# A lock-free atomic shared_ptr

Version 0.1


Timur Doumler

@timur_audio



ACCU Conference
6 April 2022


Milky Way above Lava Flow Trail at
Coconino National Forest, CA, USA
Image by Deborah Lee Soltesz

# lock-free atomic shared_ptr

# lock-free atomic shared_ptr

# 20 Memory management library [mem]

## 20.3 Smart pointers [smartptr]

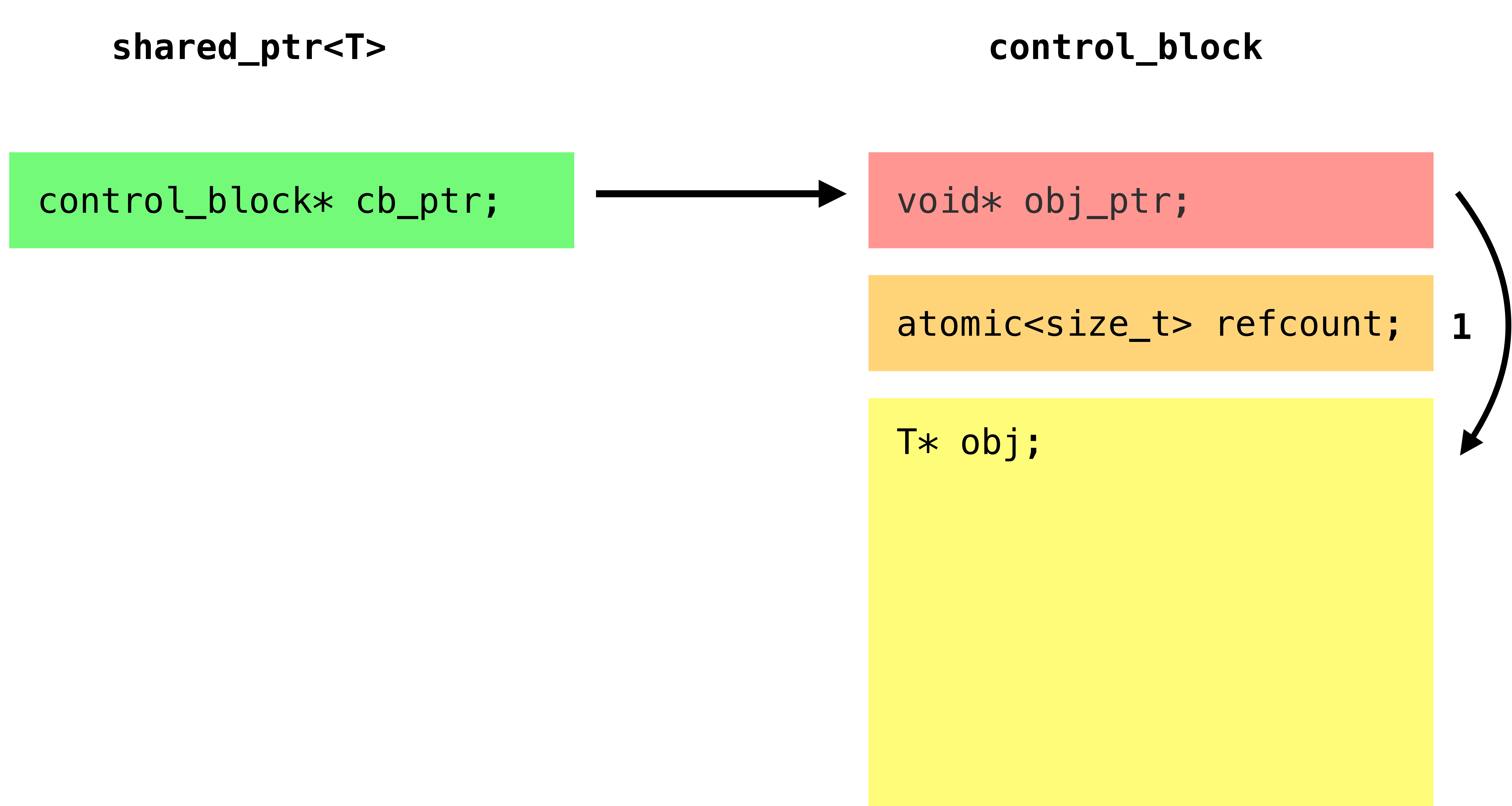
### 20.3.2 Shared-ownership pointers [util.sharedptr]

#### 20.3.2.2 Class template `shared_ptr` [util.smartptr.shared]

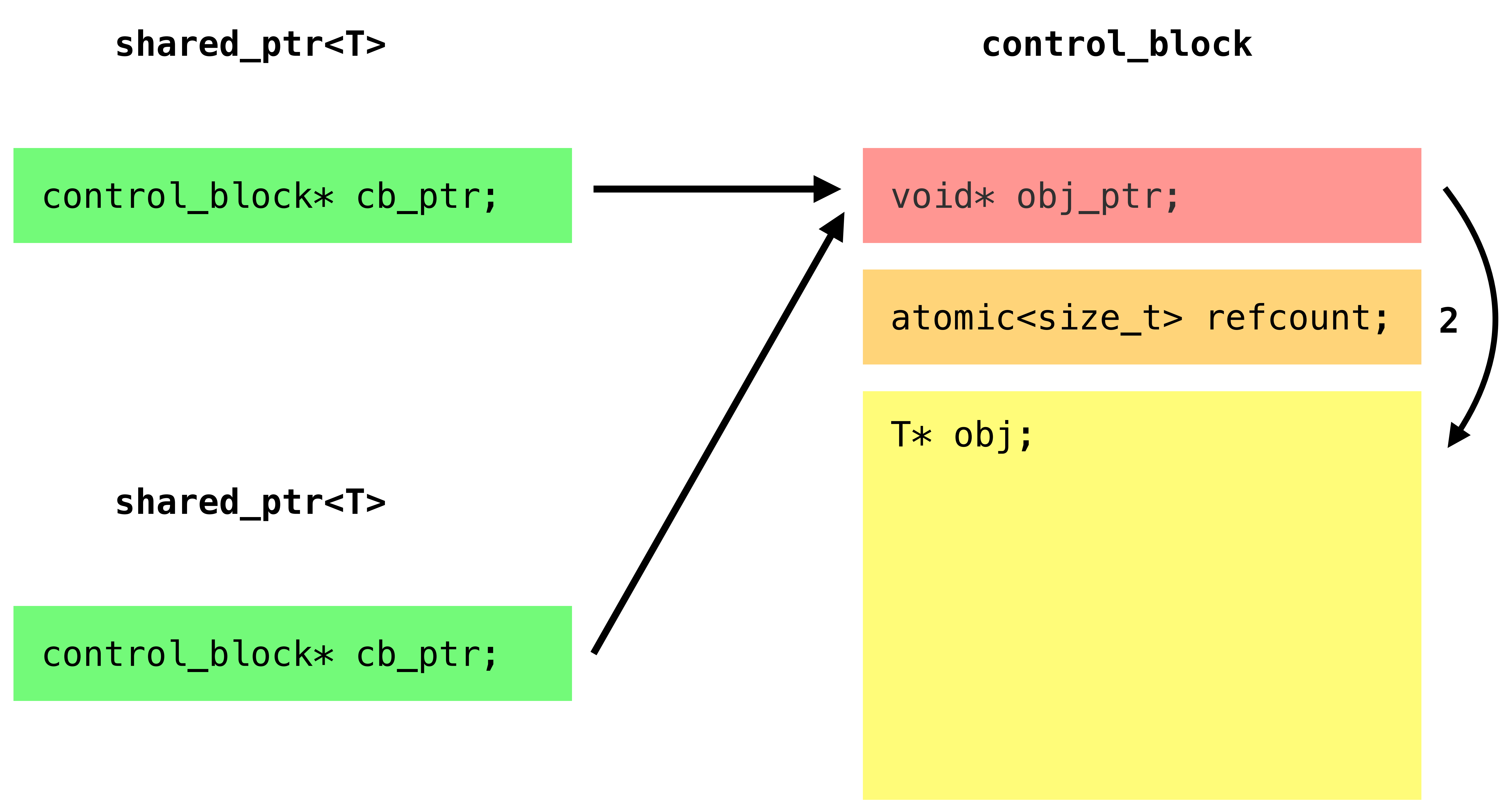
##### 20.3.2.2.1 General [util.smartptr.shared.general]

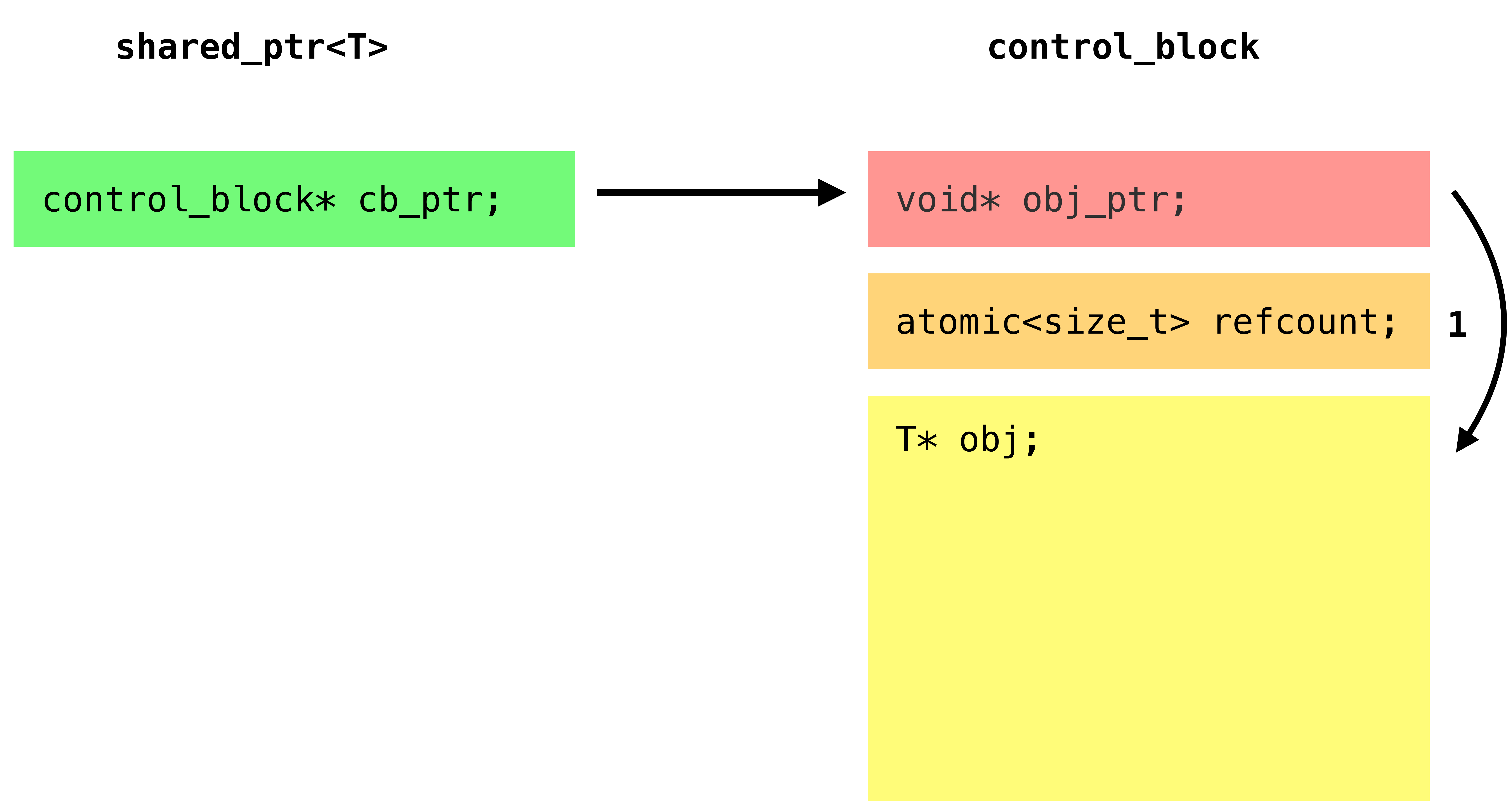
[1] The `shared_ptr` class template stores a pointer, usually obtained via `new`. `shared_ptr` implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A `shared_ptr` is said to be empty if it does not own a pointer.

```
namespace std {
  template<class T> class shared_ptr {
```



shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
1
T* obj;


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
2
T* obj;
shared_ptr<T>
control_block* cb_ptr;




Copyright (c) Timur Doumler 

**control_block**

```
void* obj_ptr;

atomic<size_t> refcount;   0

T* obj;
```

**control_block**

```
void* obj_ptr;
```

```
atomic<size_t> refcount;
```
**0**

# lock-free **atomic** shared_ptr

lock-free **atomic** shared_ptr

**indivisible**

**lock-free atomic shared_ptr**

**indivisible**

**race-free**

**lock-free atomic shared_ptr**

**indivisible**

**race-free**

**thread-safe**

Copyright (c) Timur Doumler | @timur_audio | https://timur.audio

[2] Specializations of `shared_ptr` shall be *Cpp17CopyConstructible*, *Cpp17CopyAssignable*, and *Cpp17LessThanComparable*, allowing their use in standard containers. Specializations of `shared_ptr` shall be contextually convertible to `bool`, allowing their use in boolean expressions and declarations in conditions.

[3] The template parameter `T` of `shared_ptr` may be an incomplete type.

[*Note 1*: `T` can be a function type. — *end note*]

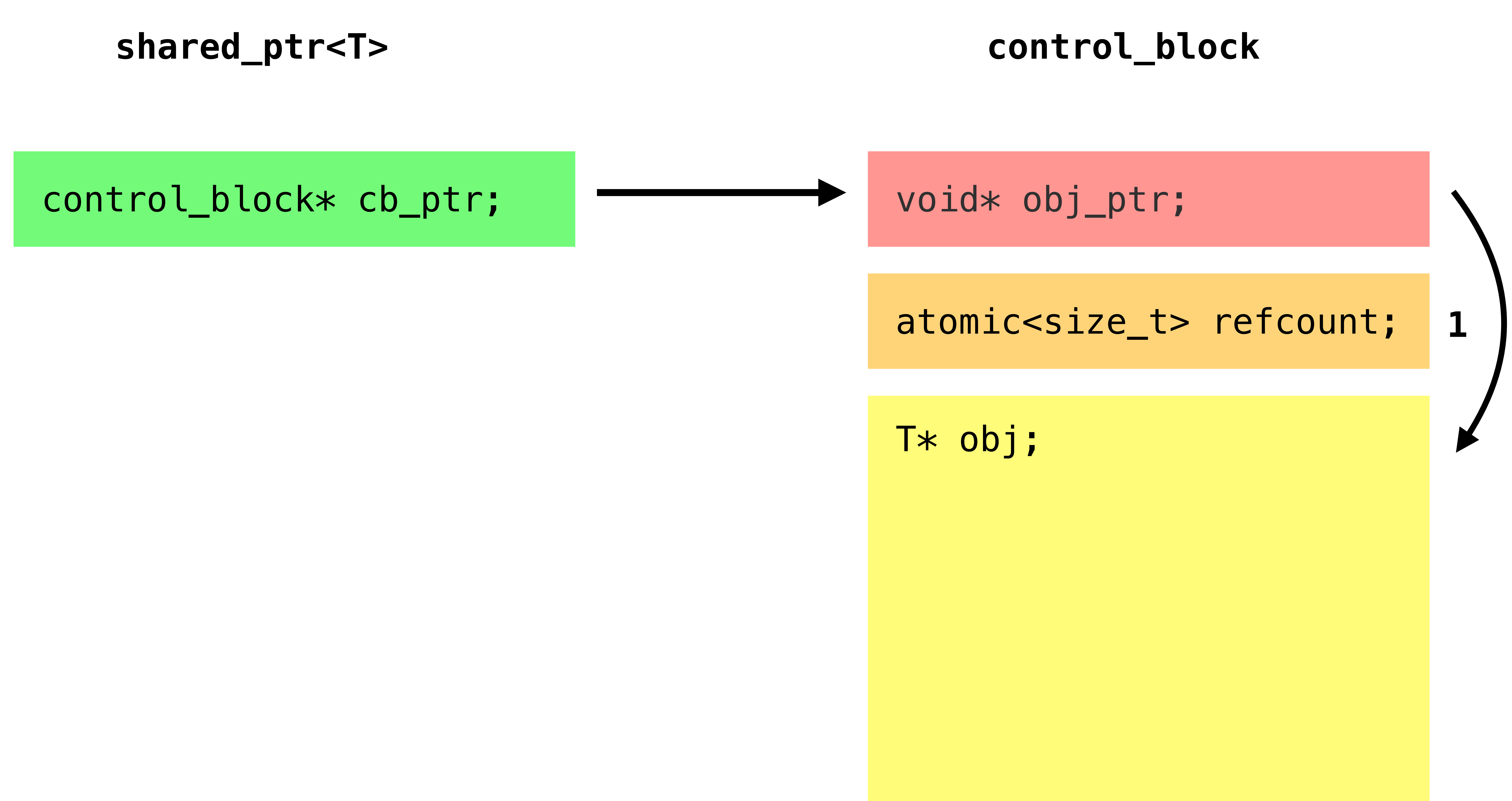
[4] [*Example 1*:

```
if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```

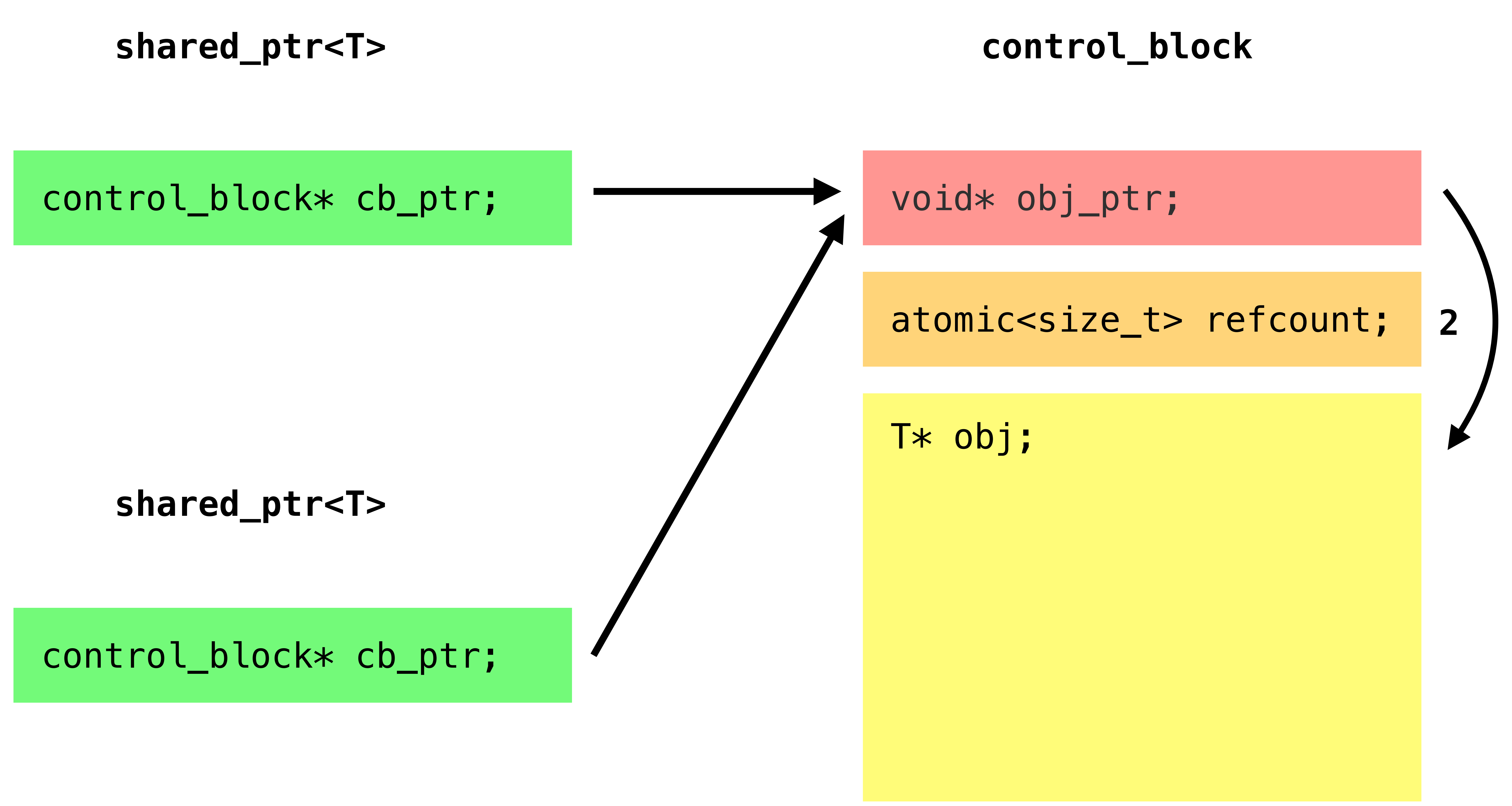
— *end example*]

[5] For purposes of determining the presence of a data race, member functions shall access and modify only the `shared_ptr` and `weak_ptr` objects themselves and not objects they refer to. Changes in `use_count()` do not reflect modifications that can introduce data races.

[6] For the purposes of subclause [smartptr], a pointer type `Y*` is said to be *compatible with* a pointer type `T*` when either `Y*` is convertible to `T*` or `Y` is `U[N]` and `T` is *cv* `U[]`.



shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
1
T* obj;


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
2
T* obj;
shared_ptr<T>
control_block* cb_ptr;

```cpp
auto ptr = std::make_shared<widget>();

// Thread 1
auto ptr1 = ptr;

// Thread 2
auto ptr2 = ptr;



// This is fine :)
```

```
auto ptr = std::make_shared<widget>();

// Thread 1
auto ptr1 = ptr;

// Thread 2
ptr = ptr2;
```

```
auto ptr = std::make_shared<widget>();

// Thread 1
auto ptr1 = ptr;

// Thread 2
ptr = ptr2;



// This is undefined behaviour :(
```

# std::atomic_...<std::shared_ptr>

| Declaration | | Version |
|---|---|---|
| `template< class T >`<br>`bool atomic_is_lock_free( const std::shared_ptr<T>* p );` | (1) | (since C++11)<br>(deprecated in C++20) |
| `template< class T >`<br>`std::shared_ptr<T> atomic_load( const std::shared_ptr<T>* p );` | (2) | (since C++11)<br>(deprecated in C++20) |
| `template< class T >`<br>`std::shared_ptr<T> atomic_load_explicit( const std::shared_ptr<T>* p,`<br>`std::memory_order mo );` | (3) | (since C++11)<br>(deprecated in C++20) |
| `template< class T >`<br>`void atomic_store( std::shared_ptr<T>* p,`<br>`std::shared_ptr<T> r );` | (4) | (since C++11)<br>(deprecated in C++20) |
| `template< class T >`<br>`void atomic_store_explicit( std::shared_ptr<T>* p,`<br>`std::shared_ptr<T> r,`<br>`std::memory_order mo);` | (5) | (since C++11)<br>(deprecated in C++20) |
| `template< class T >`<br>`std::shared_ptr<T> atomic_exchange( std::shared_ptr<T>* p,`<br>`std::shared_ptr<T> r);` | (6) | (since C++11)<br>(deprecated in C++20) |
| `template<class T>`<br>`std::shared_ptr<T> atomic_exchange_explicit( std::shared_ptr<T>* p,`<br>`std::shared_ptr<T> r,`<br>`std::memory_order mo);` | (7) | (since C++11)<br>(deprecated in C++20) |

```
template< class T >
bool atomic_compare_exchange_weak( std::shared_ptr<T>* p,
                                   std::shared_ptr<T>* expected,
                                   std::shared_ptr<T> desired);
```
(8) (since C++11) (deprecated in C++20)

```
template<class T>
bool atomic_compare_exchange_strong( std::shared_ptr<T>* p,
                                     std::shared_ptr<T>* expected,
                                     std::shared_ptr<T> desired);
```
(9) (since C++11) (deprecated in C++20)

```
template< class T >
bool atomic_compare_exchange_strong_explicit( std::shared_ptr<T>* p,
                                              std::shared_ptr<T>* expected,
                                              std::shared_ptr<T> desired,
                                              std::memory_order success,
                                              std::memory_order failure);
```
(10) (since C++11) (deprecated in C++20)

```
template< class T >
bool atomic_compare_exchange_weak_explicit( std::shared_ptr<T>* p,
                                            std::shared_ptr<T>* expected,
                                            std::shared_ptr<T> desired,
                                            std::memory_order success,
                                            std::memory_order failure);
```
(11) (since C++11) (deprecated in C++20)

If multiple threads of execution access the same `std::shared_ptr` object without synchronization and any of those accesses uses a non-const member function of `shared_ptr` then a data race will occur unless all such access is performed through these functions, which are overloads of the corresponding atomic access functions (`std::atomic_load`, `std::atomic_store`, etc.)

Note that the control block of a shared_ptr is thread-safe: different `std::shared_ptr` objects can be accessed using mutable operations, such as operator= or reset, simultaneously by multiple threads, even when these instances are copies, and share the same control block internally.

```cpp
auto ptr = std::make_shared<widget>();

// Thread 1
auto ptr1 = std::atomic_load(&ptr);

// Thread 2
std::atomic_store(&ptr, ptr2);



// OK (deprecated in C++20)
```

# Atomic Smart Pointers, rev. 1

Herb Sutter

| | |
|---|---|
| Document #: | N4162 |
| Date: | 2014-10-06 |
| Reply to: | Herb Sutter (hsutter@microsoft.com) |

This is a revision of N4058 to apply SG1 feedback in Redmond to rename `atomic<*_ptr<T>>` to `atomic_*_ptr<T>`, require default initialization to null, and add proposed wording.

## Contents

1. Motivation ........................................ 2
   1.1. Problem ........................................ 2
   1.2. Motivating example for `atomic_unique_ptr<T>`: Producer-consumer handoff ........................ 2
   1.3. Motivating example for `atomic_shared_ptr<T>`: ABA + robustness + efficiency ........................ 3
   1.4. Motivating example for `atomic_weak_ptr<T>`: Swinging a `weak_ptr` ........................ 4

# std::atomic<std::shared_ptr>

Defined in header `<memory>`

```
template <class T> struct std::atomic<std::shared_ptr<T>>;
```
(since C++20)

The partial template specialization of `std::atomic` for `std::shared_ptr<T>` allows users to manipulate shared_ptr objects atomically.

If multiple threads of execution access the same `std::shared_ptr` object without synchronization and any of those accesses uses a non-const member function of `shared_ptr` then a data race will occur unless all such access is performed through an instance of `std::atomic<std::shared_ptr>` (or, deprecated as of C++20, through the standalone functions for atomic access to `std::shared_ptr`).

Associated `use_count` increments are guaranteed to be part of the atomic operation. Associated `use_count` decrements are sequenced after the atomic operation, but are not required to be part of it, except for the `use_count` change when overriding `expected` in a failed CAS. Any associated deletion and deallocation are sequenced after the atomic update step and are not part of the atomic operation.

Note that the control block of a `shared_ptr` is thread-safe: different non-atomic `std::shared_ptr` objects can be accessed using mutable operations, such as `operator=` or `reset`, simultaneously by multiple threads, even when these instances are copies, and share the same control block internally.

The type T may be an incomplete type.

## Member types

| Member type | Definition |
|---|---|
| `value_type` | `std::shared_ptr<T>` |

## Member functions

All non-specialized `std::atomic` functions are also provided by this specialization, and no additional member functions.

```cpp
std::atomic<std::shared_ptr<widget>> ptr
    = std::make_shared<widget>();

// Thread 1
auto ptr1 = ptr.load();

// Thread 2
ptr.store(ptr2);



// OK (since C++20:)
```

```
std::atomic<std::shared_ptr<widget>> ptr
    = std::make_shared<widget>();

// Thread 1
auto ptr1 = ptr;

// Thread 2
ptr = ptr2;



// OK (since C++20:)
```

```cpp
std::atomic<std::shared_ptr<widget>> ptr
    = std::make_shared<widget>();

// Thread 1
auto ptr1 = ptr;

// Thread 2
ptr = ptr2;

assert(ptr.is_lock_free()); // false
static_assert(decltype(ptr)::is_always_lock_free); // false
```

**lock-free** atomic shared_ptr

**lock-free** atomic shared_ptr

**atomic operations in CPU**

**lock-free** atomic shared_ptr

**atomic operations in CPU**

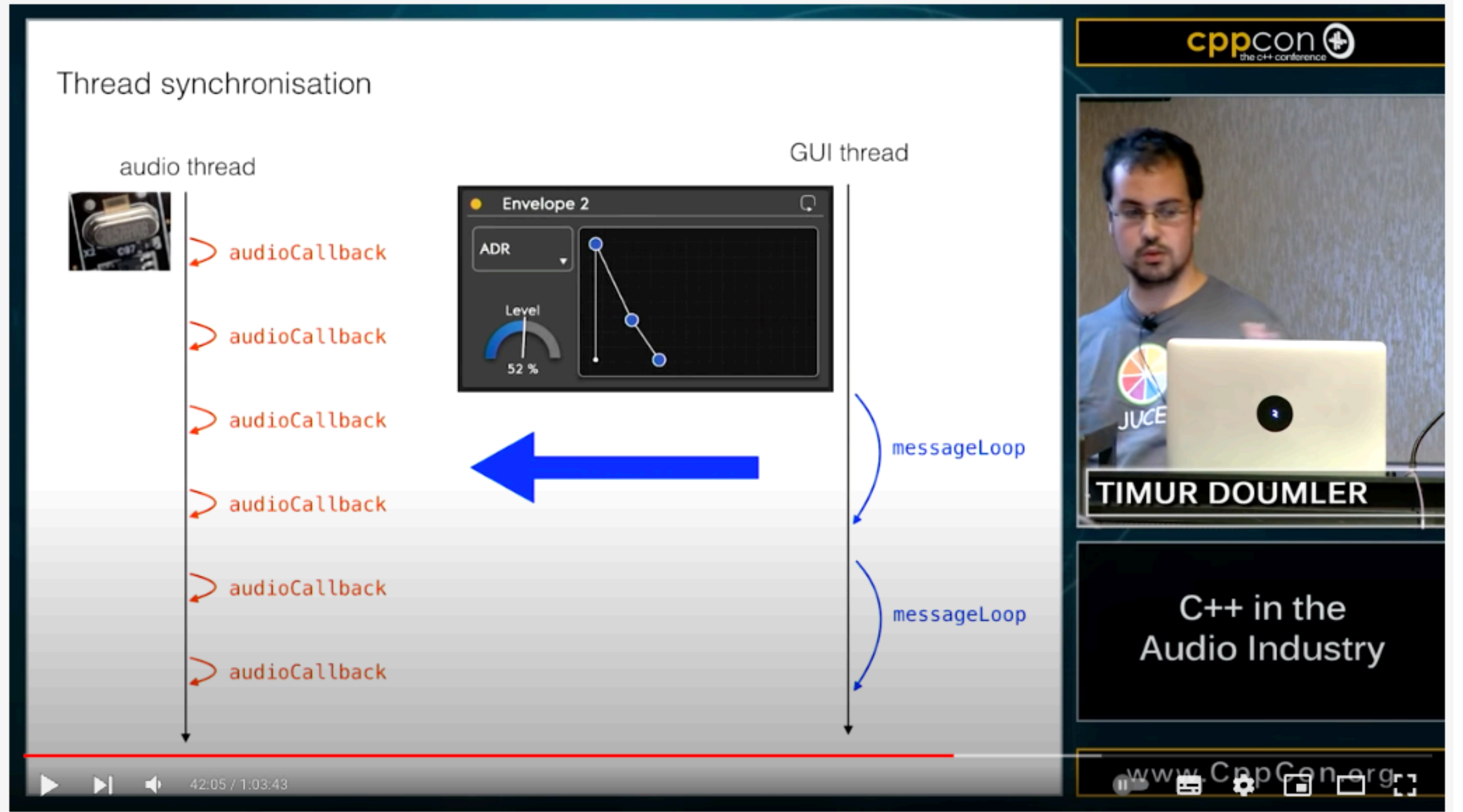
**suitable for low-latency / real-time**

**lock-free** atomic shared_ptr

**atomic operations in CPU**

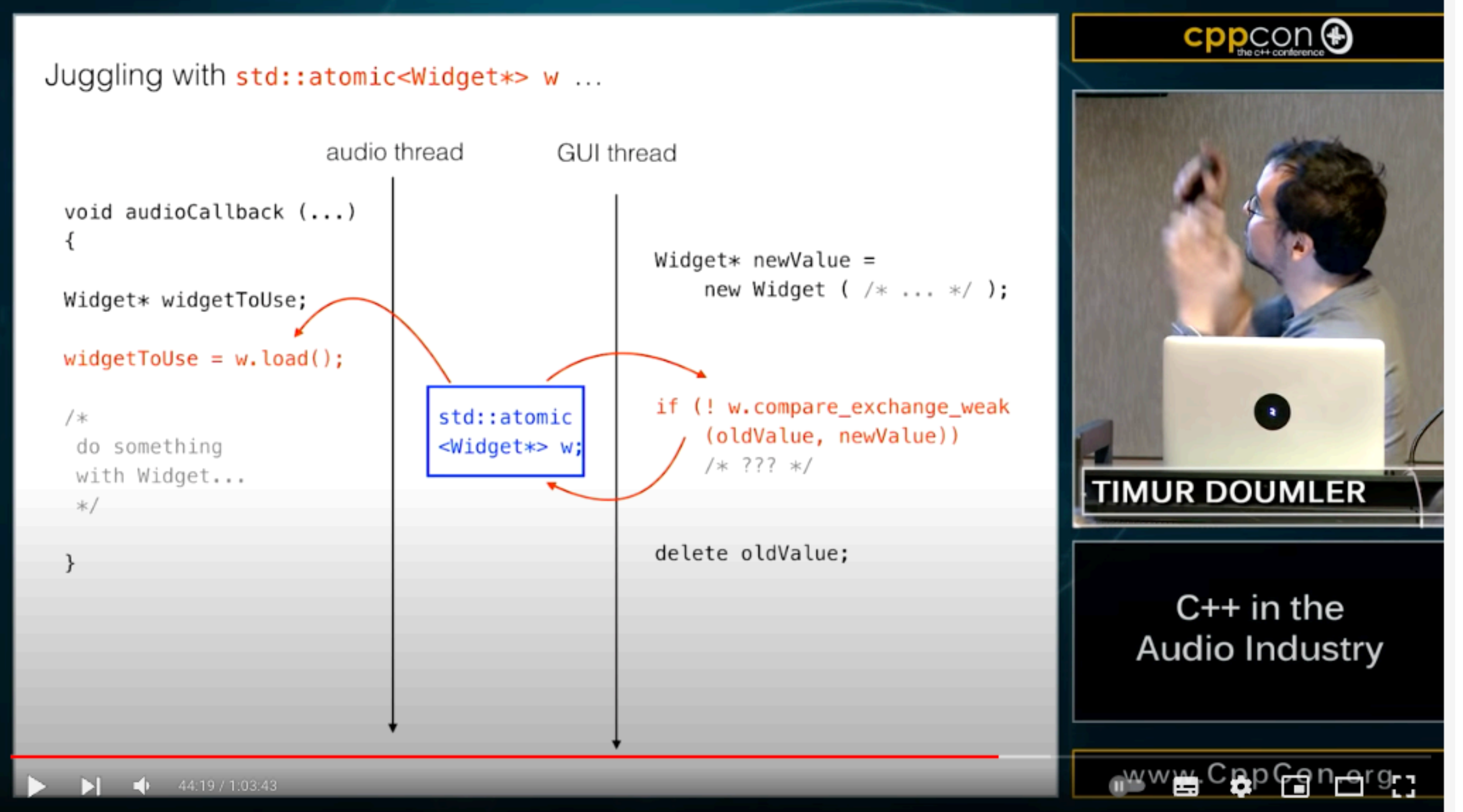
**suitable for low-latency / real-time**

**no priority inversion**




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

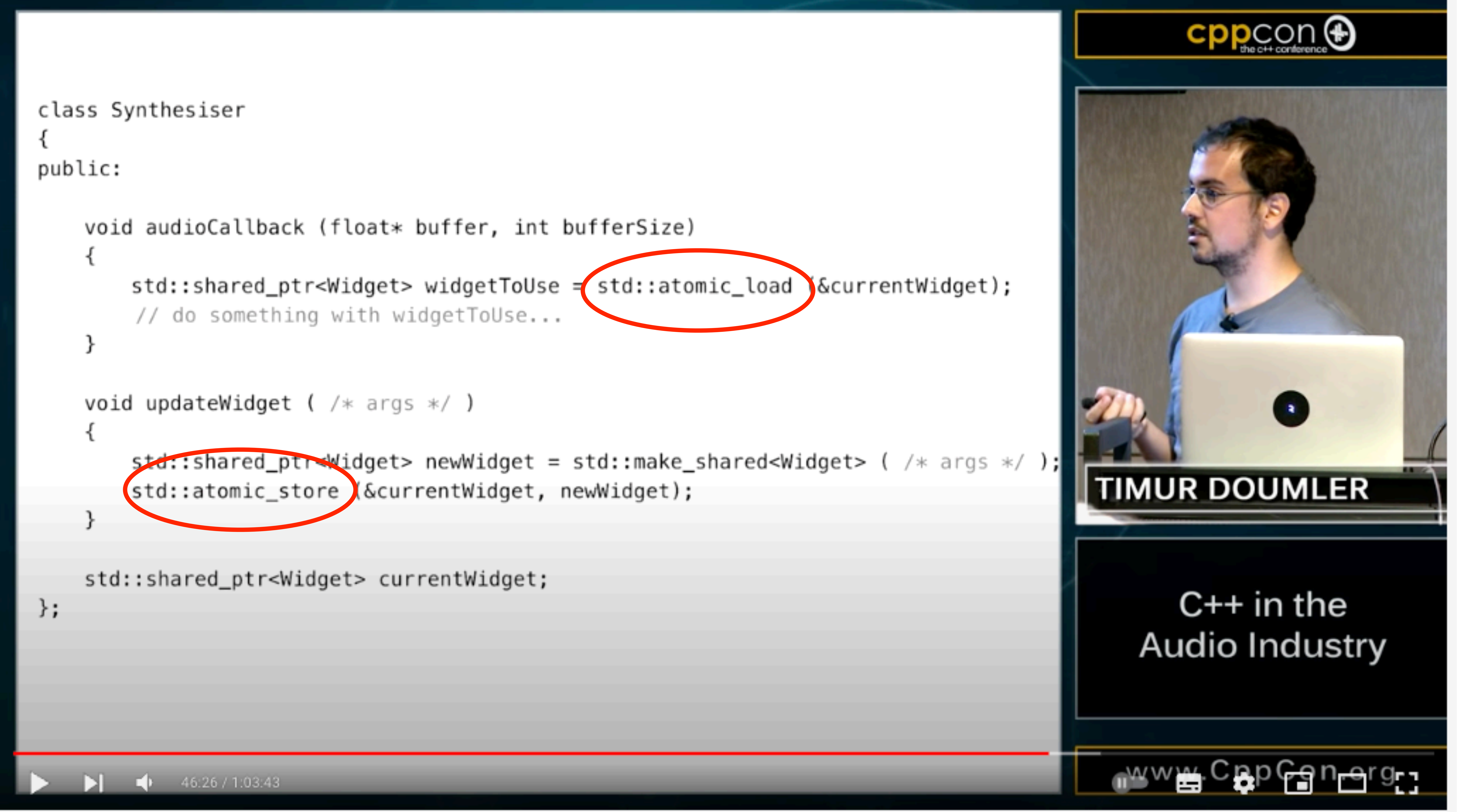
101,678 views • 9 Oct 2015
2.3K DISLIKE SHARE DOWNLOAD CLIP SAVE ...




CppCon 2015: Timur Doumler "C++ in the Audio Industry"

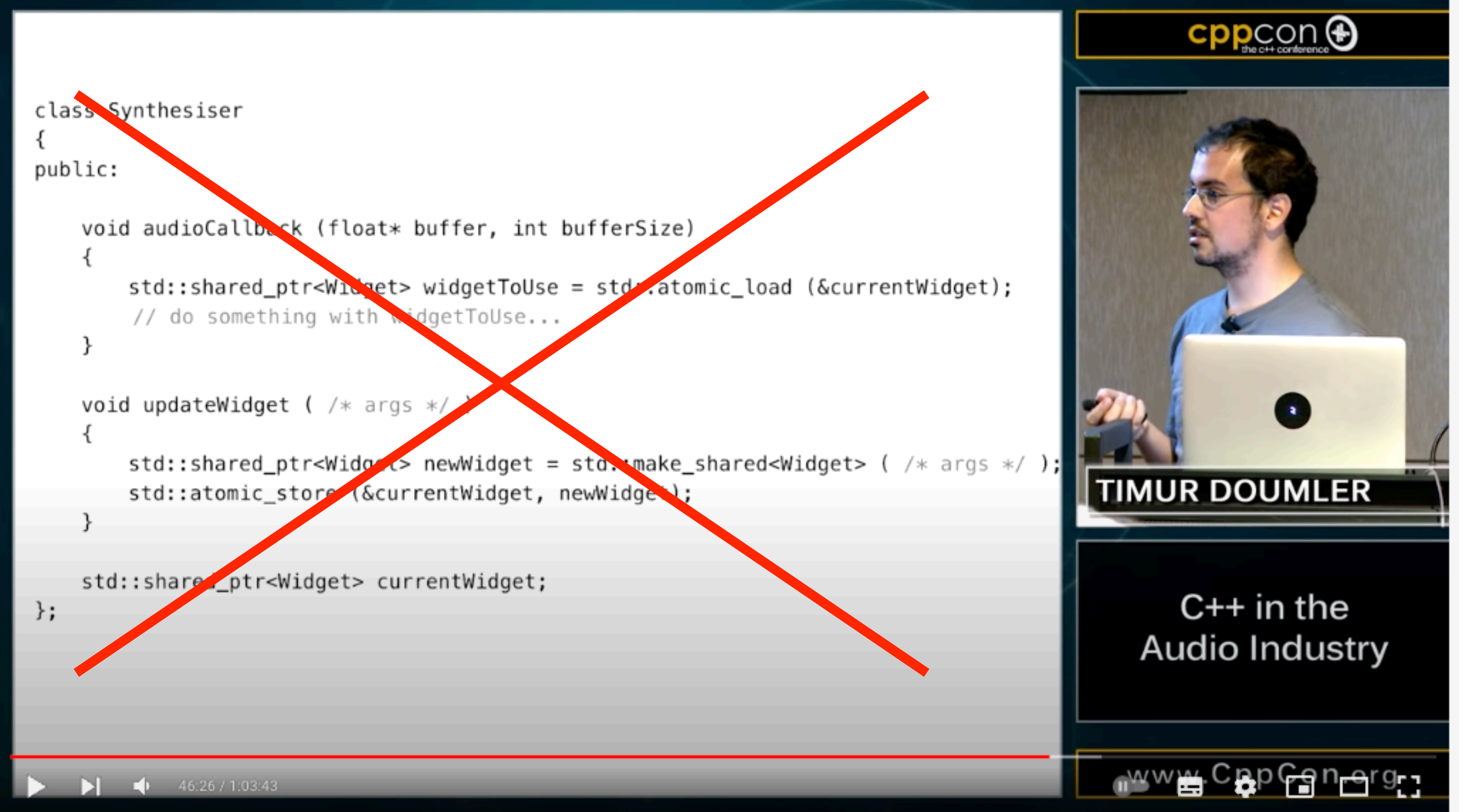
101,678 views • 9 Oct 2015

2.3K DISLIKE SHARE DOWNLOAD CLIP SAVE ...




CppCon 2015: Timur Doumler "C++ in the Audio Industry"

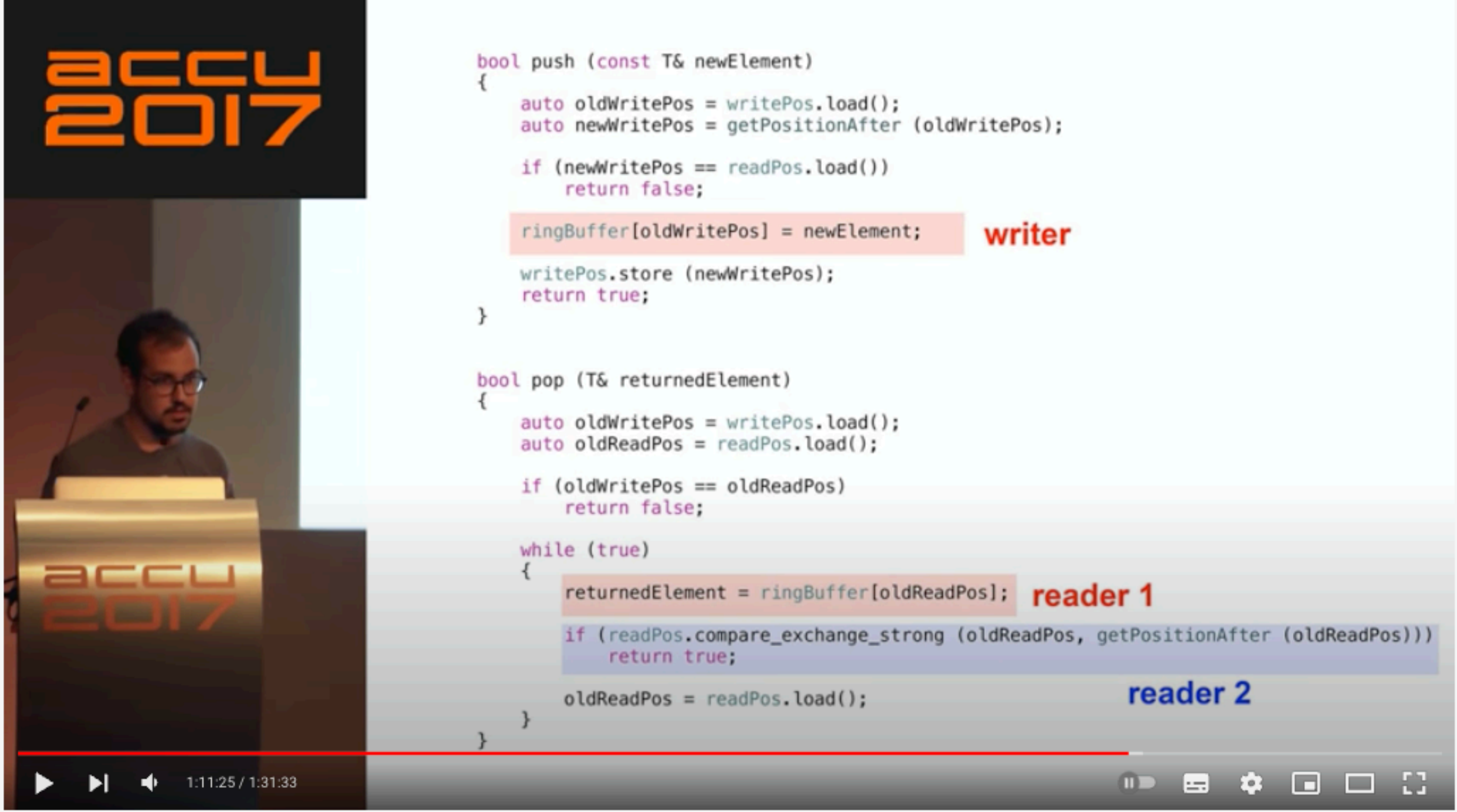
101,678 views • 9 Oct 2015 2.3K DISLIKE SHARE DOWNLOAD CLIP SAVE ...



class Synthesiser
{
public:
void audioCallback (float* buffer, int bufferSize)
{
std::shared_ptr<Widget> widgetToUse = std::atomic_load (¤tWidget);
// do something with widgetToUse...
}
void updateWidget ( /* args */ )
{
std::shared_ptr<Widget> newWidget = std::make_shared<Widget> ( /* args */ );
std::atomic_store (¤tWidget, newWidget);
}
std::shared_ptr<Widget> currentWidget;
};
cppcon
the c++ conference
TIMUR DOUMLER
C++ in the
Audio Industry
www.CppCon.org
46:26 / 1:03:43
CppCon 2015: Timur Doumler "C++ in the Audio Industry"
101,678 views • 9 Oct 2015
2.3K
DISLIKE
SHARE
DOWNLOAD
CLIP
SAVE

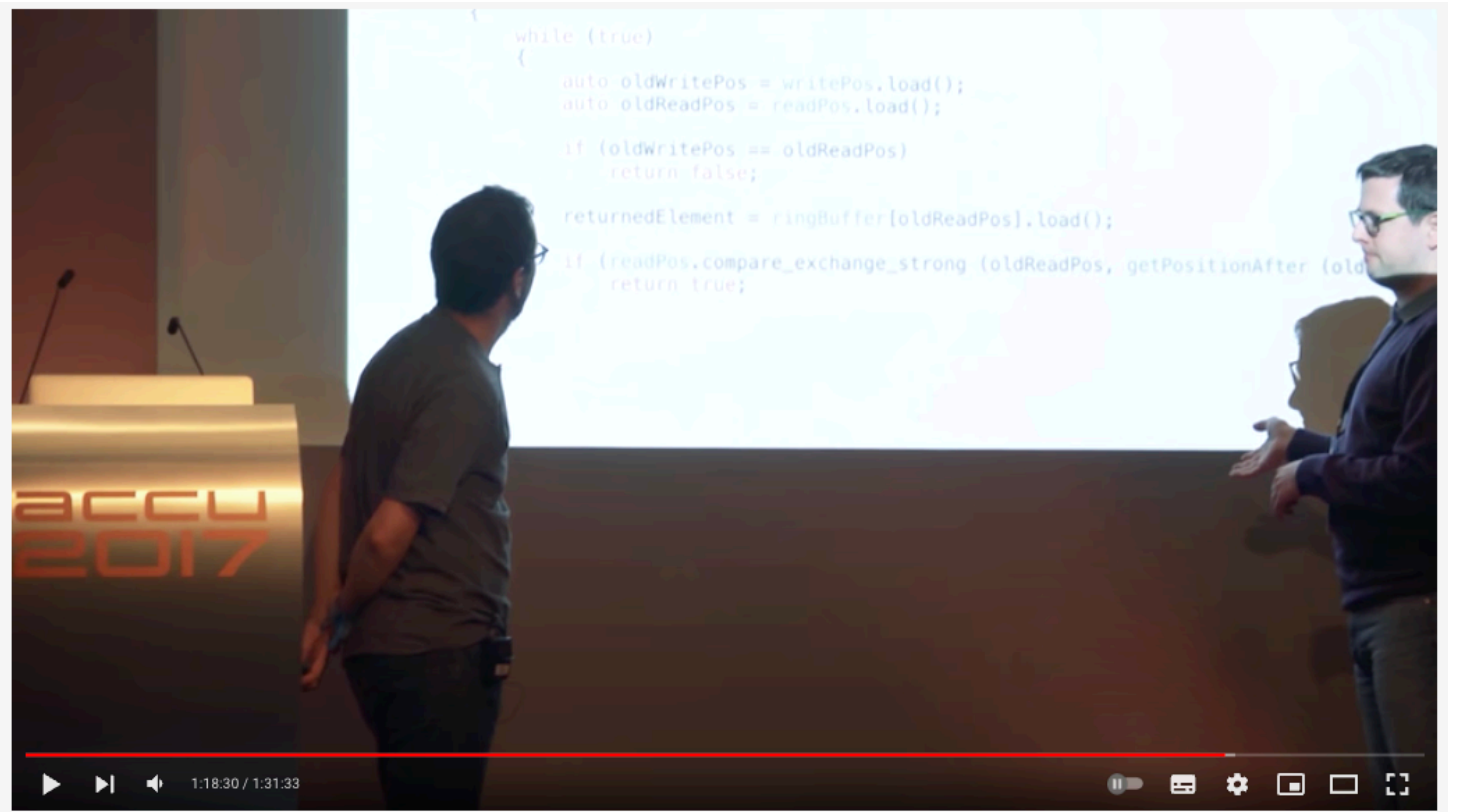
# lock-free structures

# lock-free structures & the ABA problem




- Lock-free programming with modern C++ - Timur Doumler [ACCU 2017]

14,927 views • 5 May 2017 183 DISLIKE SHARE DOWNLOAD CLIP SAVE ...




Lock-free programming with modern C++ - Timur Doumler [ACCU 2017]

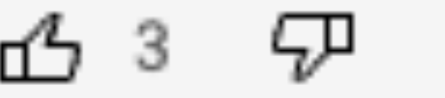
14,927 views • 5 May 2017

183 DISLIKE SHARE DOWNLOAD CLIP SAVE ...

Pinned by ACCU Conference

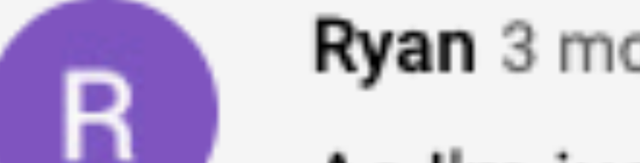
**Sa Hihi** 1 year ago

The MPMC queue still has an ABA problem, the CAS might succeed if the read pointer circled around. You could fix it with a generation counter I think.



3 REPLY

View 2 replies from ACCU Conference and others



**Ryan** 3 months ago

As I'm just now learning about lock-free and std::atomics (relatively new hobby programmer here), this talk was incredibly informative and my eyes are now large with a thirst for more knowledge on this stuff lol Very well put together in my opinion and was very easy to follow along

1 REPLY

**Giada LoopMachine** 3 years ago

This is pure gold. Thank you.

2 REPLY

```
// Author: Herb Sutter

template<typename T>
class concurrent_stack {
    struct Node {
        T t;
        shared_ptr<Node> next;
    };
    atomic<shared_ptr<Node>> head;
    concurrent_stack(concurrent_stack&) = delete;
    void operator=(concurrent_stack&) = delete;

public:
    concurrent_stack() = default;
    ~concurrent_stack() = default;

    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} {}
        T &operator*() { return p->t; }
        T *operator->() { return &p->t; }
    };

    auto find(T t) const {
        auto p = head.load();
        p = p->next;
        return reference(move(p));
    }

    auto front() const { return reference(head); }

    void push_front(T t) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head;
        while(!head.compare_exchange_weak(p->next, p))
            ;
    }

    void pop_front() {
        auto p = head.load();
        while (p && !head.compare_exchange_weak(p, p->next))
            ;
    }
};
```

```
// Author: Herb Sutter

template<typename T>
class concurrent_stack {
    struct Node {
        T t;
        shared_ptr<Node> next;
    };
    atomic<shared_ptr<Node>> head;
    concurrent_stack(concurrent_stack&) = delete;
    void operator=(concurrent_stack&) = delete;

public:
    concurrent_stack() = default;
    ~concurrent_stack() = default;

    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} {}
        T &operator*() { return p->t; }
        T *operator->() { return &p->t; }
    };

    auto find(T t) const {
        auto p = head.load();
        p = p->next;
        return reference(move(p));
    }

    auto front() const { return reference(head); }

    void push_front(T t) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head;
        while(!head.compare_exchange_weak(p->next, p))
            ;
    }

    void pop_front() {
        auto p = head.load();
        while (p && !head.compare_exchange_weak(p, p->next))
            ;
    }
};
```

FEDOR PIKUS

Read, Copy, Update, then what? RCU for non-kernel programmers

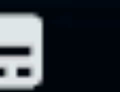
## RCU and alternatives

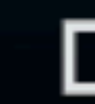
| | RCU | Hazard pointers | Atomic shared_ptr |
|---|---|---|---|
| Readers | wait-free population-oblivious | lock-free (wait-free?) | lock-free (slow) |
| Writers | single writer (BYOL) | lock-free | lock-free |
| Reclamation | blocking (or memory grows) | non-blocking | lock-free |
| Garbage | unbounded (or writers must block) | bounded by Nthreads*$N_{HP}$ | None |
| Ease of use | very easy | hard | easy (watch out for cycles) |

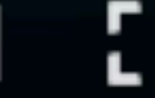
60 C++ RCU – CPPCon17 – F.G. Pikus

Mentor

CppCon.org 1:00:10 / 1:08:01







CppCon 2017: Fedor Pikus “Read, Copy, Update, then what? RCU for non-kernel programmers”

23,125 views • 19 Oct 2017 468 DISLIKE SHARE DOWNLOAD CLIP SAVE ...

**me in 2019**

VIA 9GAG.COM
Wrong.

**Timur Doumler**
@timur_audio

Hey C++ standard library implementers!

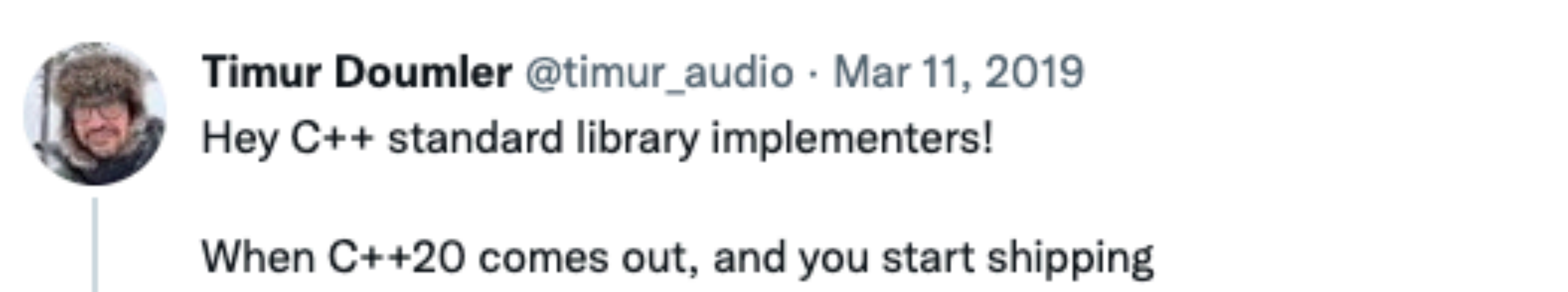
When C++20 comes out, and you start shipping `std::atomic<std::shared_ptr>`, will you ship a lock-free implementation?

@StephanTLavavej? And the others - who seem to be not on Twitter :(

5:27 PM · Mar 11, 2019 · Twitter Web Client

View Tweet analytics

**2** Retweets **1** Quote Tweet **20** Likes




`std::atomic<std::shared_ptr>`, will you ship a lock-free implementation?

@StephanTLavavej? And the others - who seem to be not on Twitter :(

5 3 20

**Stephan T. Lavavej**
@StephanTLavavej

Replying to @timur_audio

It is unclear to me how that can be implemented - double-wide atomics can copy the pointers, but how do you increment the refcount? BTW, @MalwareMinigun, @CoderCasey, @Eric01, @mclow are on Twitter (Jonathan Wakely was too, not sure if he still is).

6:10 PM · Mar 11, 2019 · Tweetbot for iOS

**Status Quo (April 2022):**

- **MSVC ships with std::atomic<std::shared_ptr>**
  - **Not lock-free**

- **GCC 12 will ship with std::atomic<std::shared_ptr>**
  - **Not lock-free**

- **Clang does not ship with std::atomic<std::shared_ptr>**

## Lock-freedom

I mentioned earlier that implementations *may* use a mutex to provide the synchronization in `atomic_shared_ptr`. They may also manage to make it lock-free. This can be tested using the `is_lock_free()` member function common to all the C++ atomic types.

The advantage of providing a lock-free `atomic_shared_ptr` should be obvious: by avoiding the use of a mutex, there is greater scope for concurrency of the `atomic_shared_ptr` operations. In particular, multiple concurrent reads can proceed unhindered.

The downside will also be apparent to anyone with any experience with lock-free code: the code is more complex, and has more work to do, so may be slower in the uncontended cases. The other big downside of lock-free code (maintenance and correctness) is passed over to the implementor!

It is my belief that the scalability benefits outweight the complexity, which is why Just::Thread v2.2 will ship with a lock-free `atomic_shared_ptr` implementation.

Posted by Anthony Williams
*[/ threading /] permanent link*
Tags: cplusplus, concurrency, multithreading
Stumble It! | Submit to Reddit | Submit to DZone

- **Anthony Williams' implementation, free version (proof of concept):**
  https://github.com/anthonywilliams/atomic_shared_ptr

- **Folly's implementation, by Dave Watson (production):**
  https://github.com/facebook/folly/blob/main/folly/concurrency/AtomicSharedPtr.h

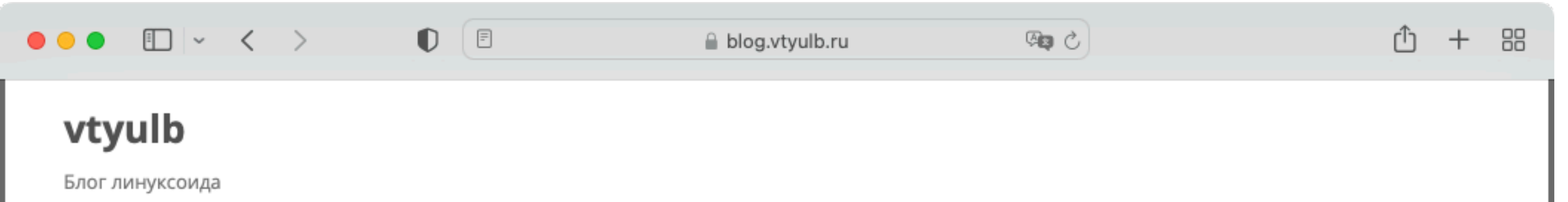
- **Vladislav Tyulbashev's implementation (proof of concept):**
  https://github.com/vtyulb/AtomicSharedPtr

- **Anthony Williams' implementation, free version (proof of concept):**
  https://github.com/anthonywilliams/atomic_shared_ptr

- **Folly's implementation, by Dave Watson (production):**
  https://github.com/facebook/folly/blob/main/folly/concurrency/AtomicSharedPtr.h

- **Vladislav Tyulbashev's implementation (proof of concept):**
  https://github.com/vtyulb/AtomicSharedPtr

**Huge thanks to Anthony Williams, David Goldblatt, and Vladislav Tyulbashev!**




ГЛАВНАЯ ПРОЕКТЫ ОБО МНЕ

# thread-safe lock-free shared_ptr — приключение на полгода

Добавить комментарий

Есть такой смешной вопрос «Is shared_ptr thread-safe?». Обычно либо человек сразу понимает о чем речь, либо ответа понять сходу невозможно.

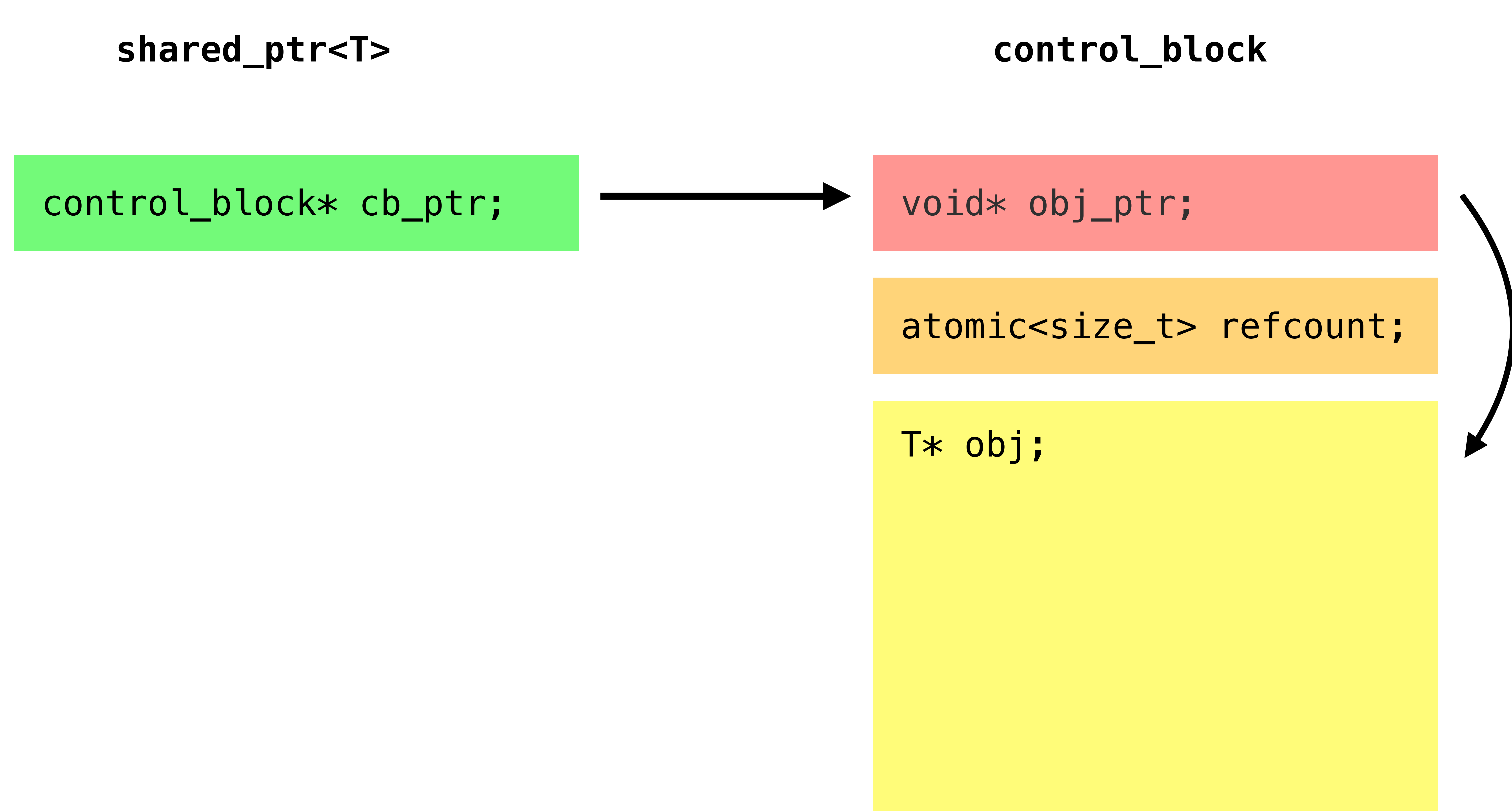
Проблема достаточно простая — shared_ptr нельзя модифицировать и читать из нескольких потоков одновременно. Контрольный блок дает thread-safe гарантии, тогда как сам shared_ptr — нет. Копирование shared_ptr состоит из трех операций:
— Скопировать указатель на объект
— Скопировать указатель на контрольный блок
— В контрольном блоке атомарно поднять refcount

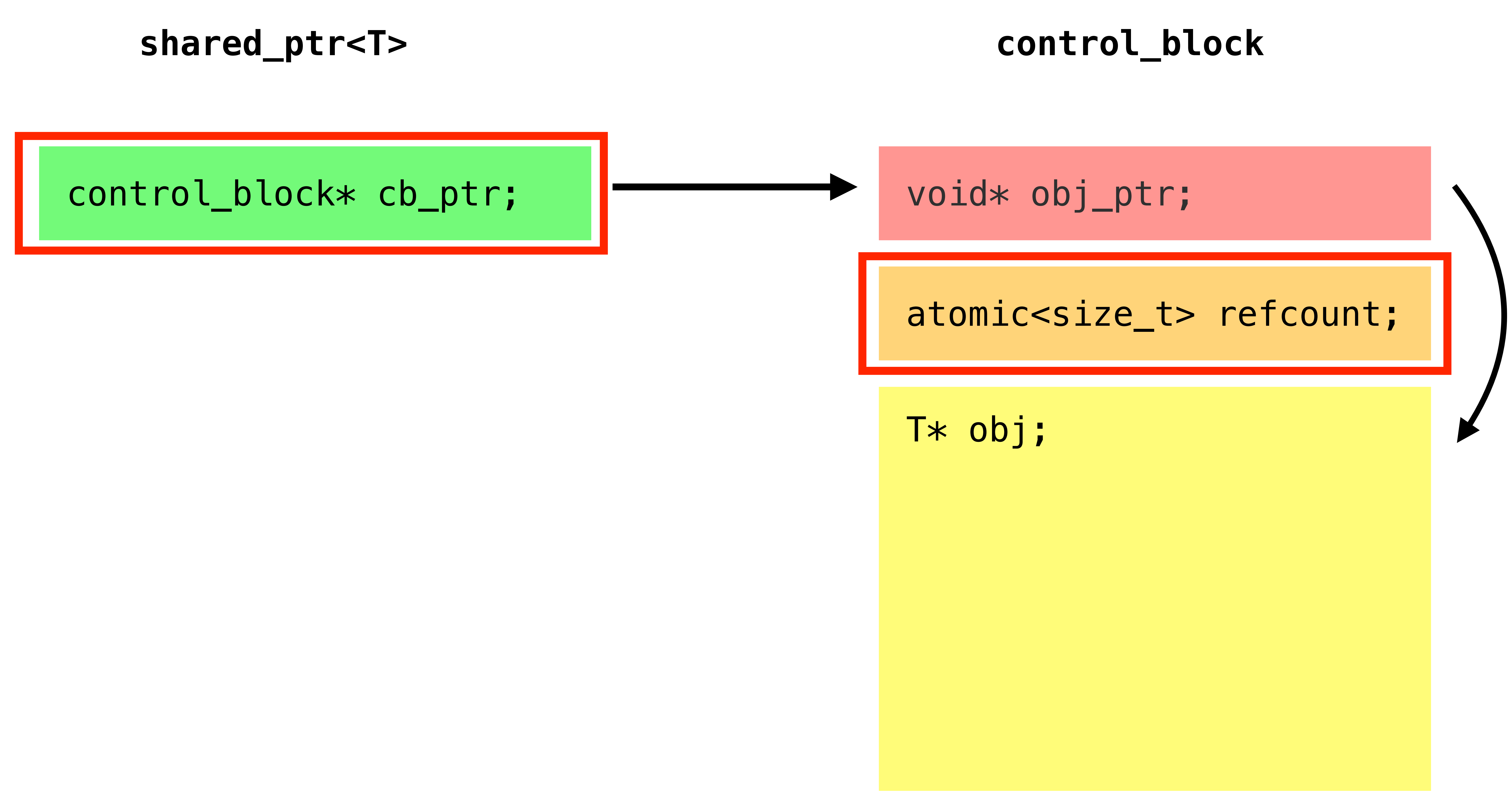
Если во время чтения shared_ptr какие-то из этих полей будут обновлены, то мы можем поднять refcount не у того объекта, либо попытаться получить доступ к уничтоженному

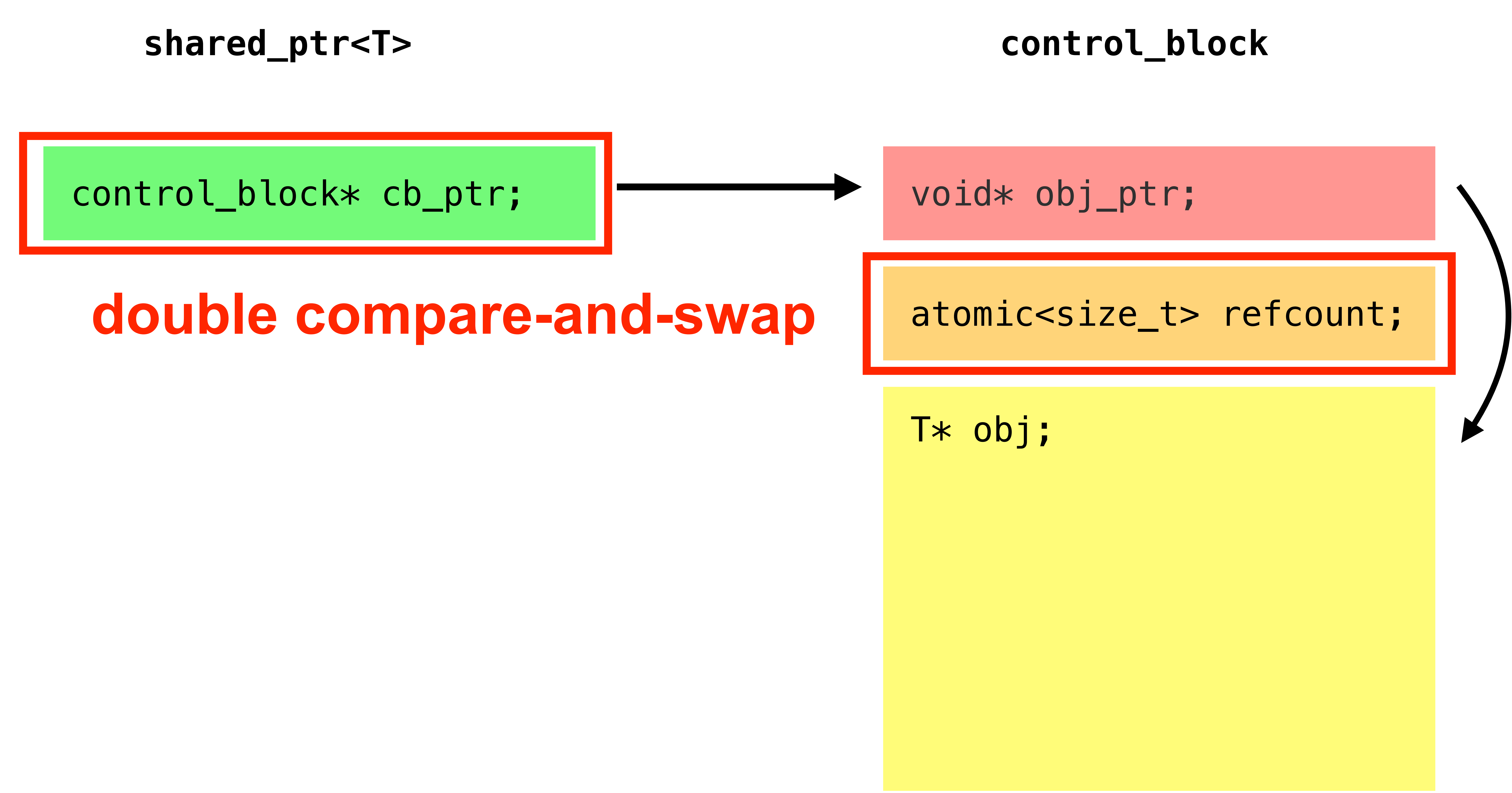
Поиск

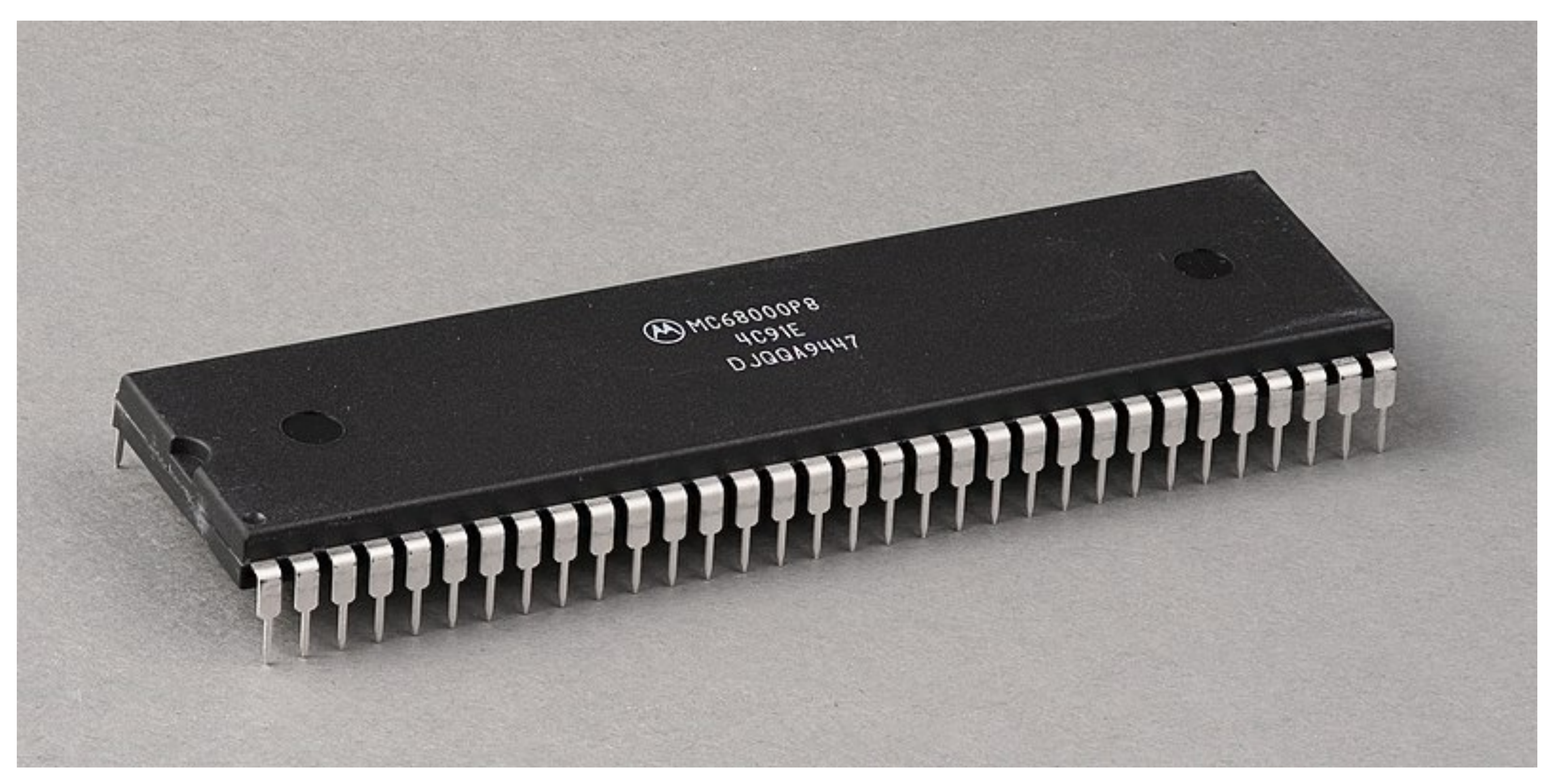
**СВЕЖИЕ ЗАПИСИ**

- Про локали C
- Очередной сервер
- Russian AI Cup 2020: CodeCraft
- CTF от безопасников
- Почему L3 лучше чем L2
- Цитаты великих
- thread-safe lock-free shared_ptr — приключение на полгода
- Баги в продакшене
- Коронавирус
- Kademlia


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
T* obj;


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
T* obj;


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
double compare-and-swap
atomic<size_t> refcount;
T* obj;




# CAS2

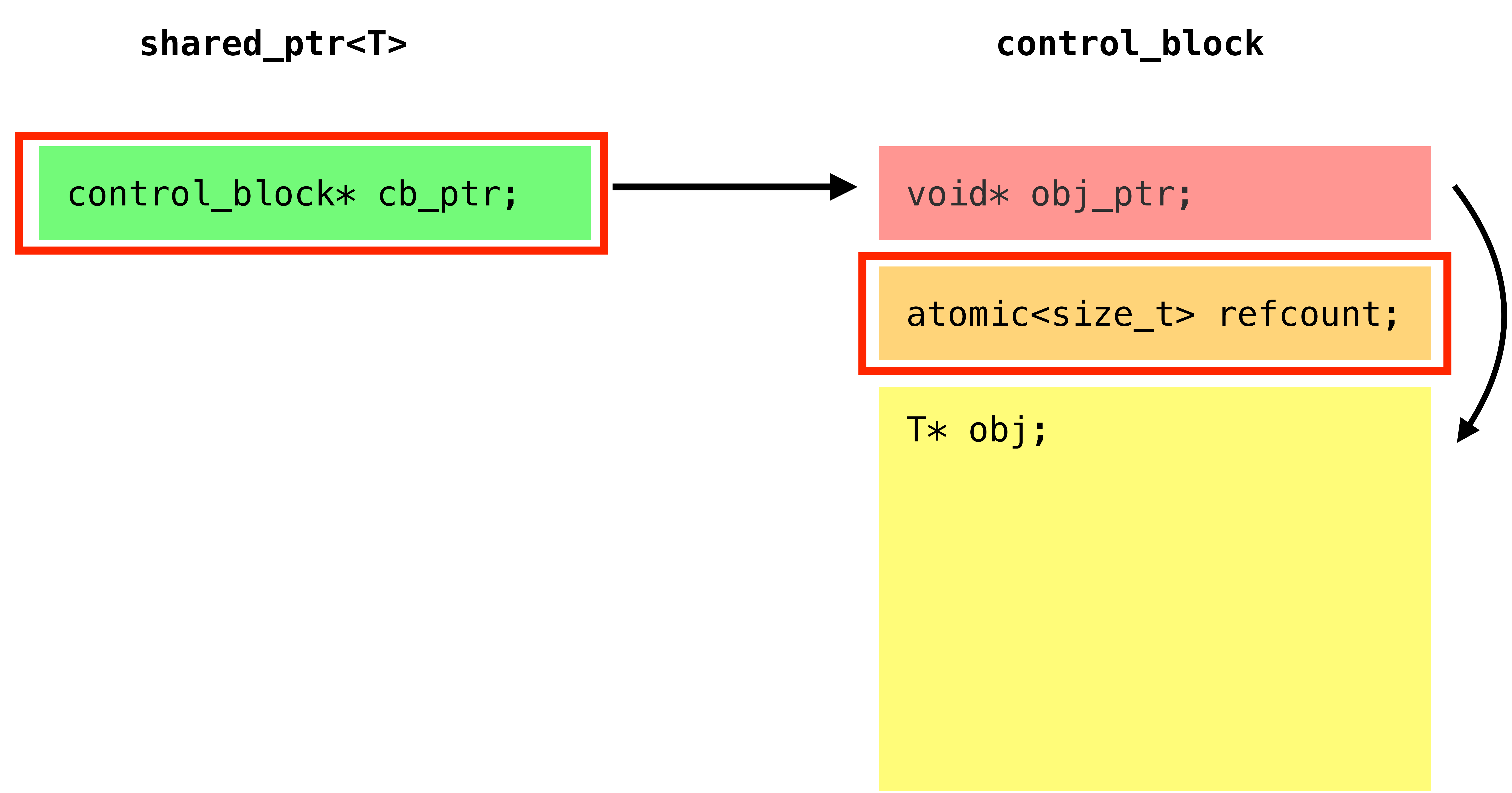
**Operation:**
CAS2 Destination 1 -- Compare 1 -->; cc;
if Z, Destination 2 -- Compare 2 -->; cc
if Z, Update 1 -->; Destination 1; Update 2 -->; Destination 2
else Destination 1 -->; Compare 1; Destination 2 -->; Compare 2

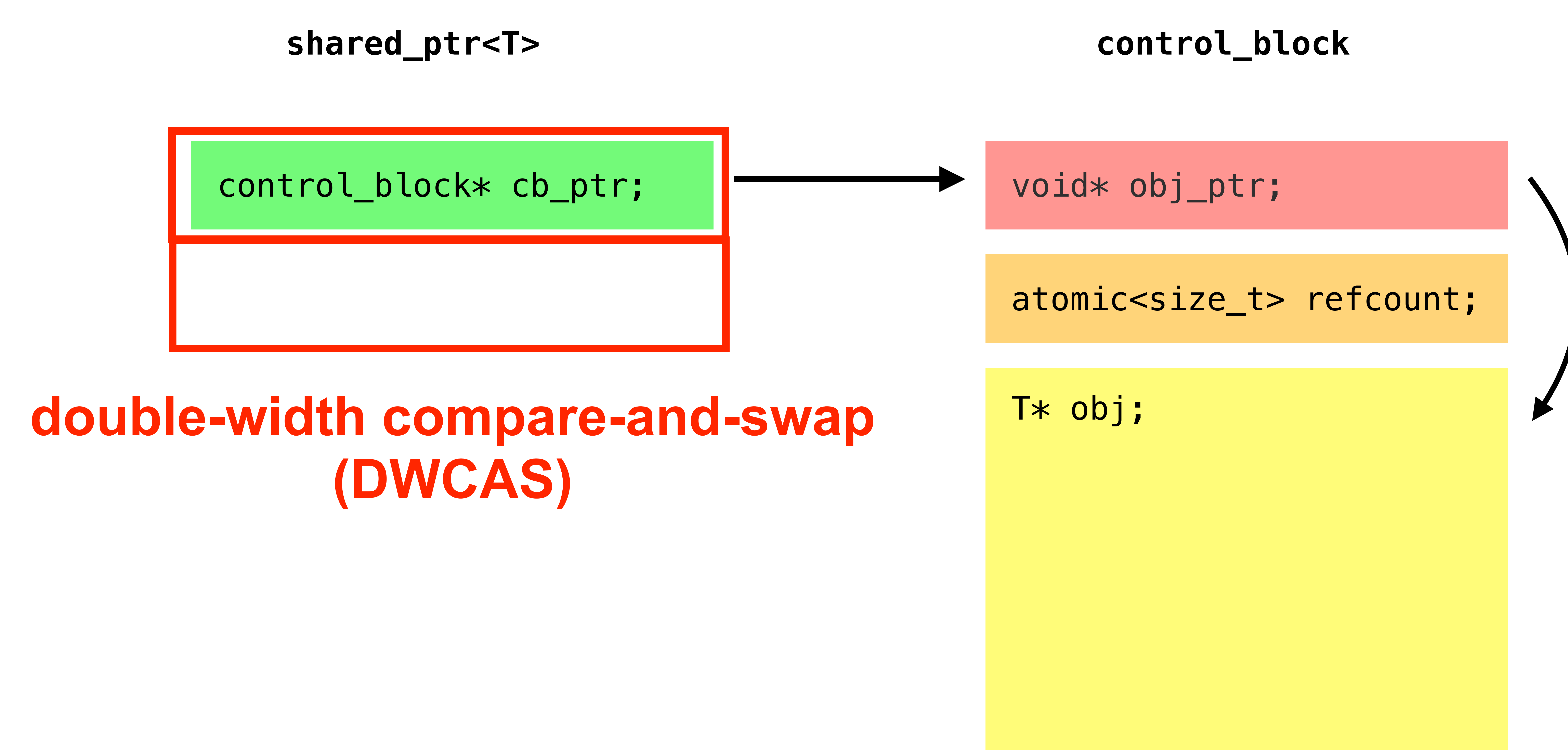
**Compatibility:** MC68020/MC68030/MC68040

**Assembler Syntax:**
CAS2 Dc1:Dc2, Du1:Du2, (Rn1):(Rn2)

**Attributes:** Size = (Word, Long)

**Description:** CAS2 compares memory operand 1 (Rn1) to compare operand 1 (Dc1). If the operands are equal, the instruction compares memory operand 2 (Rn2) to compare operand 2 (Dc2). If these operands are also equal, the instruction writes the update operands (Du1 and Du2) to the memory operands (Rn1 and Rn2). If either comparison fails, the instruction writes the memory operands (Rn1 and Rn2) to the compare operands (Dc1 and Dc2). The instruction accesses memory using locked or read-modify-write transfer sequences. This provides a means of synchronizing several processors.


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
T* obj;


shared_ptr<T>
control_block
control_block* cb_ptr;
void* obj_ptr;
atomic<size_t> refcount;
T* obj;
double-width compare-and-swap
(DWCAS)

# split refcount

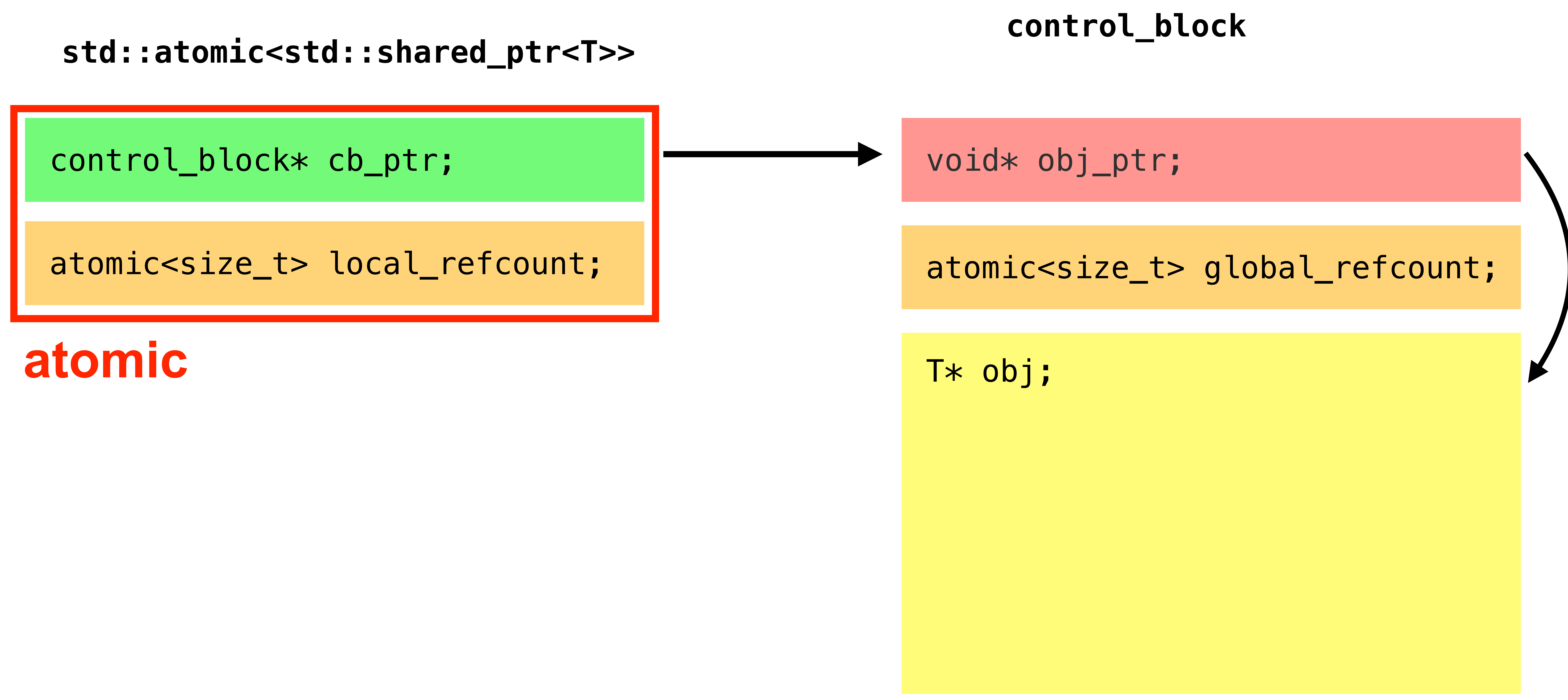
**split refcount**

*“It’s sort of a folk algorithm.”*

— David Goldblatt

- **Dmitry Vyukov: “Differential Reference Counting”**
  https://www.1024cores.net/home/lock-free-algorithms/object-life-time-management/differential-reference-counting

- **Anthony Williams: “C++ Concurrency in action”**
  Manning Publications




Copyright (c) Timur Doumler | @timur_audio | https://timur.audio

```
template<typename T>
class atomic<shared_ptr<T>> {
    struct counted_ptr {
        control_block* cb_ptr;
        size_t local_refcount;
    };

    atomic<counted_ptr> atomic_cptr;
    static_assert(decltype(value)::is_always_lock_free);

    // ...
};
```

## atomic copy: basic idea

- **atomically increment local_refcount**
- **get current [cb_ptr, local_refcount] pair**
- **try to sync refcounts:**
  - **atomically increment cb_ptr->global_refcount**
  - **atomically decrement local_refcount**
  - **if another thread modified [cb_ptr, local_refcount] in the meantime:**
    - **try again**
  - **else:**
    - **success**

```
template<typename T>
class atomic<shared_ptr<T>> {
    struct counted_ptr {
        control_block* cb_ptr;
        size_t local_refcount;
    };

    atomic<counted_ptr> atomic_cptr;
    static_assert(decltype(value)::is_always_lock_free);
```

```
template<typename T>
class atomic<shared_ptr<T>> {
    struct counted_ptr {
        control_block* cb_ptr;
        size_t local_refcount;
    };

    atomic<counted_ptr> atomic_cptr;
    static_assert(decltype(value)::is_always_lock_free);

    shared_ptr<T> load() {
        // 1. increment local refcount
        auto value = cptr.load();
        while (true) {
            auto new_value = value;
            ++new_value.local_refcount;
            if (atomic_cptr.compare_exchange_weak(value, new_value))
                break;
        }
        ++value.local_refcount;
```

```
template<typename T>
class atomic<shared_ptr<T>> {
    struct counted_ptr {
        control_block* cb_ptr;
        size_t local_refcount;
    };

    atomic<counted_ptr> atomic_cptr;
    static_assert(decltype(value)::is_always_lock_free);

    shared_ptr<T> load() {
        // 1. increment local refcount
        auto value = cptr.load();
        while (true) {
            auto new_value = value;
            ++new_value.local_refcount;
            if (atomic_cptr.compare_exchange_weak(value, new_value))
                break;
        }
        ++value.local_refcount;

        // 2. get copy
        auto cblock = new_value.cblock;
```

```
// 3. increment global refcount
cblock->global_refcount.fetch_add(1);
```

```
// 3. increment global refcount
cblock->global_refcount.fetch_add(1);

// 4. decrement local refcount
auto value_before;
while (true)
{
    auto new_value = value;
    --new_value.local_refcount;
    if (atomic_cptr.compare_exchange_weak(value, new_value))
        break;

    // if value changed, we were not supposed to modify global_refcount!
    if (value_before->cblock != value->cblock) {
        cblock->global_refcount.fetch_sub(1);
        break;
    }
}
```

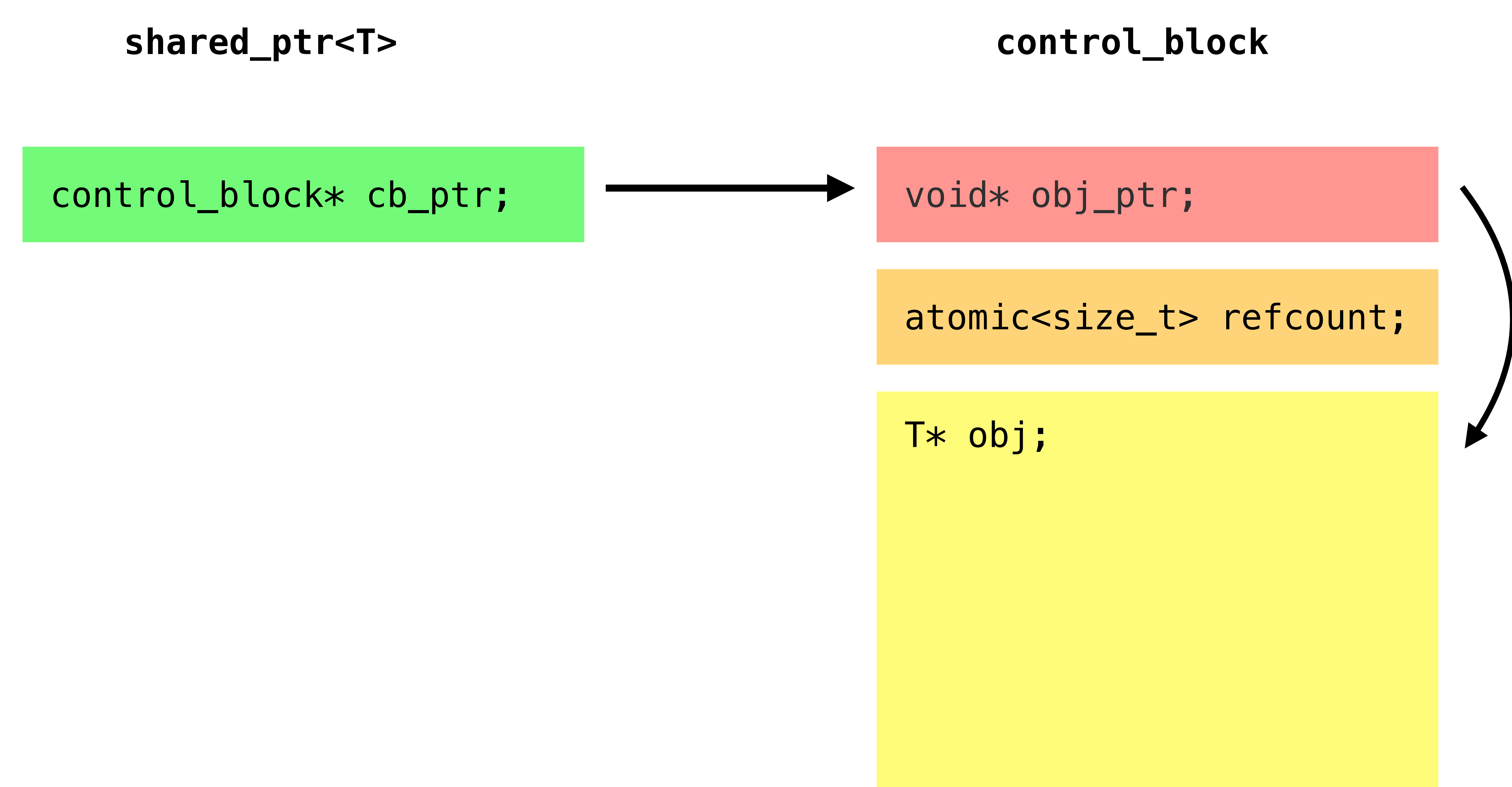
```
        // 3. increment global refcount
        cblock->global_refcount.fetch_add(1);

        // 4. decrement local refcount
        auto value_before;
        while (true)
        {
            auto new_value = value;
            --new_value.local_refcount;
            if (atomic_cptr.compare_exchange_weak(value, new_value))
                break;

            // if value changed, we were not supposed to modify global_refcount!
            if (value_before->cblock != value->cblock) {
                cblock->global_refcount.fetch_sub(1);
                break;
            }
        }

        // 5. return value
        return shared_ptr<T>{cblock};
    }
};
```

- **each atomic<shared_ptr> access:**
  - **at least 3 atomic operations, more under contention**
  - **slower than mutex when single-threaded**
  - **but much better than mutex under high contention**

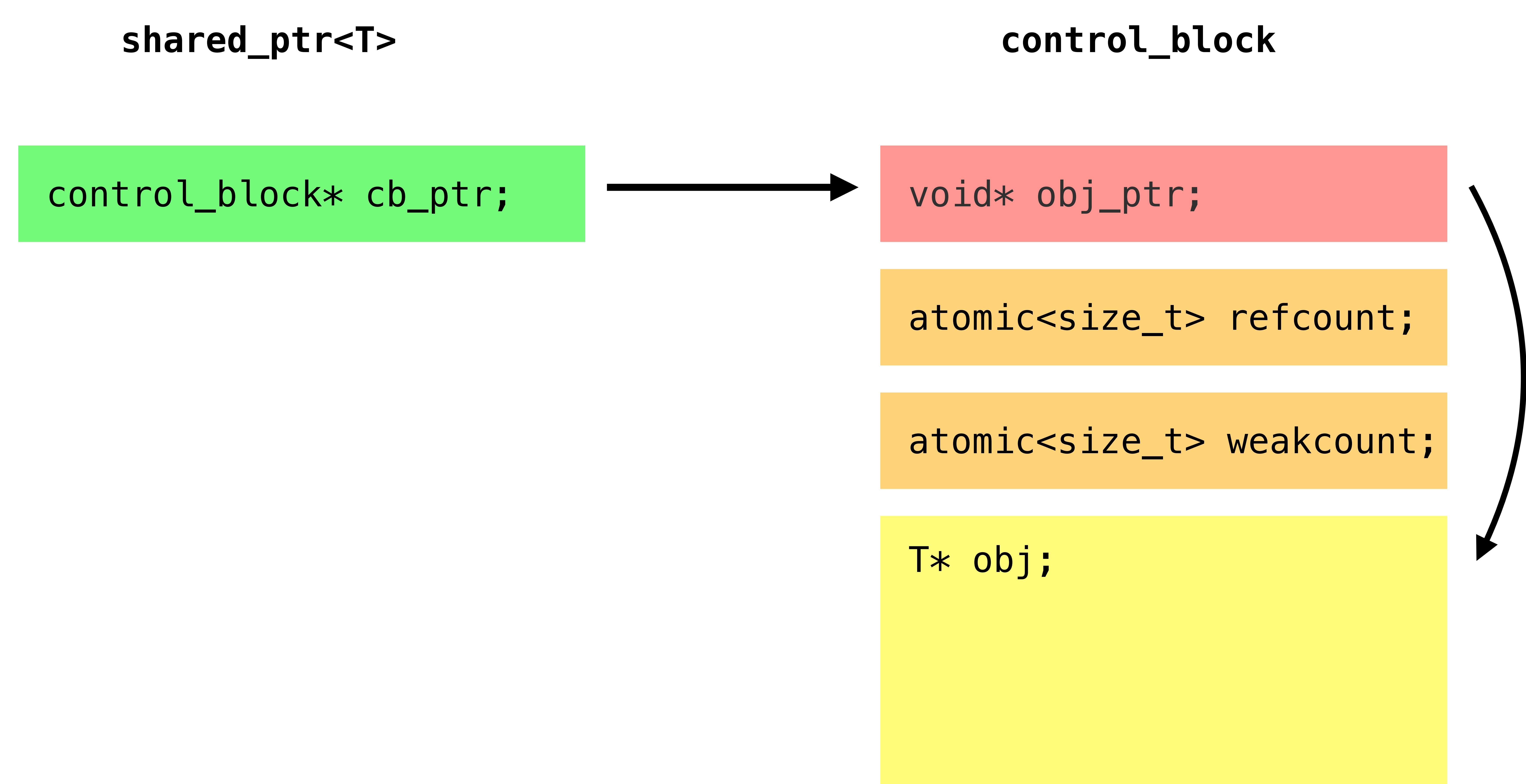
- **each atomic<shared_ptr> access:**
  - **at least 3 atomic operations, more under contention**
  - **slower than mutex when single-threaded**
  - **but much better than mutex under high contention**

- **folly optimisation: “refcount batching”**
  - **increment global refcount to some number K > numThreads**
  - **next K/2 loads do not have to touch cblock (= 1 atomic operation)**
  - **tradeoff: use_count() becomes meaningless**

# lock-free atomic shared_ptr?




Copyright (c) Timur Doumler 




Copyright (c) Timur Doumler

**shared_ptr<T>**

**control_block**

```
control_block* cb_ptr;
```

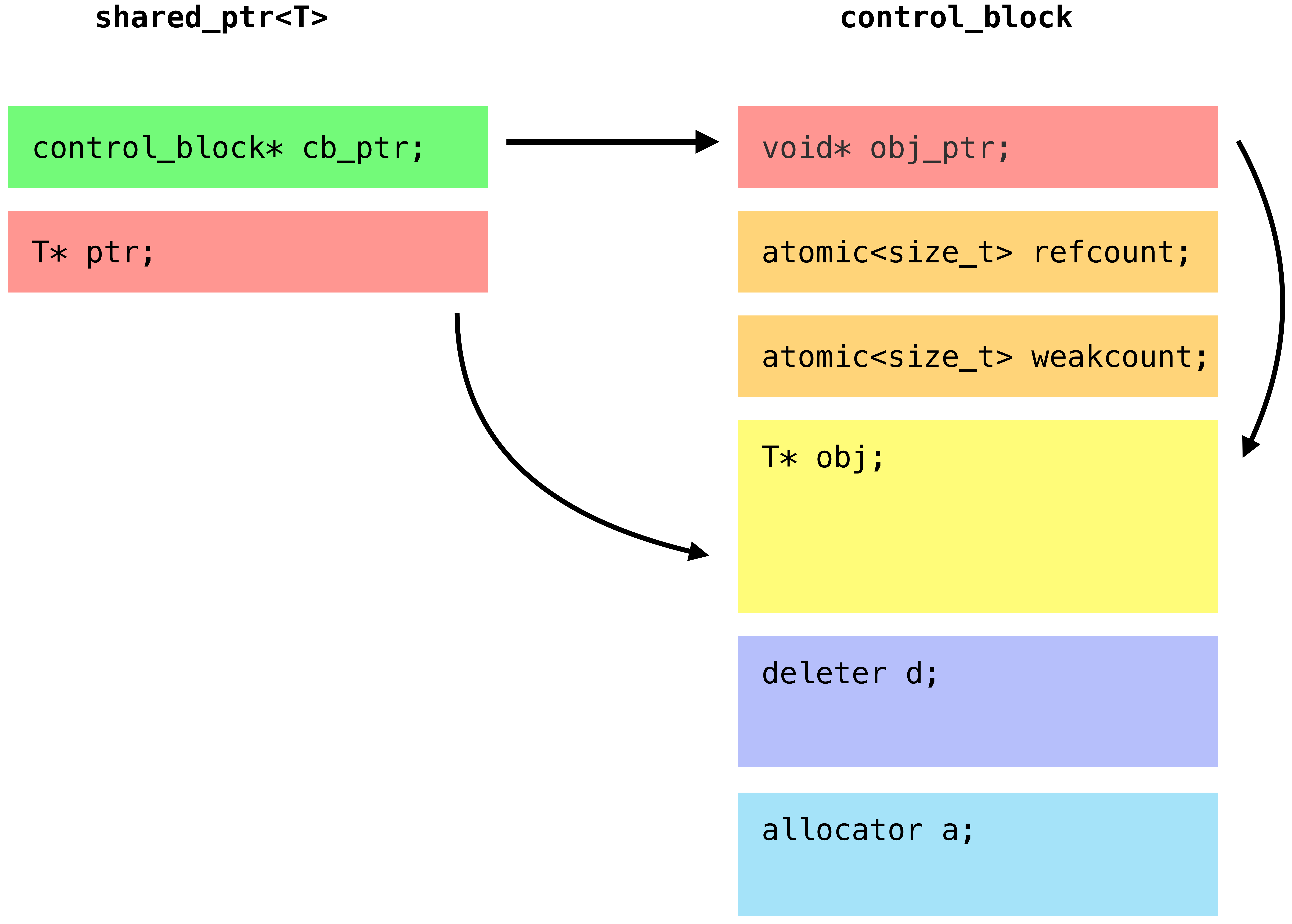
```
void* obj_ptr;
atomic<size_t> refcount;
atomic<size_t> weakcount;
T* obj;
deleter d;
```

**shared_ptr<T>**

**control_block**

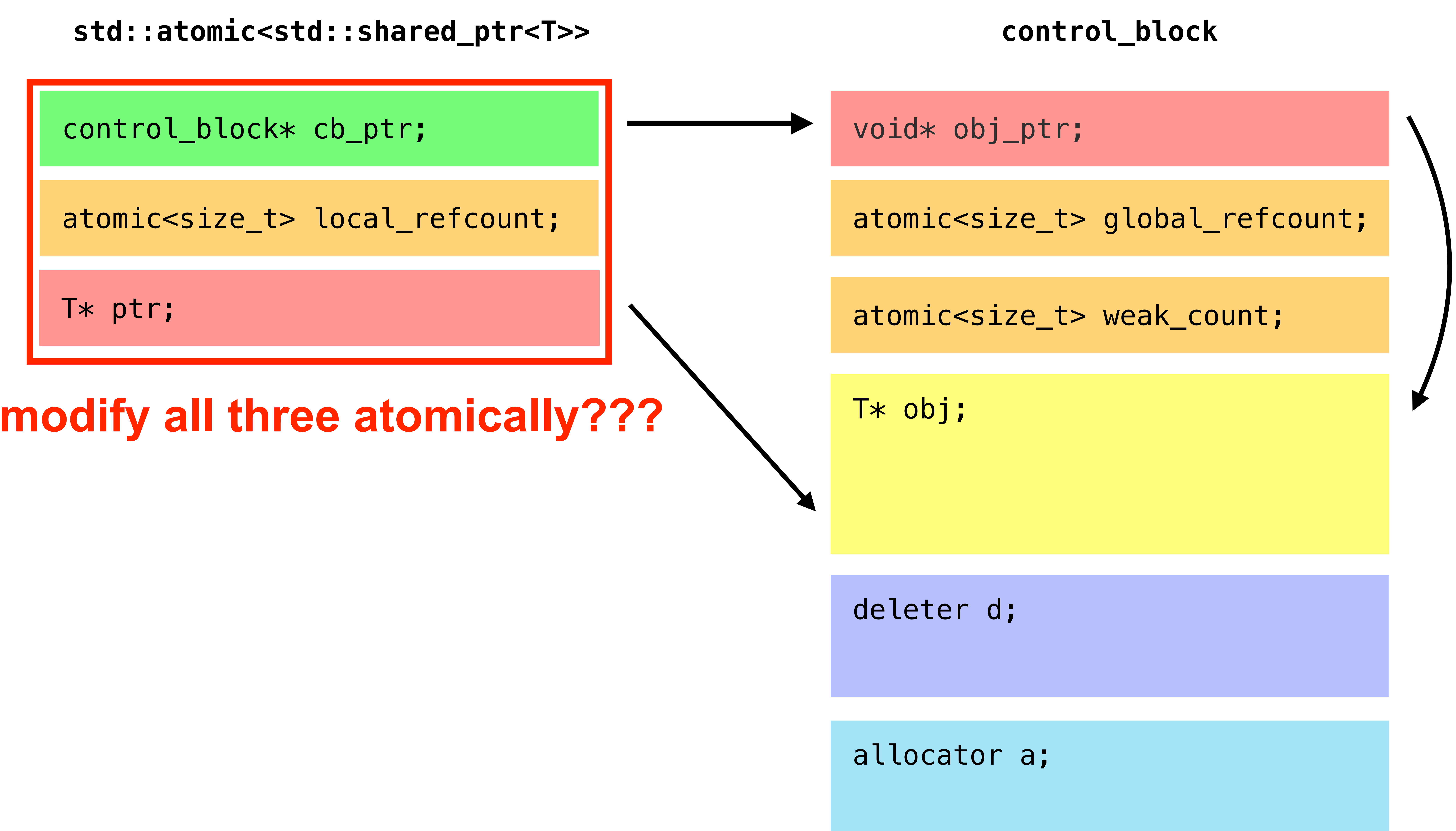
```
control_block* cb_ptr;
```

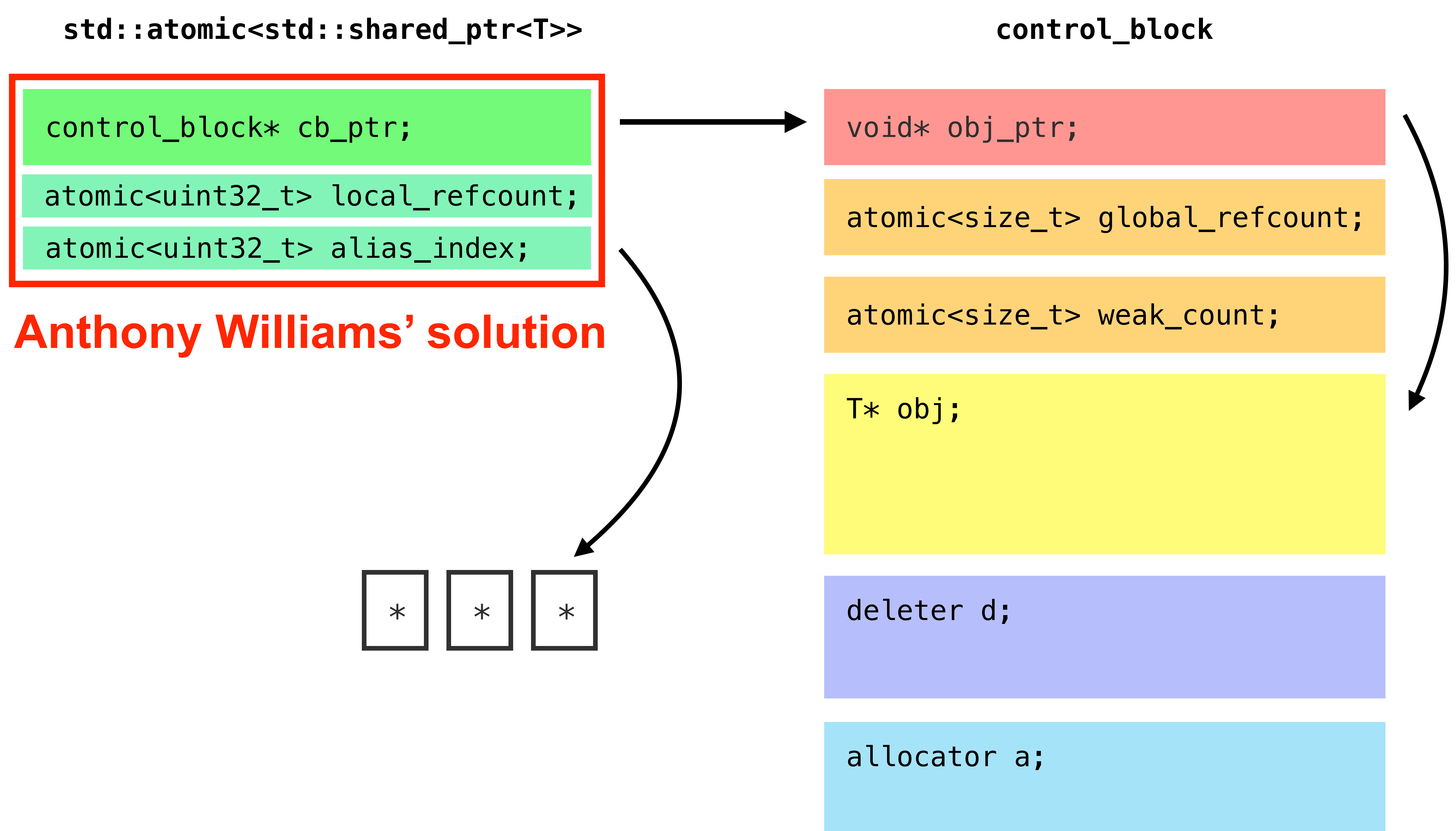
```
void* obj_ptr;
atomic<size_t> refcount;
atomic<size_t> weakcount;
T* obj;
deleter d;
allocator a;
```

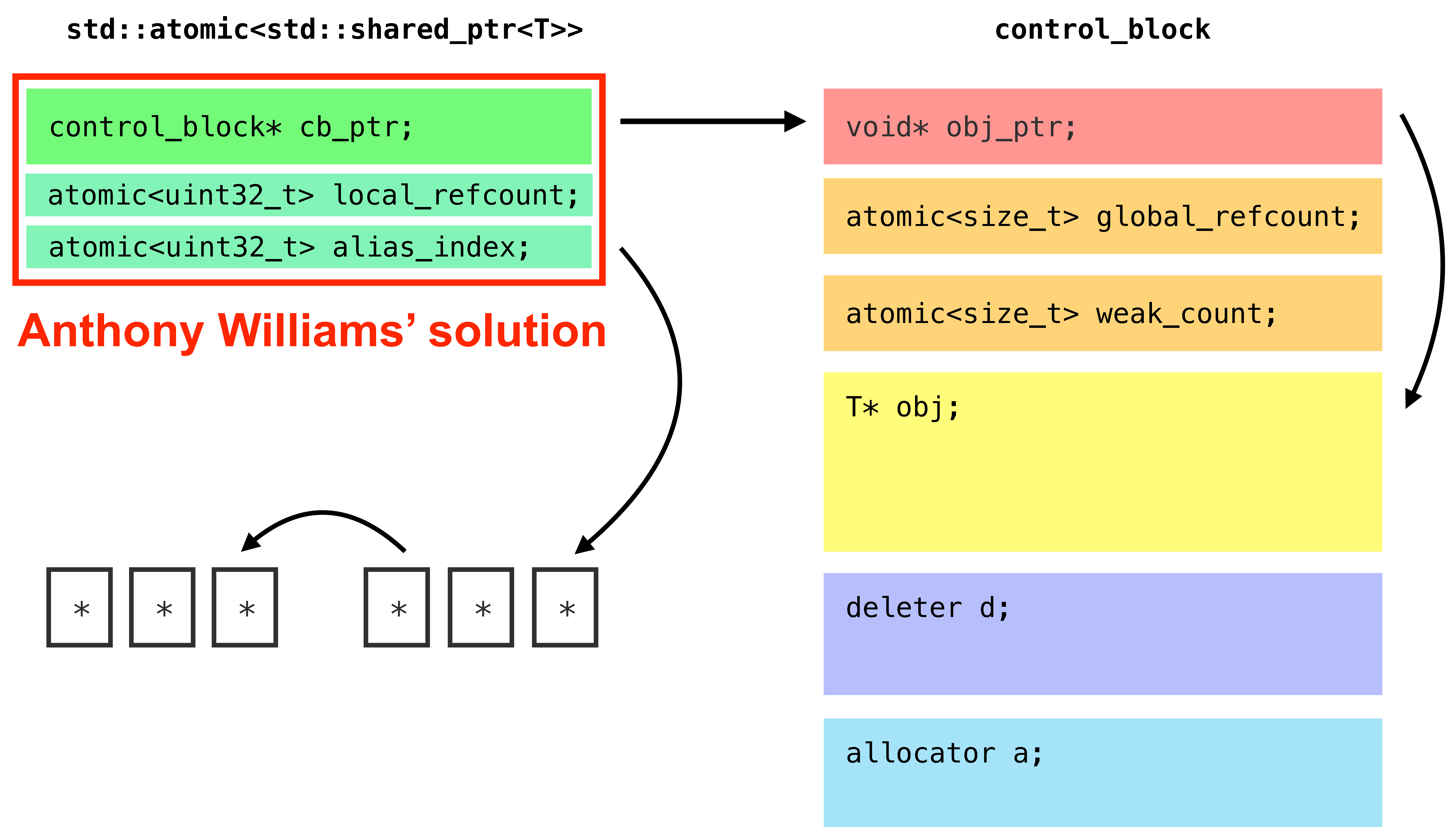

shared_ptr<T>
control_block
control_block* cb_ptr;
T* ptr;
void* obj_ptr;
atomic<size_t> refcount;
atomic<size_t> weakcount;
T* obj;
deleter d;
allocator a;

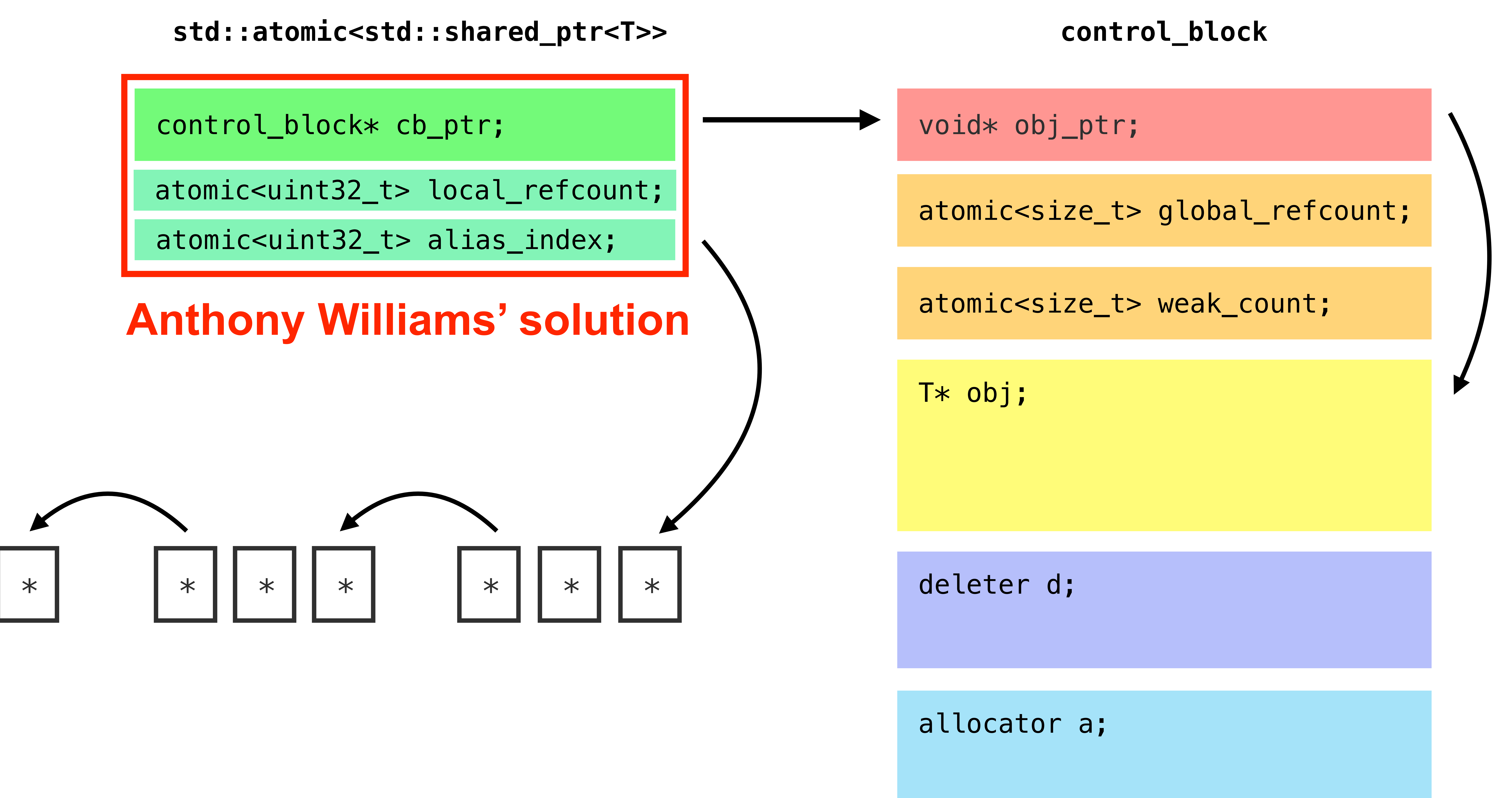
```
struct widget {
    int i = 0;
    int j = 0;
};

int main() {
    auto pw = std::make_shared<widget>(2, 5);
    std::shared_ptr<int> pj(pw, &pw->j);
}
```

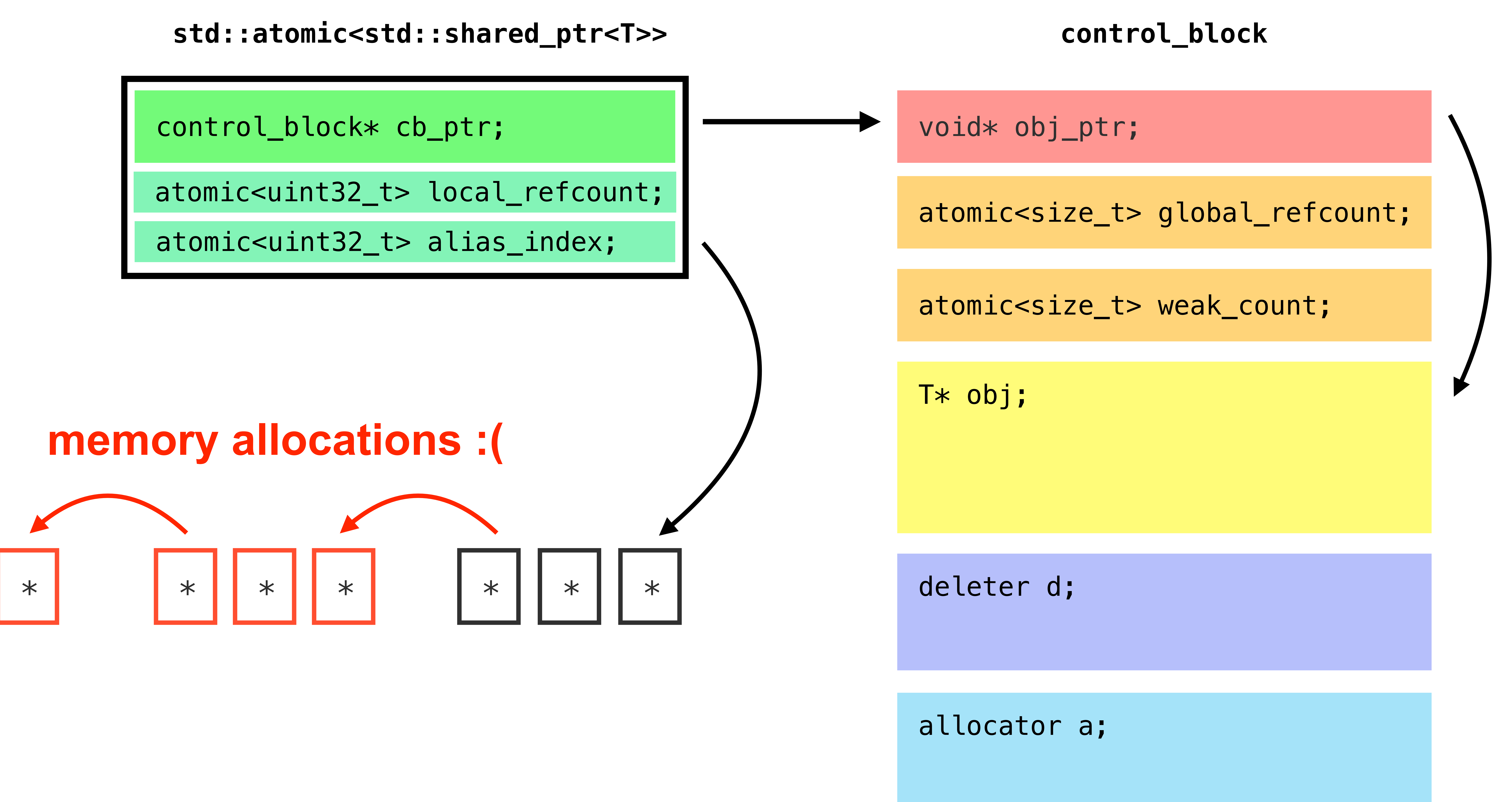
```
std::shared_ptr<base> ptr(new derived);
```


std::atomic<std::shared_ptr<T>>
control_block
control_block* cb_ptr;
atomic<size_t> local_refcount;
T* ptr;
modify all three atomically???
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;


std::atomic<std::shared_ptr<T>>
control_block* cb_ptr;
atomic<uint32_t> local_refcount;
atomic<uint32_t> alias_index;
Anthony Williams' solution
* * *
control_block
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;


std::atomic<std::shared_ptr<T>>
control_block* cb_ptr;
atomic<uint32_t> local_refcount;
atomic<uint32_t> alias_index;
Anthony Williams' solution
* * * * * *
control_block
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;


std::atomic<std::shared_ptr<T>>
control_block* cb_ptr;
atomic<uint32_t> local_refcount;
atomic<uint32_t> alias_index;
Anthony Williams' solution
*
*
*
*
*
*
*
control_block
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;


std::atomic<std::shared_ptr<T>>
control_block* cb_ptr;
atomic<uint32_t> local_refcount;
atomic<uint32_t> alias_index;
control_block
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;
memory allocations :(
*
*
*
*
*
*
*

**JF Bastien** @jfbastien · Mar 11, 2019

Replying to @timur_audio and @StephanTLavavej

Last I looked you needed a lock-free allocator to implement atomic<shared_ptr<T>> in case your inline buffer of class inheritance was too small. That's because the T can travel you class hierarchy. @a_williams has an implementation, probably knows better.

2 2

**Anthony Williams** @a_williams · Mar 12, 2019

You're right: if you have many different shared_ptr values that point to the same block (e.g. derived ptr, base ptr, other base ptr, aliased ptr, etc.) then my implementation needs a lock-free allocator for the extension blocks. The bitbucket version allows 3 pointers w/o alloc.

1 3

**JF Bastien** @jfbastien · Mar 11, 2019

Replying to @timur_audio and @StephanTLavavej

Last I looked you needed a lock-free allocator to implement atomic<shared_ptr<T>> in case your inline buffer of class inheritance was too small. That's because the T can travel you class hierarchy. @a_williams has an implementation, probably knows better.

2 2

**Anthony Williams** @a_williams · Mar 12, 2019

You're right: if you have many different shared_ptr values that point to the same block (e.g. derived ptr, base ptr, other base ptr, aliased ptr, etc.) then my implementation needs a lock-free allocator for the extension blocks. The bitbucket version allows 3 pointers w/o alloc.

1 3

## my suggested solution:

- **move allocation from atomic_shared_ptr to shared_ptr constructor**

## my suggested solution:

- move allocation from atomic_shared_ptr to shared_ptr constructor
  - but that’s overhead for every aliased shared_ptr :(

**my suggested solution:**

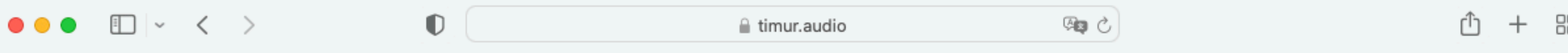
- **move allocation from atomic_shared_ptr to shared_ptr constructor**
- **compute ptrdiff between object ptr and aliased ptr**

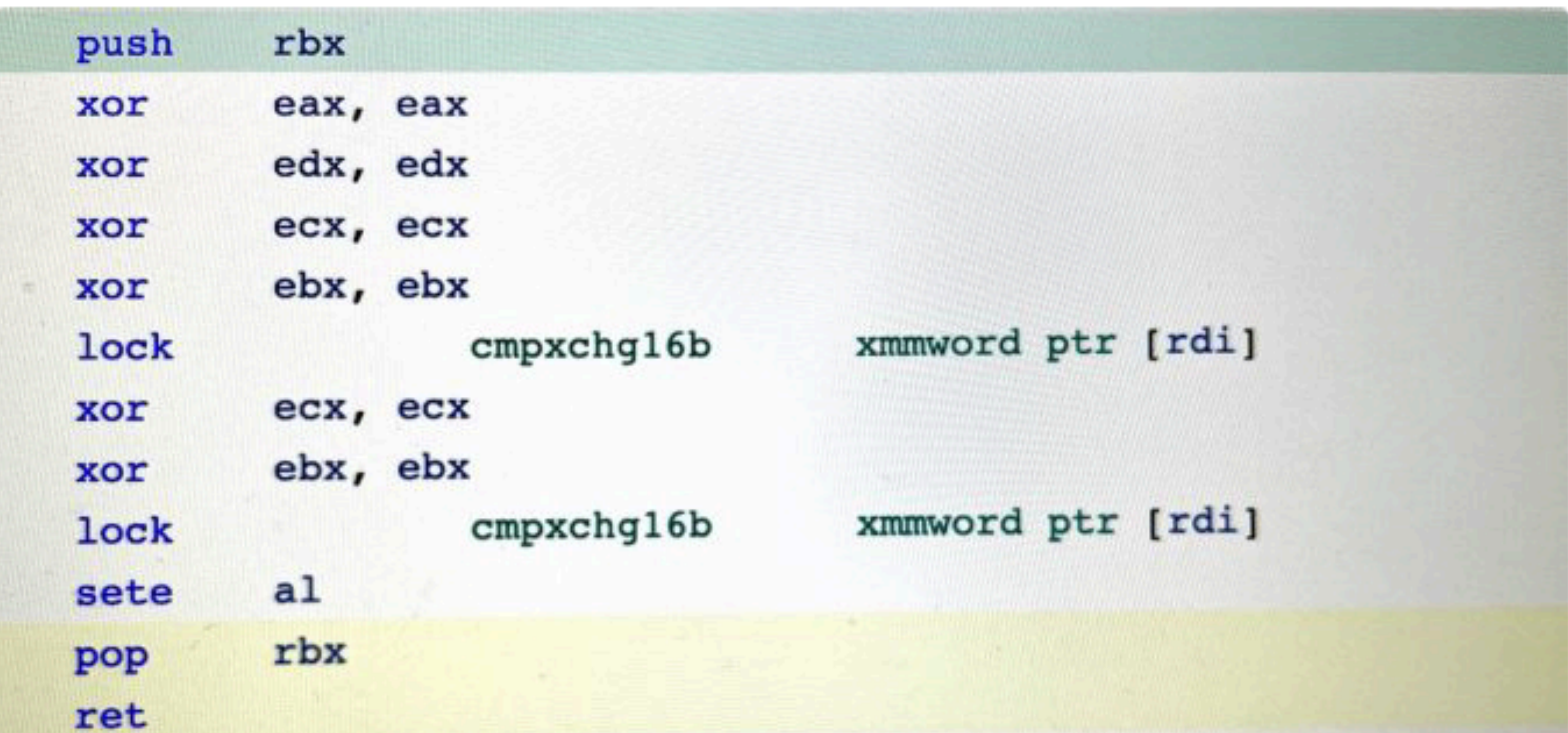
**my suggested solution:**

- **move allocation from atomic_shared_ptr to shared_ptr constructor**
- **compute ptrdiff between object ptr and aliased ptr**
  - **if it fits into 32 bytes (on x64):**
    - **just store that directly**
  - **otherwise:**
    - **use extension list (will happen very rarely!)**
  - **add 1 bit flag to indicate method**

# lock-free atomic shared_ptr?

# DWCAS?




# DWCAS in C++

31 MARCH 2022 / TIMUR DOUMLER / 2 COMMENTS

Implementing your own lock-free data structures using standard C++ isn't something you should attempt unless you really, really know what you are doing (this article summarises why). But you can never become an expert in something if you don't try, so I went in and did it anyway. Among the stumbling blocks I found there was one that I found particularly surprising, so I decided to write a blog post about it: what happens if your lock-free data structure relies on DWCAS.

DWCAS (Double-width compare-and-swap) is a CPU instruction performing an atomic compare-and-swap operation on a memory location that's double the native word size. In other words, it gives you 128-bit atomic lock-free variables on a 64-bit CPU. It is available on every x86-64 chip (`test cmpxchg16b` in x86 assembly) apart from some early AMD and Intel Core 2 chips (we're talking 2008

Search … SEARCH

RECENT POSTS

- DWCAS in C++
- Trip report: C++ Siberia 2020
- Using locks in real-time audio processing, safely
- Trip report: February 2020 ISO C++ committee meeting, Prague
- How to make a container from a C++20 range

RECENT COMMENTS

- Timur Doumler on DWCAS in C++
- Jim Cownie on DWCAS in C++
- How to iterate over a range-v3 action? – Windows Questions on How to make a container from a C++20 range
- How to iterate over a range-v3 action? –

**DWCAS?**

**x86: cmpxchg8b**

**x64: cmpxchg16b**

**armv7/arm64: LL/SC**

# DWCAS?

**x86: cmpxchg8b**

**x64: cmpxchg16b** except some old AMD chips (<= 2008)

**armv7/arm64: LL/SC**

# DWCAS performance?

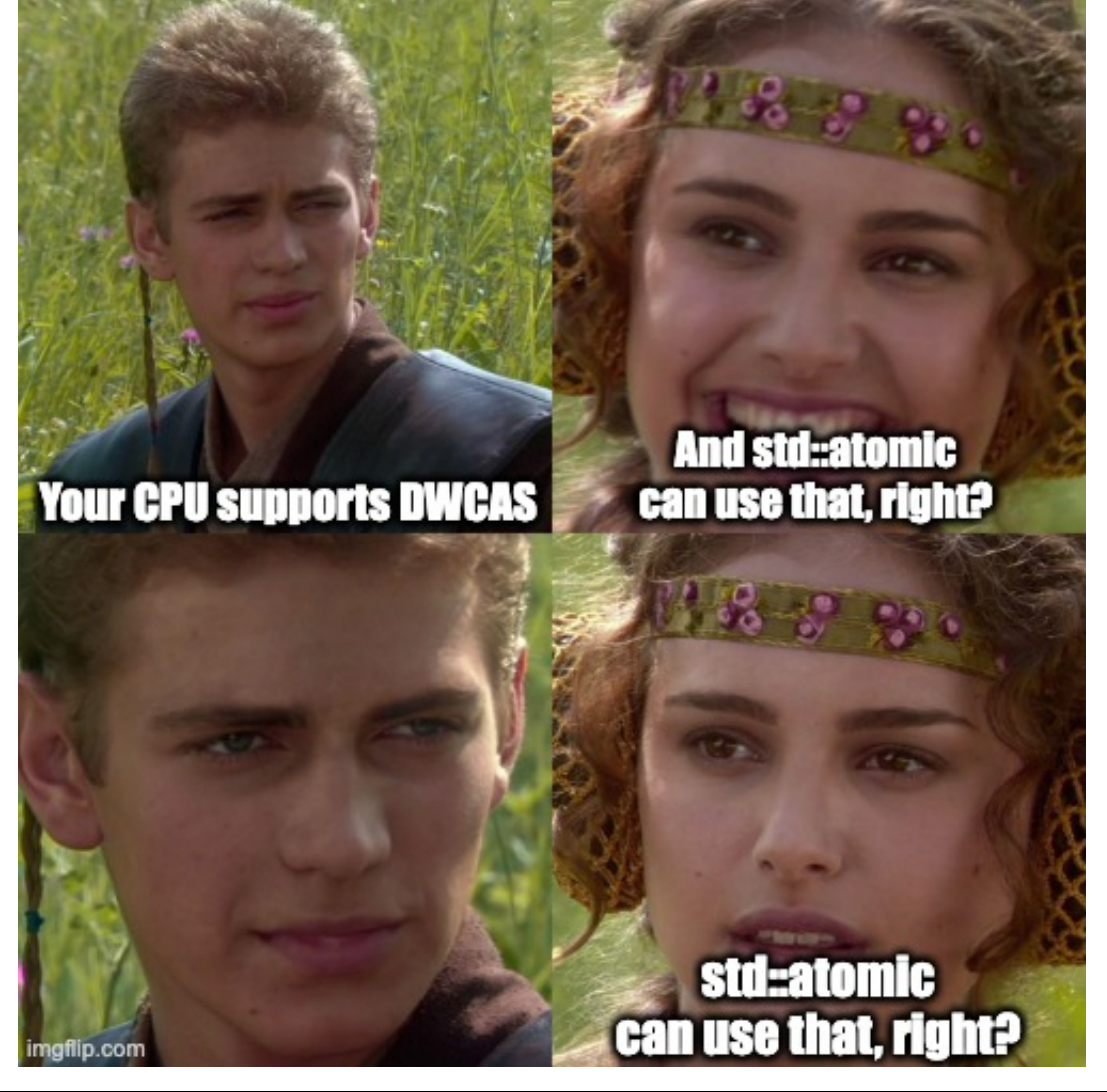
**DWCAS performance?**

**~ 2x slower than CAS on x64**

**~ 1.5x slower than CAS on arm64**

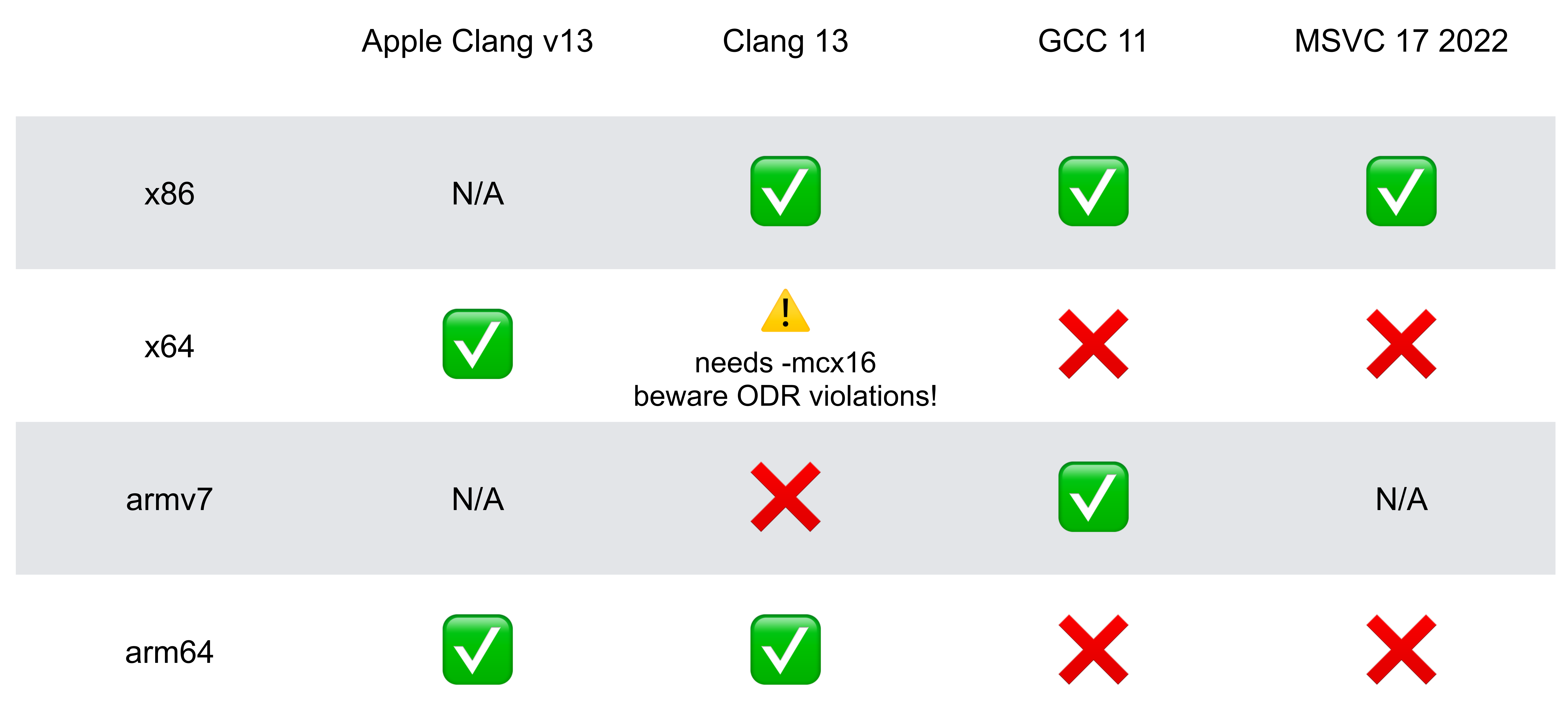
# DWCAS with std::atomic?

```
template<typename T>
class atomic<shared_ptr<T>> {
    struct alignas(2 * sizeof(void*)) counted_ptr {
        control_block* cb_ptr;
        half_uintptr_t local_refcount;
        half_uintptr_t alias;
    };

    atomic<counted_ptr> atomic_cptr;
    static_assert(decltype(value)::is_always_lock_free);
};
```




**me in 2022**

VIA 9GAG.COM
Wrong.



| | Apple Clang v13 | Clang 13 | GCC 11 | MSVC 17 2022 |
|---|---|---|---|---|
| x86 | N/A | ✅ | ✅ | ✅ |
| x64 | ✅ | ⚠️ needs -mcx16 beware ODR violations! | ❌ | ❌ |
| armv7 | N/A | ❌ | ✅ | N/A |
| arm64 | ✅ | ✅ | ❌ | ❌ |

Copyright (c) Timur Doumler | @timur_audio | https://timur.audio

## possible solutions

- **implement our own std::atomic**

## possible solutions

- **implement our own std::atomic**
- **don’t use DWCAS**

## possible solutions

- **implement our own std::atomic**
- **don’t use DWCAS**
  - **“packed pointer”**
    - **used by vtyulb, folly**
    - **64 bit only**

**x64 actually uses a 48-bit address space (4-level paging, 256 TB):**

```
aaaaaaaa|aaaaaaaa|abcdefgh|abcdefgh|abcdefgh|abcdefgh|abcdefgh|abcde000
```

unused bits

alignment bits

# Memory Layout on AArch64 Linux

Author: Catalin Marinas <catalin.marinas@arm.com>

This document describes the virtual memory layout used by the AArch64 Linux kernel. The architecture allows up to 4 levels of translation tables with a 4KB page size and up to 3 levels with a 64KB page size.

AArch64 Linux uses either 3 levels or 4 levels of translation tables with the 4KB page configuration, allowing 39-bit (512GB) or 48-bit (256TB) virtual addresses, respectively, for both user and kernel. With 64KB pages, only 2 levels of translation tables, allowing 42-bit (4TB) virtual address, are used but the memory layout is the same.

ARMv8.2 adds optional support for Large Virtual Address space. This is only available when running with a 64KB page size and expands the number of descriptors in the first level of translation.

User addresses have bits 63:48 set to 0 while the kernel addresses have the same bits set to 1. TTBRx selection is given by bit 63 of the virtual address. The swapper_pg_dir contains only kernel (global) mappings while the user pgd contains only user (non-global) mappings. The swapper_pg_dir address is written to TTBR1 and never written to TTBR0.

```
template<typename T>
class atomic<shared_ptr<T>> {
    struct atomic_packed_ptr {
        atomic<uintptr_t> value;
        static_assert(decltype(value)::is_always_lock_free);  // succeeds without DWCAS :)

        static constexpr uintptr_t offset = 16;
        static constexpr uintptr_t mask = 0x0000'0000'0000'FFFF;

        T* get_ptr() const noexcept {
            return value.load() >> offset;
        }

        uint16_t get_count() const noexcept{
            return value.load() & mask;
        }
    };

    // ...
};
```

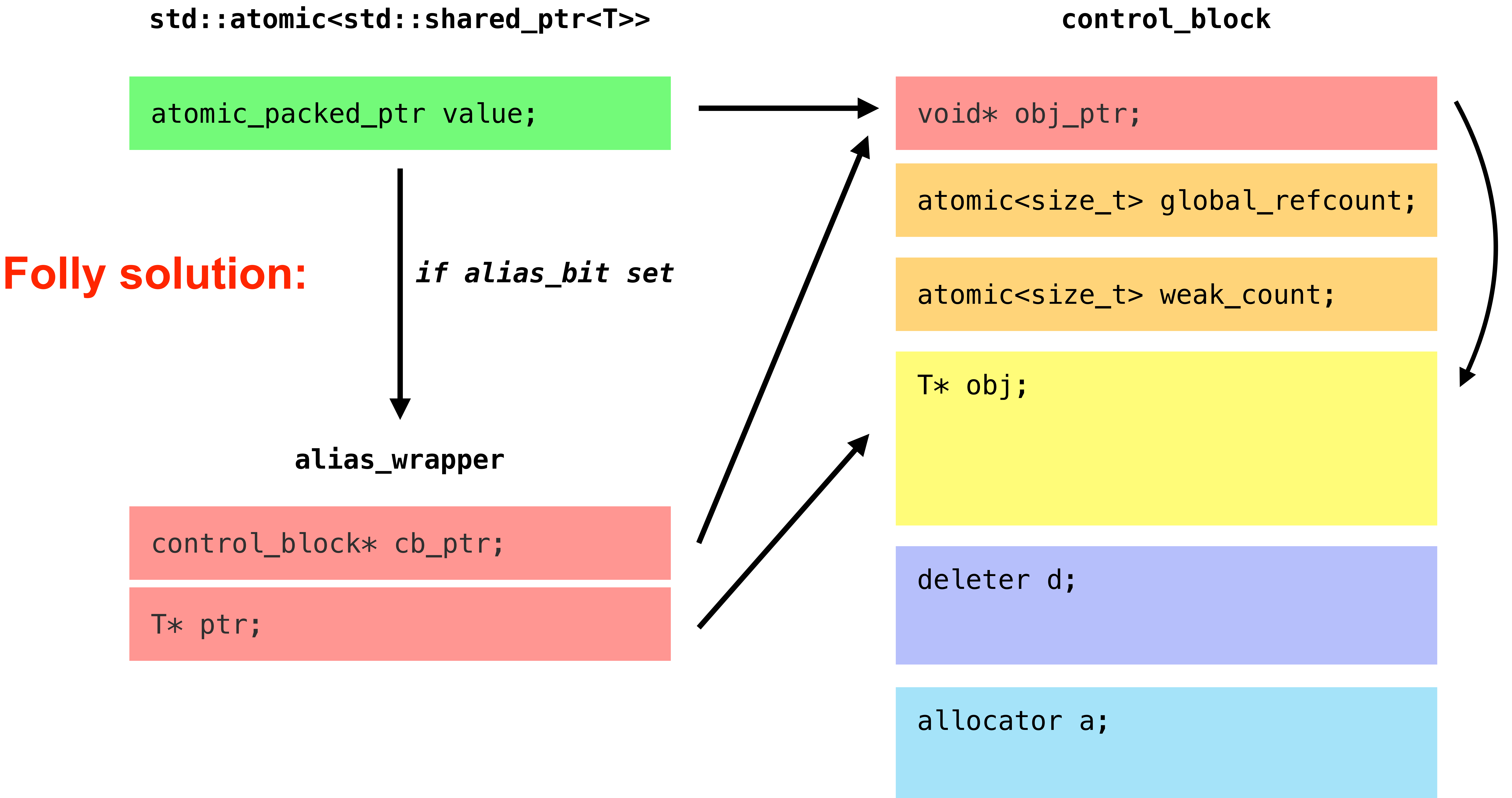
```cpp
template<typename T>
class atomic<shared_ptr<T>> {
    struct atomic_packed_ptr {
        atomic<uintptr_t> value;
        static_assert(decltype(value)::is_always_lock_free);  // succeeds without DWCAS :)

        static constexpr uintptr_t offset = 16;
        static constexpr uintptr_t mask = 0x0000'0000'0000'FFFF;

        T* get_ptr() const noexcept {
            return value.load() >> offset;
        }

        uint16_t get_count() const noexcept{
            return value.load() & mask;
        }
    };

    // ...
};
```

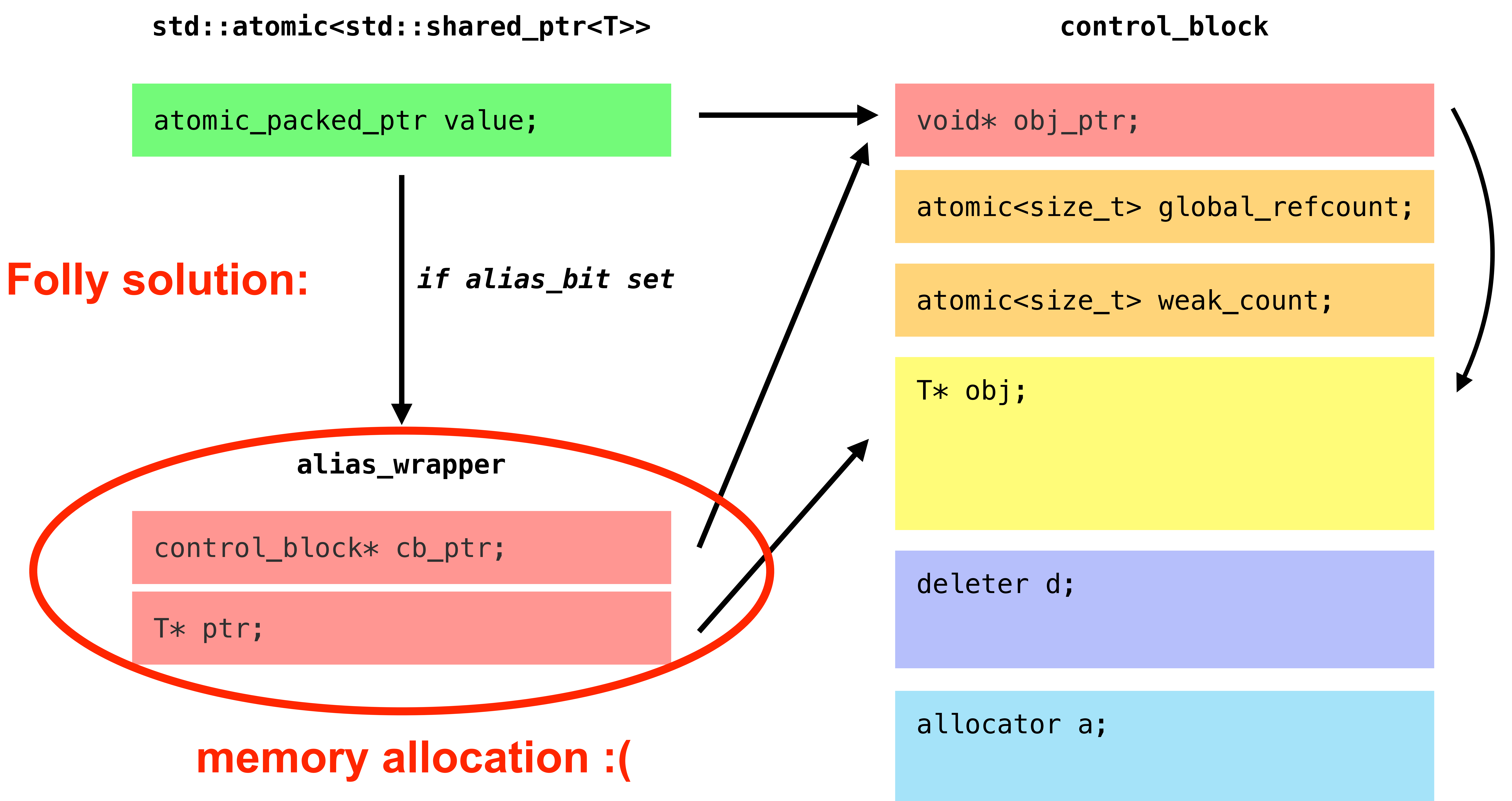
**std::atomic<std::shared_ptr<T>>**

**control_block**

```
atomic_packed_ptr value;
```

```
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;
```


std::atomic<std::shared_ptr<T>>
control_block
atomic_packed_ptr value;
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
Folly solution:
if alias_bit set
T* obj;
alias_wrapper
control_block* cb_ptr;
T* ptr;
deleter d;
allocator a;


std::atomic<std::shared_ptr<T>>
atomic_packed_ptr value;
Folly solution:
if alias_bit set
alias_wrapper
control_block* cb_ptr;
T* ptr;
memory allocation :(
control_block
void* obj_ptr;
atomic<size_t> global_refcount;
atomic<size_t> weak_count;
T* obj;
deleter d;
allocator a;

## my suggested solution:

- **move allocation from atomic_shared_ptr to shared_ptr constructor**

## my suggested solution:

- **move allocation from atomic_shared_ptr to shared_ptr constructor**
    - **but that's overhead for every aliased shared_ptr :(**
    - **…and this time we can't get around it**

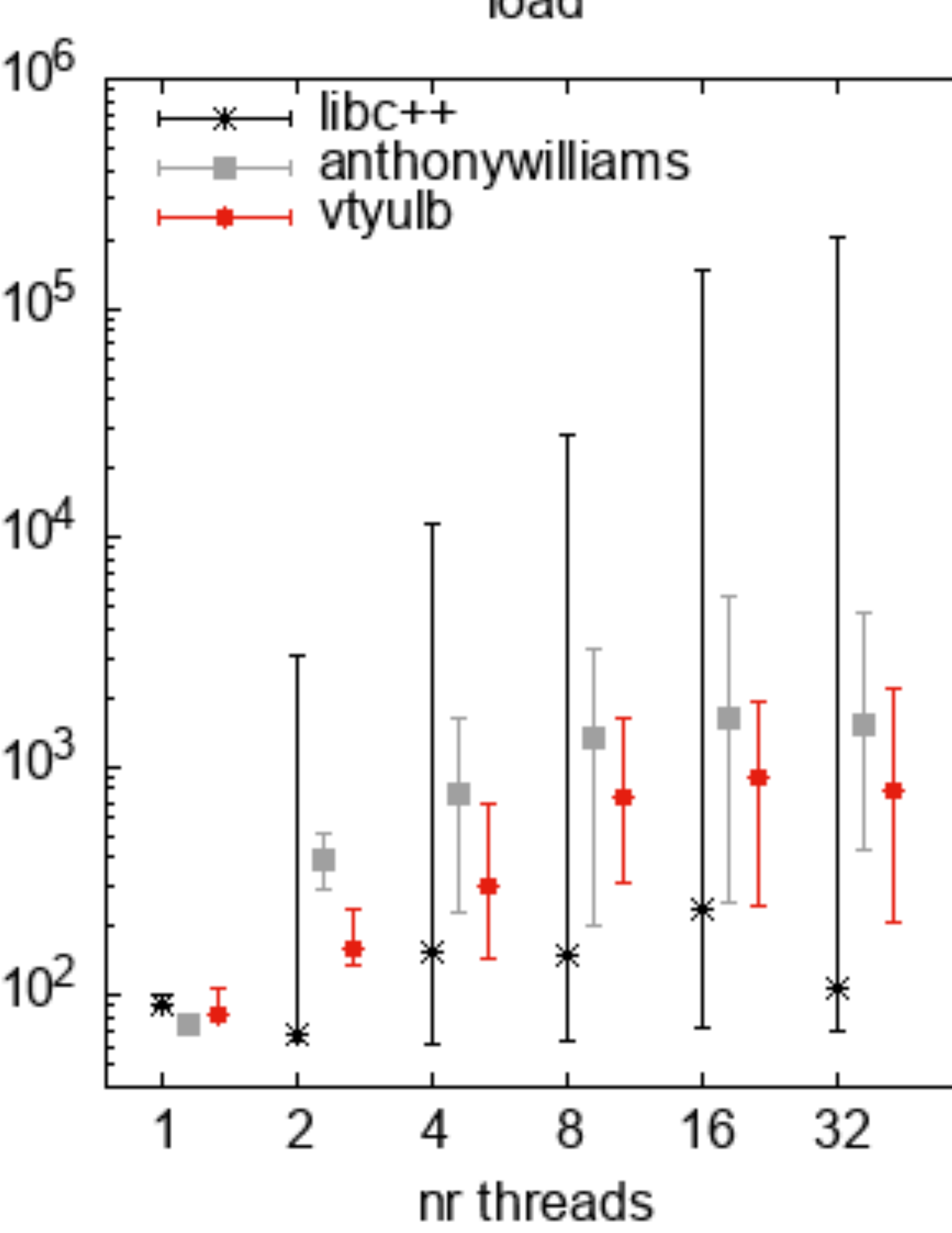
# existing free and open-source implementations

| | anthonywilliams | vtyulb | folly |
|---|---|---|---|
| Portable? | no (requires DWCAS support in std::atomic) | on 64 bit architectures with 48-bit address space | no (requires libstdc++ std::shared_ptr) |
| Lock-free aliased ptrs? | no (can allocate) | no (unimplemented) | no (can allocate) |
| Standard-conforming? | mostly (only a few small things missing) | no (only very partial implementation) | partially (no weak_ptr, etc.) |
| Production-ready? | Proof of concept (Commercial version available) | Proof of concept | yes |

# Benchmarks

```
// Shared:
struct widget {};
atomic<shared_ptr<widget>> aptr = ... ;

// Threads 2 ... N:
{
    shared_ptr<widget> new_ptr = ... ;

    while (true) {
        shared_ptr<widget> old_ptr = aptr.load();
        while (!aptr.compare_exchange_weak(old_ptr, new_ptr))
            /* loop */;
    }
}

// Thread 1:
{
    BENCHMARK(auto ptr = aptr.load());
}
```

# Apple Clang v13



# Apple Clang v13, macOS 12.2.1, x64

# GCC 11, Ubuntu 2021.10, x64

# Takeaways

# Is a lock-free implementation of std::atomic<std::shared_ptr> possible?

## Is a lock-free implementation of std::atomic<std::shared_ptr> possible?

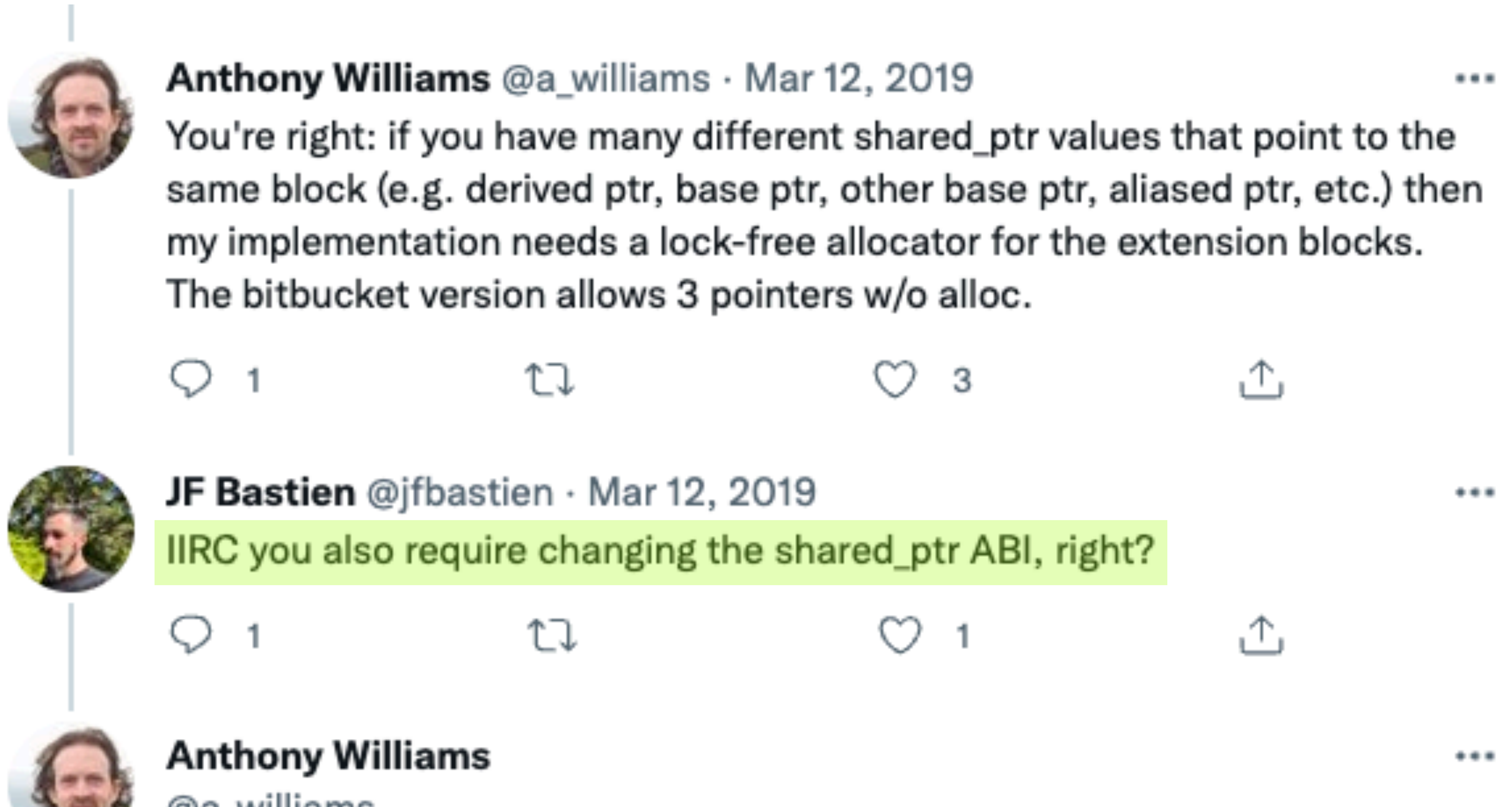
- **yes, if you have DWCAS or free pointer bits**
  - **if no DWCAS:**
    - **nr of free bits determines nr of threads accessing simultaneously**
    - **supporting aliased ptr will cause overhead in shared_ptr ctor**

## Is a lock-free implementation of std::atomic<std::shared_ptr> possible?

- **yes, if you have DWCAS or free pointer bits**
  - **if no DWCAS:**
    - **nr of free bits determines nr of threads accessing simultaneously**
    - **supporting aliased ptr will cause overhead in shared_ptr ctor**

- **conformance vs. performance tradeoffs**
  - **e.g. refcount batching vs. use_count()**

## Is a lock-free implementation of std::atomic<std::shared_ptr> possible in std:: without an ABI break?


Anthony Williams @a_williams · Mar 12, 2019
You're right: if you have many different shared_ptr values that point to the same block (e.g. derived ptr, base ptr, other base ptr, aliased ptr, etc.) then my implementation needs a lock-free allocator for the extension blocks. The bitbucket version allows 3 pointers w/o alloc.
1
3
JF Bastien @jfbastien · Mar 12, 2019
IIRC you also require changing the shared_ptr ABI, right?
1
1
Anthony Williams
@a_williams
Replying to @jfbastien @timur_audio and @StephanTLavavej
Yes, the shared_ptr block needs to handle the split reference count. The basic algorithm is at bitbucket.org/anthonyw/atomi…
6:35 PM · Mar 12, 2019 · Hootsuite Inc.

## Is a lock-free implementation of std::atomic<std::shared_ptr> possible in std:: without an ABI break?

- **In general, yes (folly uses libstdc++ std::shared_ptr)**

## Future work

- **Better benchmarks**
- **My own implementation (work in progress!)**
  - **switchable between DWCAS and pointer-packing**
  - **actually lock-free: non-allocating store of aliased ptr**
  - **experiment further with refcount batching and other optimisations**
  - **publish on github.com/timuraudio**
  - **long-term potential goal: own impl of atomic with DWCAS support**

# existing free and open-source implementations

| | anthonywilliams | vtyulb | folly |
|---|---|---|---|
| Portable? | no (requires DWCAS support in std::atomic) | on 64 bit architectures with 48-bit address space | no (requires libstdc++ std::shared_ptr) |
| Lock-free aliased ptrs? | no (can allocate) | no (unimplemented) | no (can allocate) |
| Standard-conforming? | mostly (only a few small things missing) | no (only very partial implementation) | partially (no weak_ptr, etc.) |
| Production-ready? | proof of concept (commercial version available) | proof of concept | yes |

**work in progress :)**

| | anthonywilliams | vtyulb | folly | timuraudio |
|---|---|---|---|---|
| Portable? | no (requires DWCAS support in std::atomic) | on 64-bit architectures with 48-bit address space | no (requires libstdc++ std::shared_ptr) | yes (can choose DWCAS vs. pointer-packing) |
| Lock-free aliased ptrs? | no (can allocate) | no (unimplemented) | no (can allocate) | yes |
| Standard-conforming? | mostly (only a few small things missing) | no (only very partial implementation) | partially (no weak_ptr, etc.) | yes |
| Production-ready? | proof of concept (commercial version available) | proof of concept | yes | hopefully one day :) |

# A lock-free atomic shared_ptr

Version 0.1


Timur Doumler

@timur_audio


ACCU Conference
6 April 2022

Milky Way above Lava Flow Trail at
Coconino National Forest, CA, USA
Image by Deborah Lee Soltesz