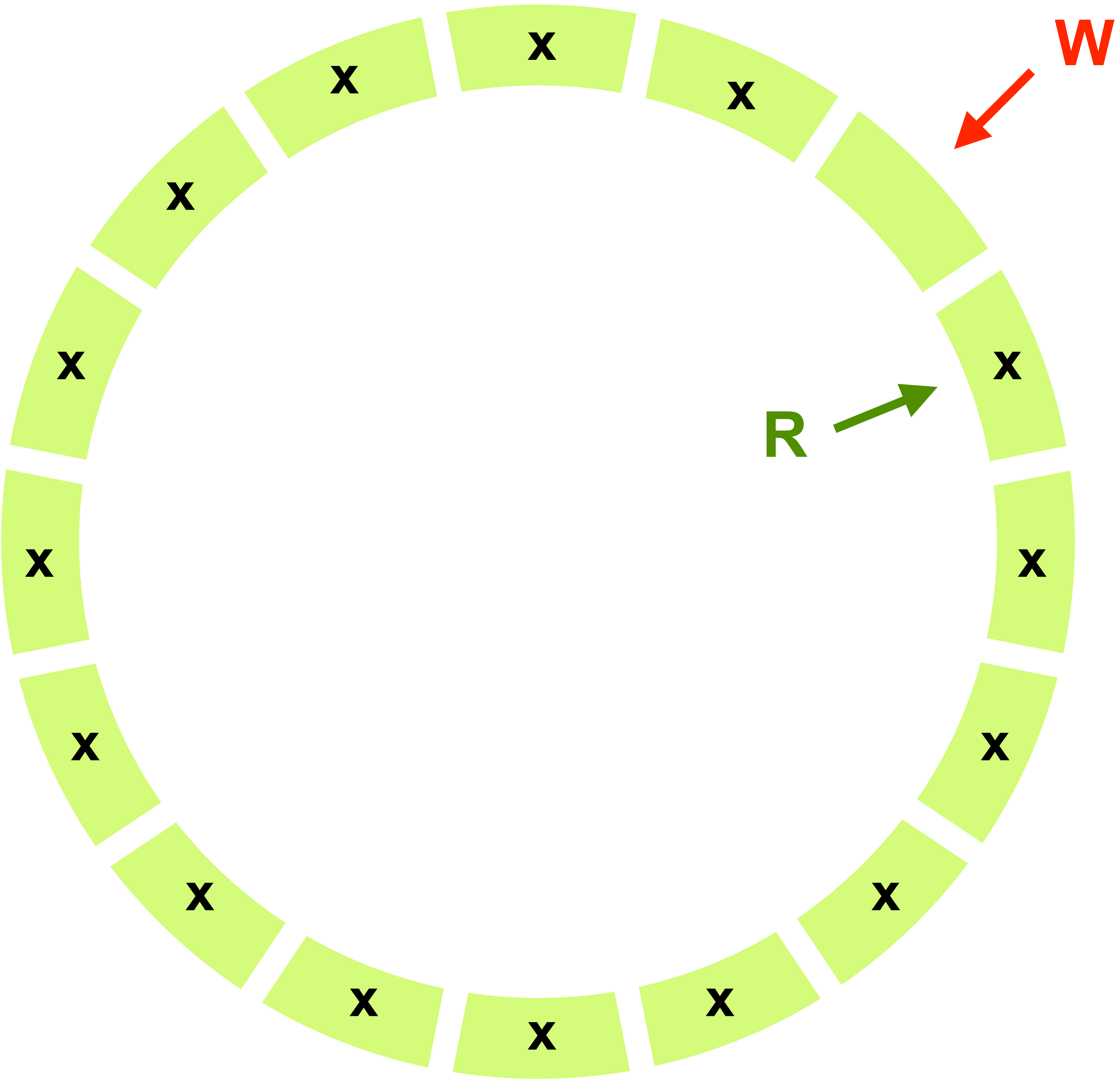

x
x
R
W


x
R
W

# queue empty



# queue full



```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        // TODO
    }
};
```

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        // TODO
    }

private:
    static constexpr size_t ringBufferSize = size + 1;
};
```

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        // TODO
    }

private:
    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
};
```

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        // TODO
    }

private:
    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = { 0 }, writePos = { 0 };
};
```

```cpp
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();
    }

private:
    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = { 0 }, writePos = { 0 };
};
```

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;
    }

private:
    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = { 0 }, writePos = { 0 };
};
```

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = std::move (ringBuffer[oldReadPos]);

        readPos.store (++oldReadPos));
        return true;
    }

private:
    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = { 0 }, writePos = { 0 };
};
```

```cpp
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        // TODO
    }

    bool pop (T& returnedElement)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = std::move (ringBuffer[oldReadPos]);

        readPos.store (getPositionAfter (oldReadPos));
        return true;
    }

private:
    static constexpr size_t getPositionAfter (size_t pos) noexcept
    {
        return ++pos == ringBufferSize ? 0 : pos;
    }

    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = { 0 }, writePos = { 0 };
};
```

```cpp
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        auto oldWritePos = writePos.load();
        auto newWritePos = getPositionAfter (oldWritePos);

        if (newWritePos == readPos.load())
            return false;
    }

    bool pop (T& returnedElement)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = ringBuffer[oldReadPos];

        readPos.store (getPositionAfter (oldReadPos));
        return true;
    }

private:
    static constexpr size_t getPositionAfter (size_t pos) noexcept
    {
        return ++pos == ringBufferSize ? 0 : pos;
    }

    static constexpr size_t ringBufferSize = size + 1;
    std::array<T, ringBufferSize> ringBuffer;
    std::atomic<size_t> readPos = { 0 }, writePos = { 0 };
};
```

```cpp
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement)
    {
        auto oldWritePos = writePos.load();
        auto newWritePos = getPositionAfter (oldWritePos);

        if (newWritePos == readPos.load())
            return false;

        ringBuffer[oldWritePos] = newElement;

        writePos.store (newWritePos);
        return true;
    }

    bool pop (T& returnedElement)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = std::move (ringBuffer[oldReadPos]);

        readPos.store (getPositionAfter (oldReadPos));
        return true;
    }

private:
    static constexpr size_t getPositionAfter (size_t pos) noexcept
    {
        return ++pos == ringBufferSize ? 0 : pos;
    }
```

✓

✓ lock-free

✓ wait-free

easy

# possible additions

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement);
    bool pop (T& returnedElement);
};
```

# possible additions

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement);
    bool push (T&& newElement);
    bool pop (T& returnedElement);
};
```

# possible additions

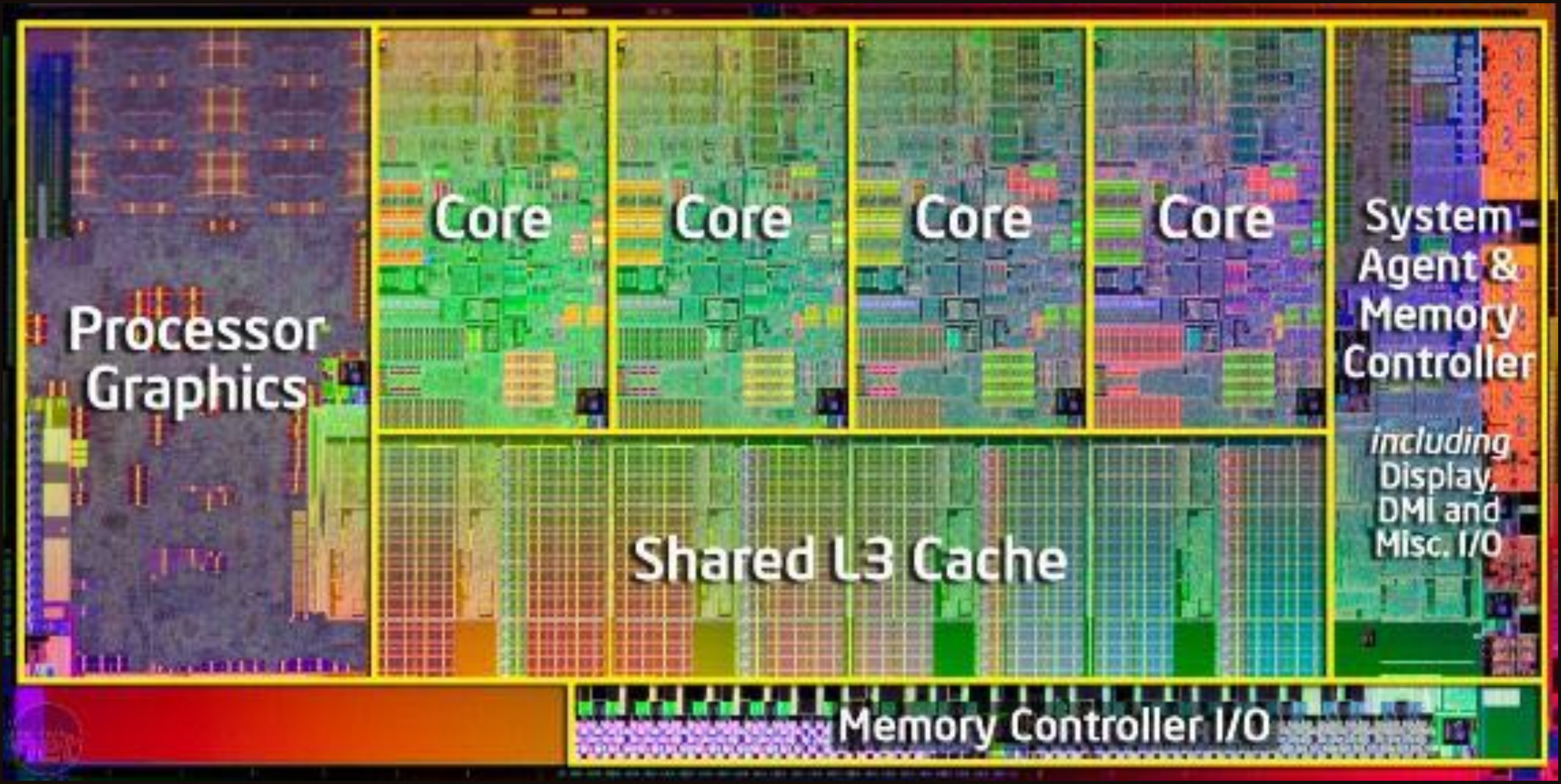
```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement);
    bool push (T&& newElement);
    bool pop (T& returnedElement);

    size_t size() const noexcept;
    void clear();
};
```

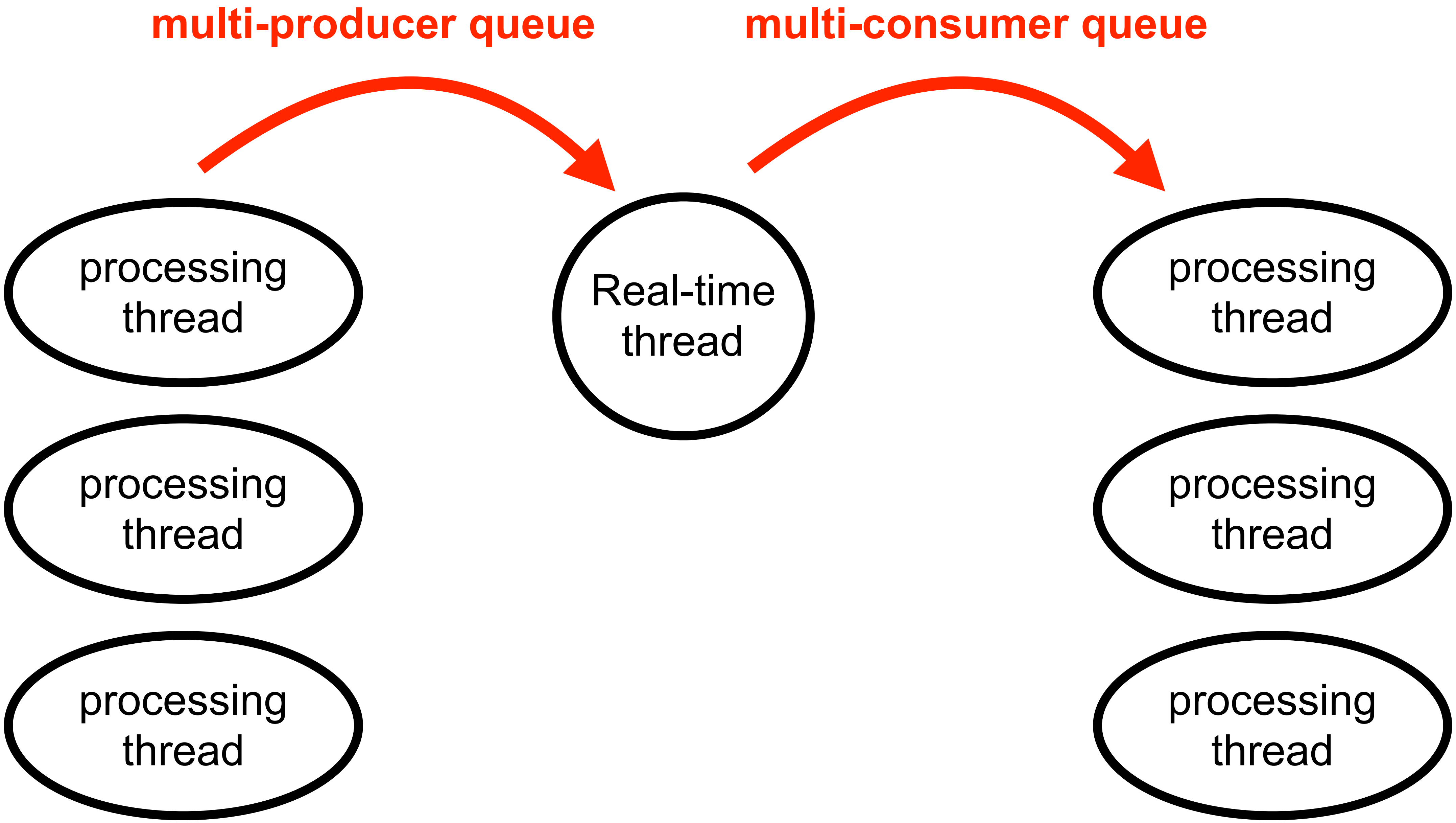
# possible additions

```
template <typename T, size_t size>
class LockFreeQueue
{
public:
    bool push (const T& newElement);
    bool push (T&& newElement);
    bool pop (T& returnedElement);

    size_t size() const noexcept;
    void clear();

    void pushRange (InputIterator* first, InputIterator* last);
    void pushElements (InputIterator* first, size_t numElements);
    size_t popAll (OutputIterator* iter);
    size_t popElements (OutputIterator* iter, size_t numElements);
};
```

SPSC

LEVEL COMPLETED!


Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including Display, DMI and Misc. I/O
Shared L3 Cache
Memory Controller I/O



multi-producer queue

multi-consumer queue

processing thread

processing thread

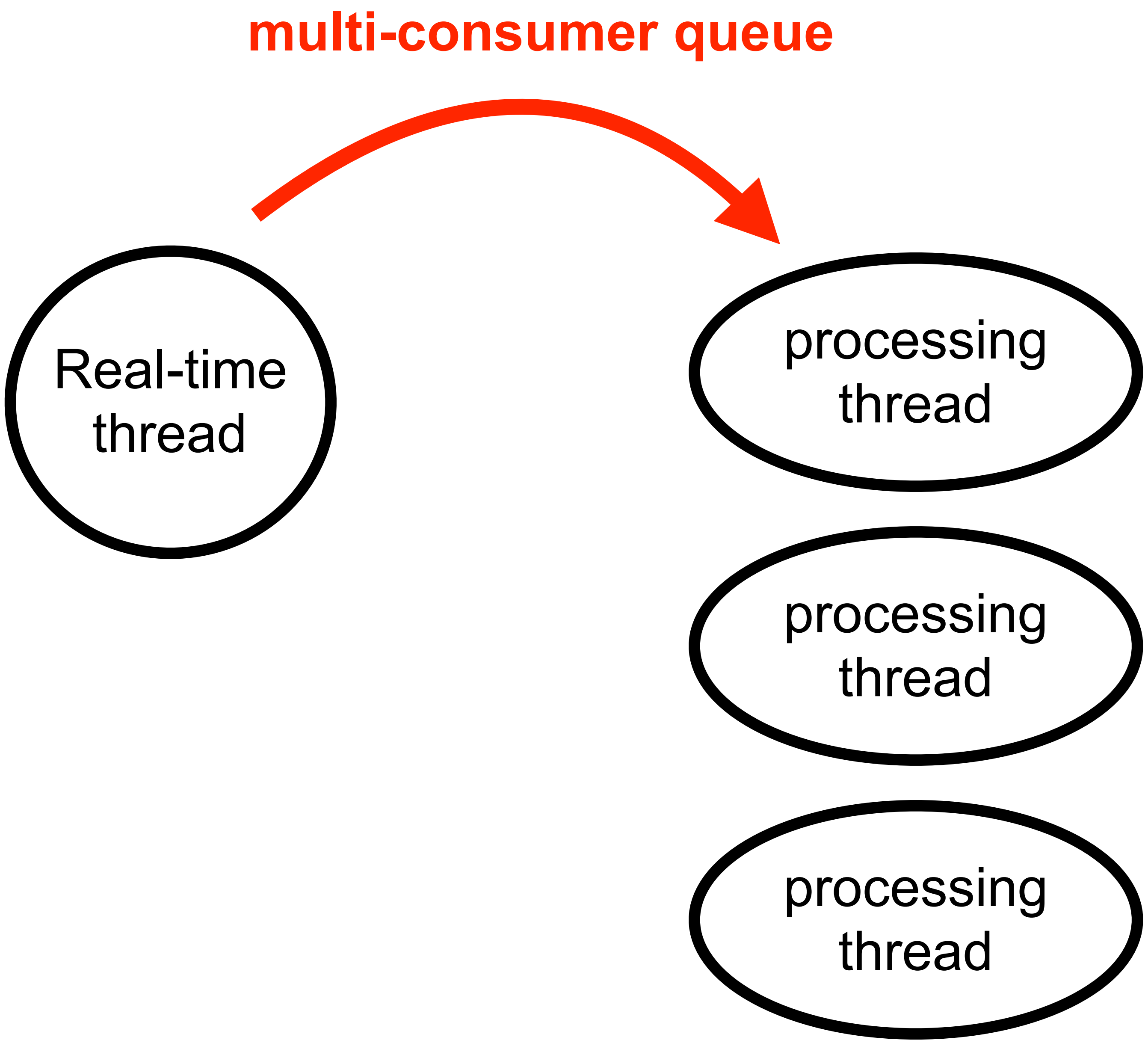
processing thread

Real-time thread

processing thread

processing thread

processing thread



multi-consumer queue

Real-time thread

processing thread

processing thread

processing thread

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos] = newElement;

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    returnedElement = std::move (ringBuffer[oldReadPos]);

    readPos.store (getPositionAfter (oldReadPos));
    return true;
}
```

**someone else might be reading it at the same time…**

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos] = newElement;

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    returnedElement = ringBuffer[oldReadPos];

    readPos.store (getPositionAfter (oldReadPos));
    return true;
}
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos] = newElement;

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    returnedElement = ringBuffer[oldReadPos];

    if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
        return true;
}
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos] = newElement;

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    returnedElement = ringBuffer[oldReadPos];

    if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
        return true;

    oldReadPos = readPos.load();
}
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos] = newElement;          writer

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos];          reader 1

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();                          reader 2
    }
}
```

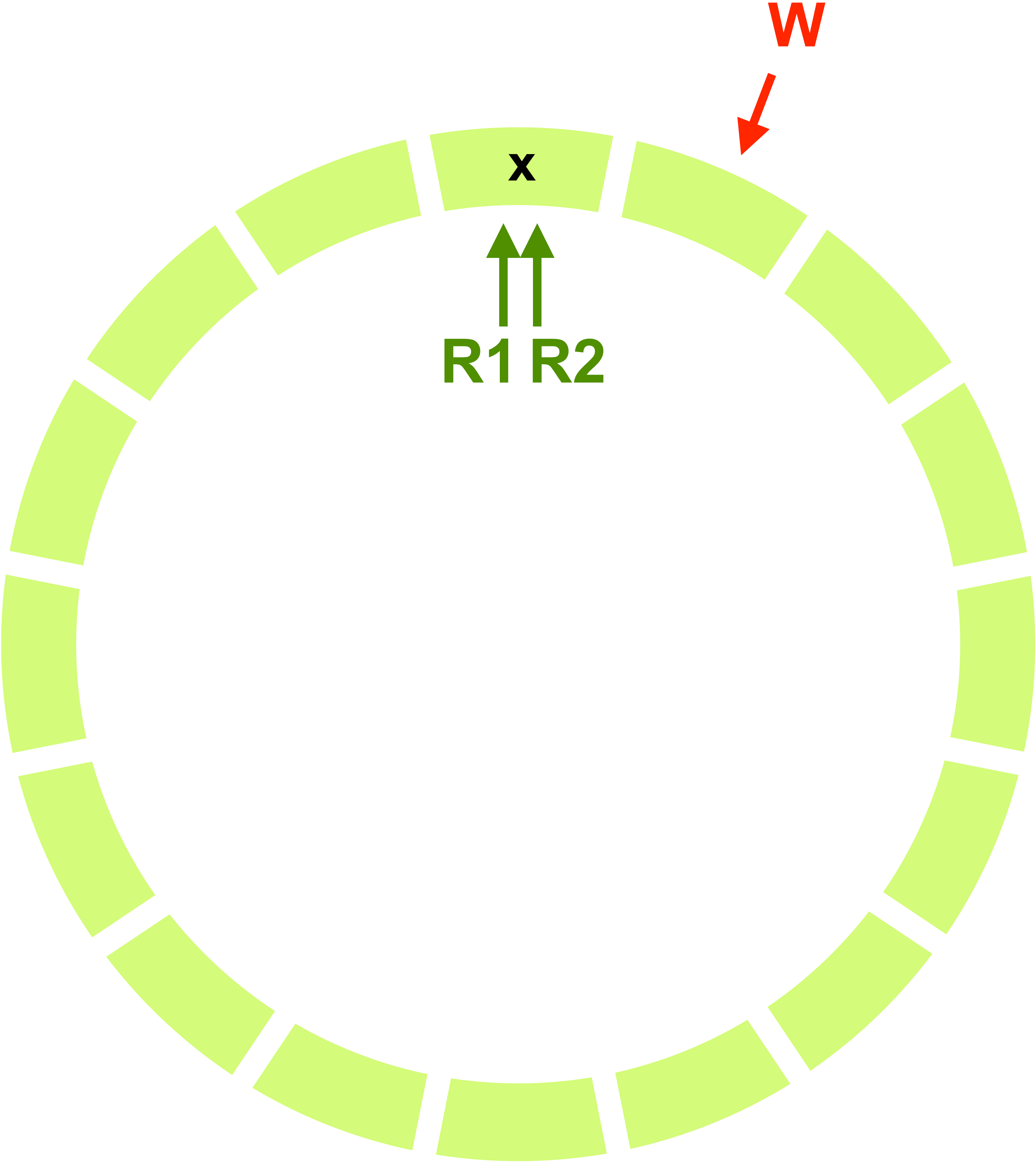
```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos] = newElement;          writer

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos];    reader 1

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;
                                                           reader 2
        oldReadPos = readPos.load();
    }
}
```

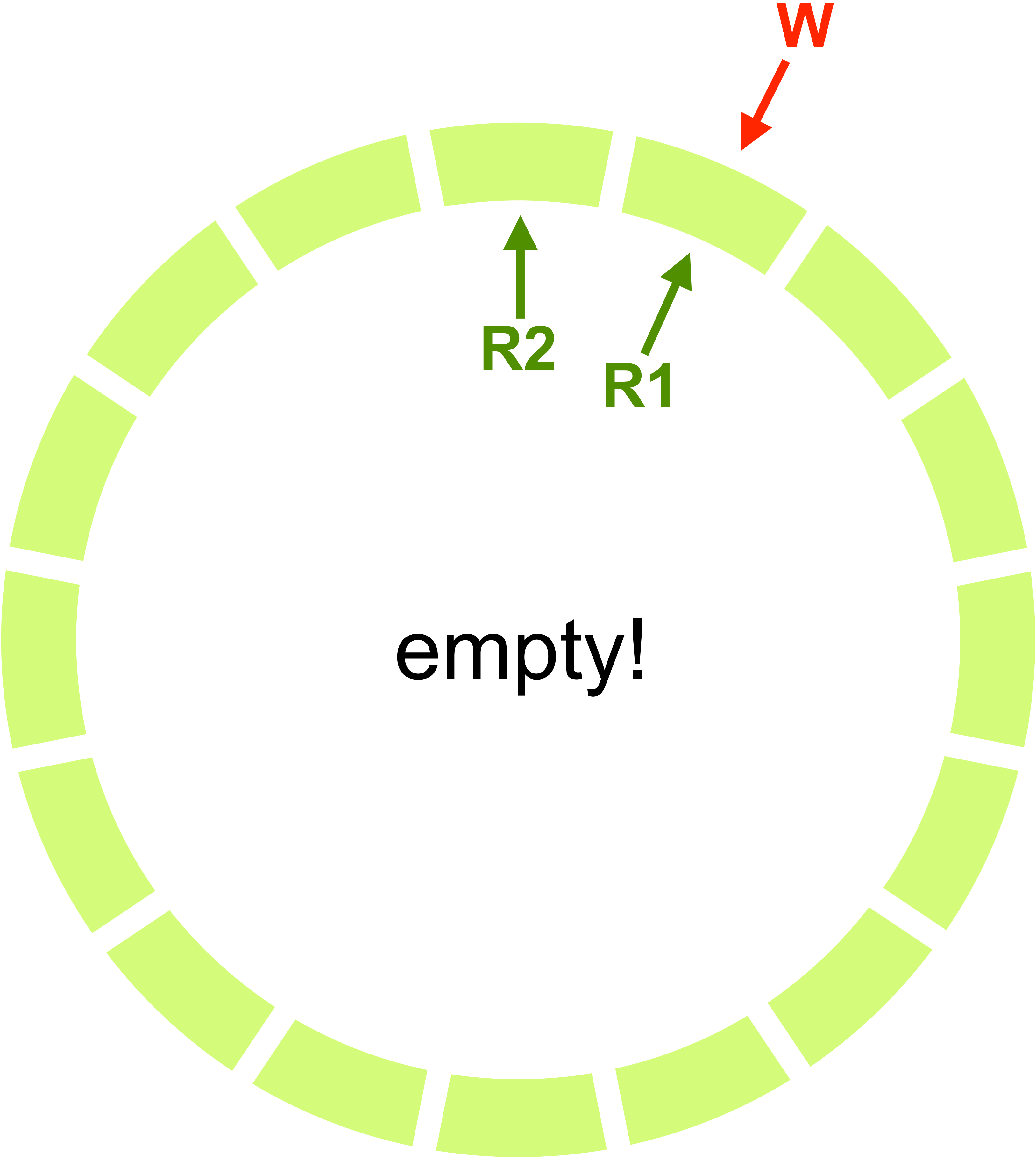
~~`std::array<T, ringBufferSize> ringBuffer;`~~

```
std::array<std::atomic<T>, ringBufferSize> ringBuffer;
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```


W
x
R1
R2

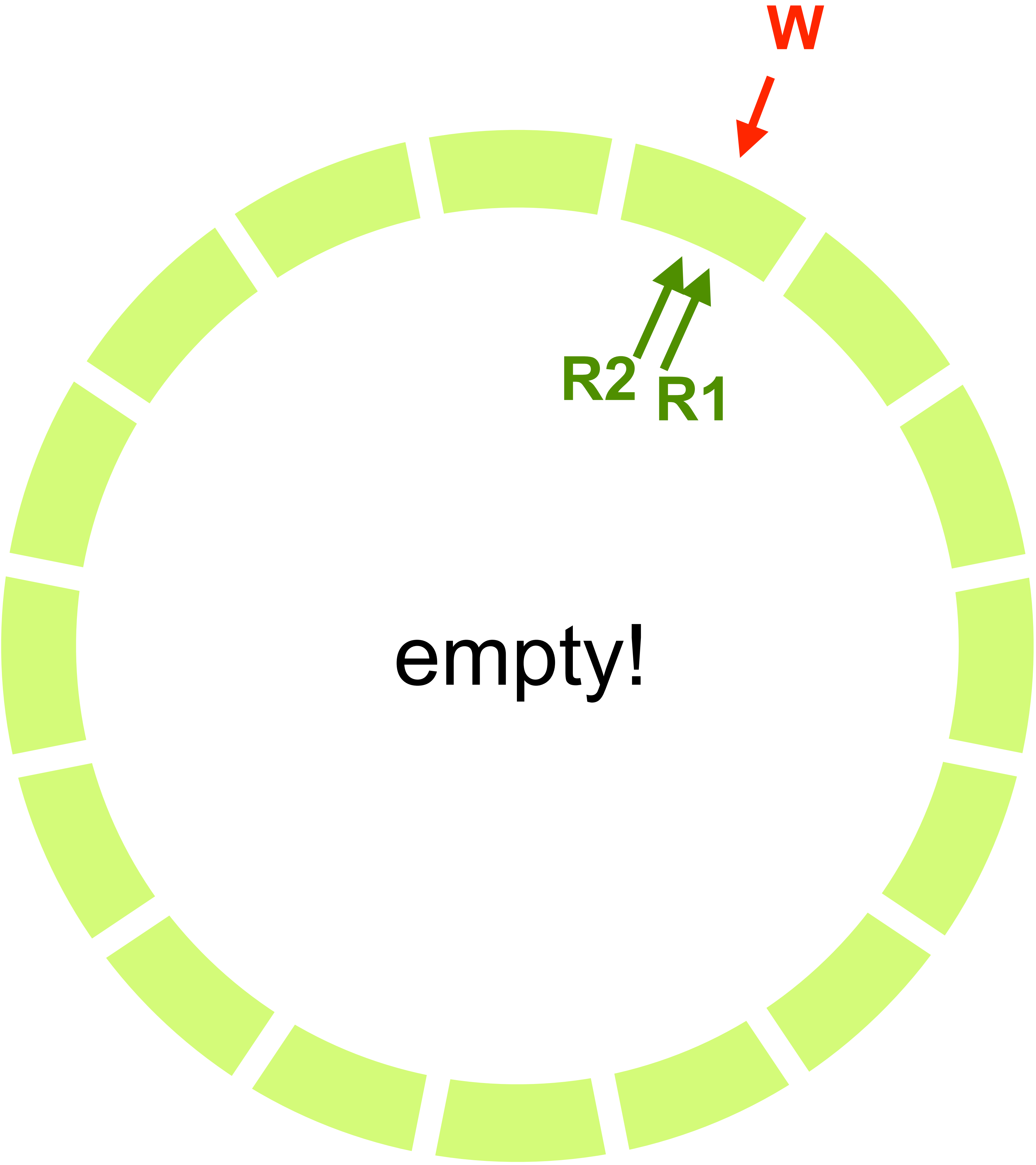

W
R2
R1
empty!

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}
```

writer

```
bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```

reader1, reader2



empty!

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```

**reader2**

**(queue is empty!)**

```cpp
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    auto oldWritePos = writePos.load();
    auto oldReadPos = readPos.load();

    if (oldWritePos == oldReadPos)
        return false;

    while (true)
    {
        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    while (true)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    while (true)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;

        oldReadPos = readPos.load();
    }
}
```

✓

```
bool push (const T& newElement)
{
    auto oldWritePos = writePos.load();
    auto newWritePos = getPositionAfter (oldWritePos);

    if (newWritePos == readPos.load())
        return false;

    ringBuffer[oldWritePos].store (newElement);

    writePos.store (newWritePos);
    return true;
}


bool pop (T& returnedElement)
{
    while (true)
    {
        auto oldWritePos = writePos.load();
        auto oldReadPos = readPos.load();

        if (oldWritePos == oldReadPos)
            return false;

        returnedElement = ringBuffer[oldReadPos].load();

        if (readPos.compare_exchange_strong (oldReadPos, getPositionAfter (oldReadPos)))
            return true;
    }
}
```

✓ lock-free

✗ not wait-free

# Thank you!