Tutorial 03 - Debugging & shared_ptr

Warm up - Spot the bugs (Section 1)

```
void reader(int* foo) {
    std::cout << *foo;
    delete foo;
}

void writer(int* foo) {
    (*foo)++;
    delete foo;
}

void schedule_unsafe() {
    int* foo = new int;

    std::thread { reader, foo }.detach();
    std::thread { writer, foo }.detach();
}
```

1. Double Free (ASan)

```
void reader(int* foo) {
    std::cout << *foo;
    delete foo;
}

void writer(int* foo) {
    (*foo)++;
    delete foo;
}

void schedule_unsafe() {
    int* foo = new int;

    std::thread { reader, foo }.detach();
    std::thread { writer, foo }.detach();
}
```

```
==================================================================
==39903==ERROR: AddressSanitizer: attempting double-free on 0x
    #0 0x5600620a5d0a in operator delete(void*)
    (truncated)

freed by thread T1 here:
    #0 0x5600620a5d0a in operator delete(void*)
    (truncated)

SUMMARY: AddressSanitizer: double-free in operator delete(void
    (truncated)
```

2. Use After Free (ASan)

```
void reader(int* foo) {
    std::cout << *foo;
    delete foo;
}

void writer(int* foo) {
    (*foo)++;
    delete foo;
}

void schedule_unsafe() {
    int* foo = new int;

    std::thread { reader, foo }.detach();
    std::thread { writer, foo }.detach();
}
```

```
==================================================================
==39903==ERROR: AddressSanitizer: heap-use-after-free on addre
  at pc 0x5600620a8209 bp 0x7f69578fdd20 sp 0x7f69578fdd18
READ of size 4 at 0x602000000010 thread T2
    #0 0x5600620a8208 in writer(int*)
    (truncated)

0x602000000010 is located 0 bytes inside of 4-byte region [0x6
freed by thread T1 here:
    #0 0x5600620a5d0a in operator delete(void*)
    (truncated)

SUMMARY: AddressSanitizer: heap-use-after-free in writer(int*)
    (truncated)

==================================================================
```

3. Uninitialized variable (MSan)

```
void reader(int* foo) {
    std::cout << *foo;
    delete foo;
}

void writer(int* foo) {
    (*foo)++;
    delete foo;
}

void schedule_unsafe() {
    int* foo = new int;

    std::thread { reader, foo }.detach();
    std::thread { writer, foo }.detach();
}
```

https://i.imgur.com/3wlxtI0.gifv

std::shared_ptr

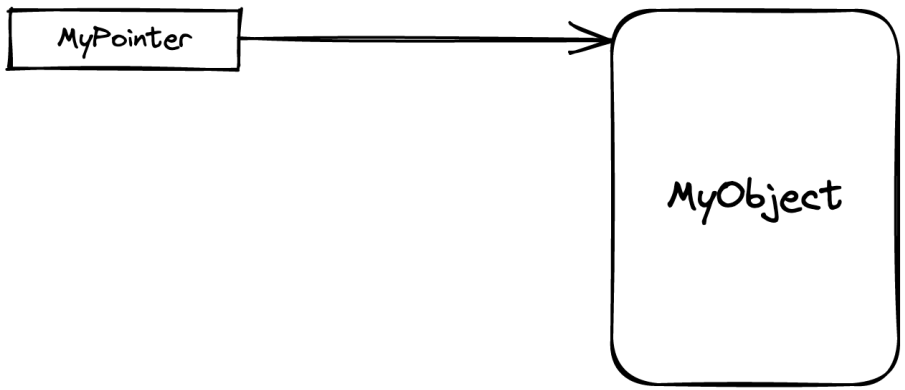
Why use it?
How does it solve the previous problems?

std::shared_ptr

Reference Counting
Section 2.1

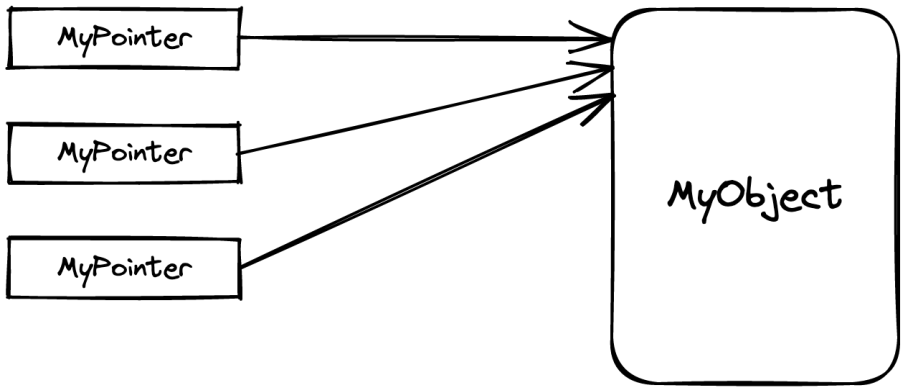
What’s Reference Counting?

the object is stored somewhere on the heap



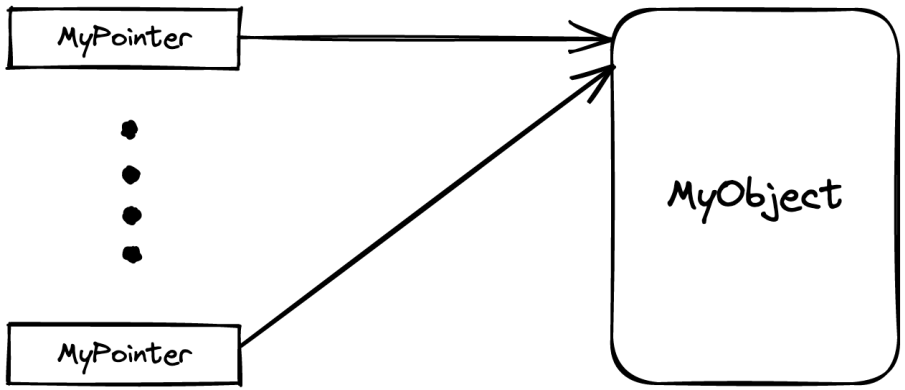
What’s Reference Counting?

multiple pointers can point to the same object



What’s Reference Counting?

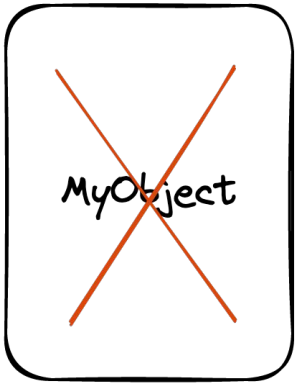
the pointers can get created and destroyed…



What’s Reference Counting?

when the *last pointer* goes away,

the object destructor is called.



What’s Reference Counting?

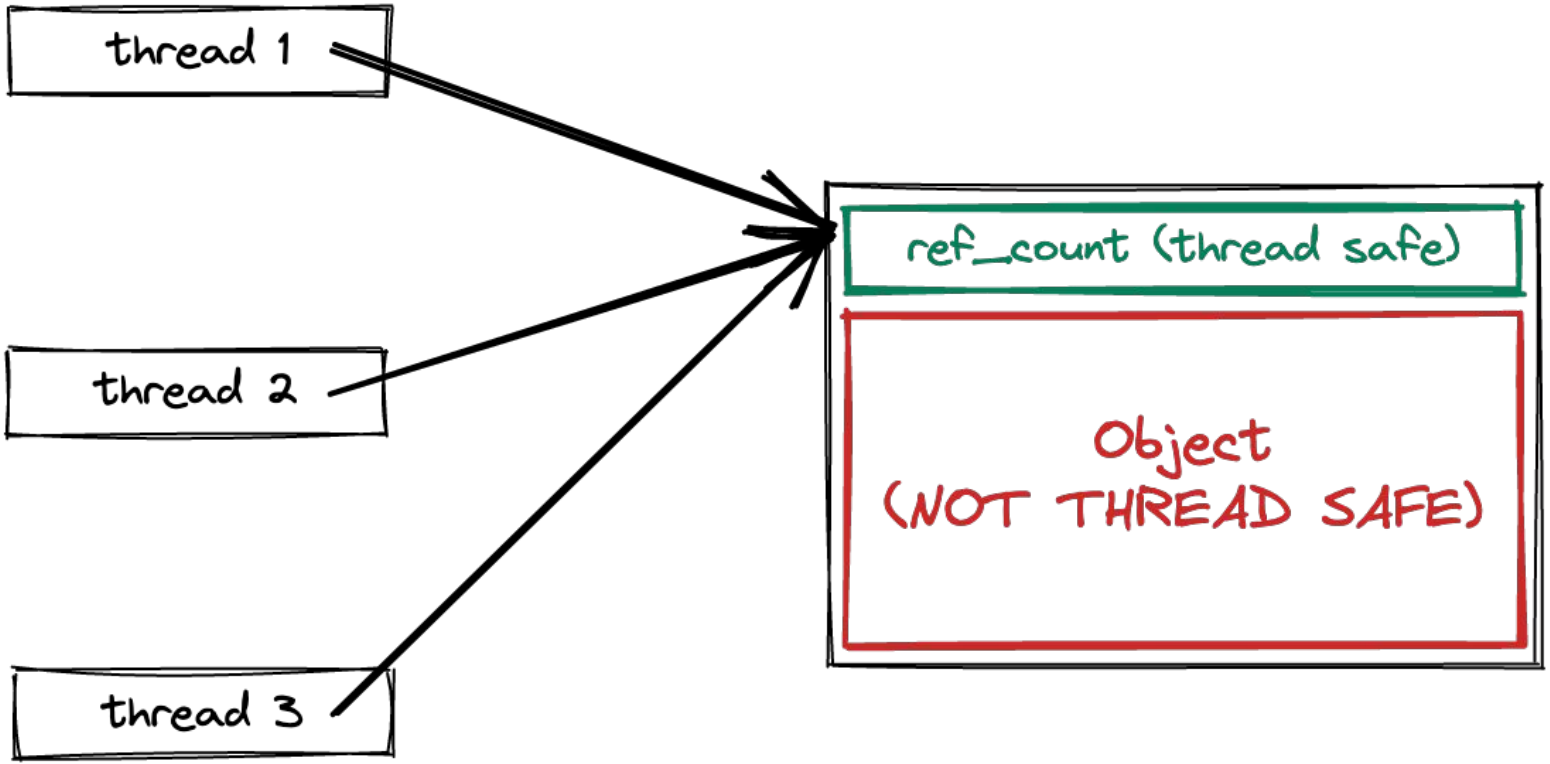
try it yourself

(example 2.1)

4. Is std::shared_ptr thread-safe?

```
void reader2(std::shared_ptr<int> foo) {
    std::cout << *foo;
} // delete foo

void writer2(std::shared_ptr<int> foo) {
    (*foo)++;
} // delete foo

void schedule_safe() {
    std::shared_ptr<int> foo { std::make_shared<int>(0) };

    std::thread { reader2, foo }.detach();
    std::thread { writer2, foo }.detach();
} // delete foo
```

Is std::shared_ptr thread-safe?



Solution

```
void reader2(std::shared_ptr<std::atomic<int>> foo) {
    std::cout << foo->load();
}

void writer2(std::shared_ptr<std::atomic<int>> foo) {
    foo->fetch_add(1);
}

void schedule_safe() {
    auto foo { std::make_shared<std::atomic<int>>(0) };

    std::thread { reader2, foo }.detach();
    std::thread { writer2, foo }.detach();
}
```

std::shared_ptr

The antipattern
(Section 2.4)

Problem 1. Find the bug

```
int main() {
    std::shared_ptr<int> ptr = std::make_shared<int>(0);

    auto reader = std::jthread([](std::shared_ptr<int> ptr) {
    for(int i = 0; i < 100; i++)
        printf("%d\n", *ptr);
    }, ptr);

    auto writer = std::jthread([](std::shared_ptr<int> ptr) {
    for(int i = 0; i < 100; i++)
        *ptr = i;
    }, ptr);
}
```

Problem 1. Data Race on Value (TSan)

```
int main() {
    std::shared_ptr<int> ptr = std::make_shared<int>(0);

    auto reader = std::jthread([](std::shared_ptr<int> ptr) {
    for(int i = 0; i < 100; i++)
        printf("%d\n", *ptr);
    }, ptr);

    auto writer = std::jthread([](std::shared_ptr<int> ptr) {
    for(int i = 0; i < 100; i++)
        *ptr = i;
    }, ptr);
}
```

Recap: What's the fix?

Problem 1. Data Race on Value (TSan)

```cpp
int main() {
    std::shared_ptr<int> ptr = std:

    auto reader = std::jthread([](s
    for(int i = 0; i < 100; i++)
        printf("%d\n", *ptr);
    }, ptr);

    auto writer = std::jthread([](s
    for(int i = 0; i < 100; i++)
        *ptr = i;
    }, ptr);
}
```

```
==================
WARNING: ThreadSanitizer: data race (pid=48593)
  Write of size 4 at 0x7b0800000030 by thread T2:
    #0 func2()::$_3::operator()(std::shared_ptr<int>) const
    (truncated)

  Previous read of size 4 at 0x7b0800000030 by thread T1:
    #0 func2()::$_2::operator()(std::shared_ptr<int>) const
    (truncated)
```

Recap: What's the fix?

Problem 2. Find the bug

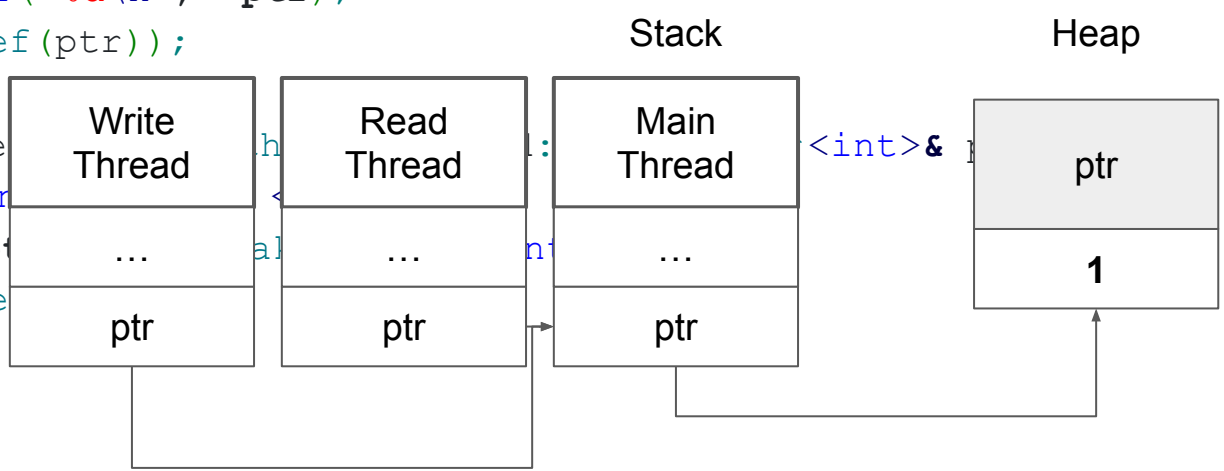
```
int main() {
    std::shared_ptr<int> ptr;

    auto reader = std::jthread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr)
            ;
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto writer = std::jthread([](std::shared_ptr<int>& ptr) {
        for(int i = 0; i < 100; i++)
            ptr = std::make_shared<int>(i);
    }, std::ref(ptr));
}
```

Problem 2. Data Race on Pointer (TSan)

```
int main() {
    std::shared_ptr<int> ptr;

    auto reader = std::jthread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr)
            ;
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto writer = std::jthread([](std::shared_ptr<int>& ptr) {
        for(int i = 0; i < 100; i++)
            ptr = std::make_shared<int>(i);
    }, std::ref(ptr));
}
```

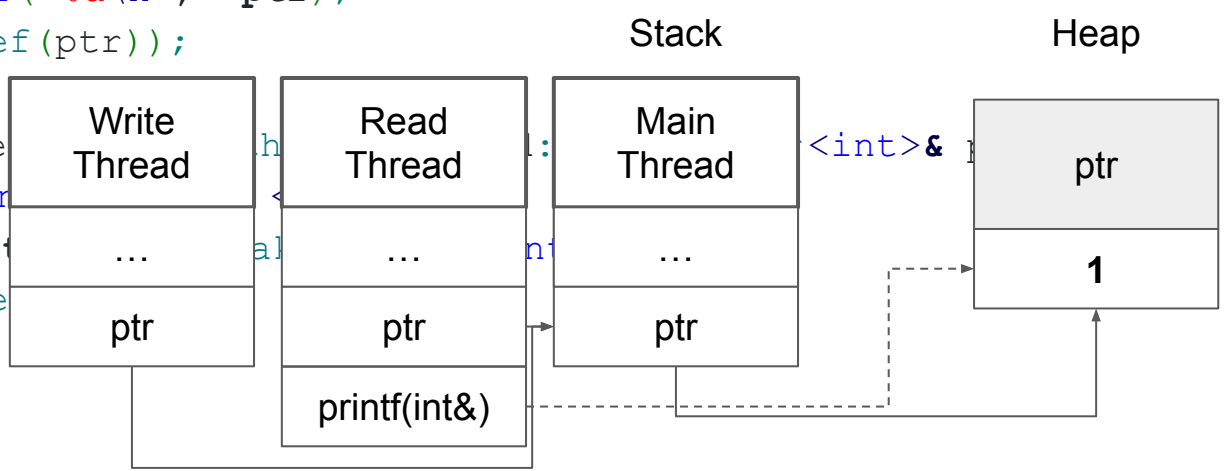
Problem 2. Data Race on Pointer (TSan)

```
int main() {
    std::shared_ptr<int> ptr;

    auto reader = std::jthread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr)
            ;
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto write
        for(in
            pt
    }, std::re
}
```



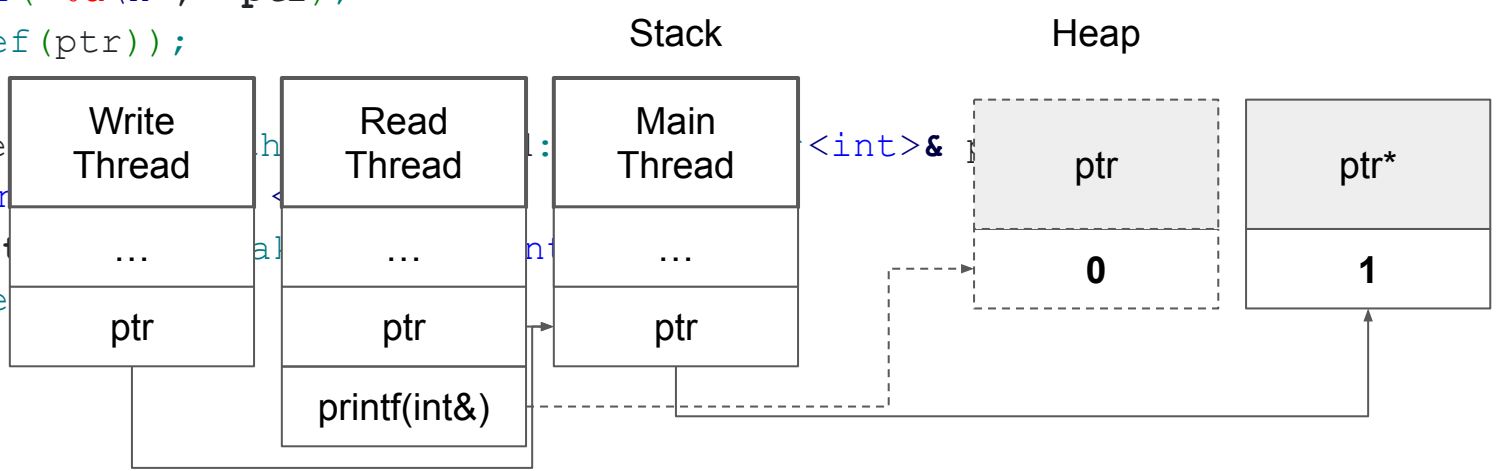
Problem 2. Data Race on Pointer (TSan)

```
int main() {
    std::shared_ptr<int> ptr;

    auto reader = std::jthread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr)
            ;
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto write
        for(in
            pt
    }, std::re
}
```



Problem 2. Data Race on Pointer (TSan)

```
int main() {
    std::shared_ptr<int> ptr;

    auto reader = std::jthread([](std::shared_ptr<int>& ptr) {
        while(ptr == nullptr)
            ;
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto write
        for(in
            pt
    }, std::re
}
```



Problem 2. Data Race on Pointer (TSan)

```cpp
int main() {
    std::shared_ptr<int> ptr;

    auto reader = std::jthread(
        while(ptr == nullptr)
            ;
        printf("%d\n", *ptr);
    }, std::ref(ptr));

    auto writer = std::jthread(
        for(int i = 0; i < 100; i++)
            ptr = std::make_shared<int>(i);
    }, std::ref(ptr));
}
```

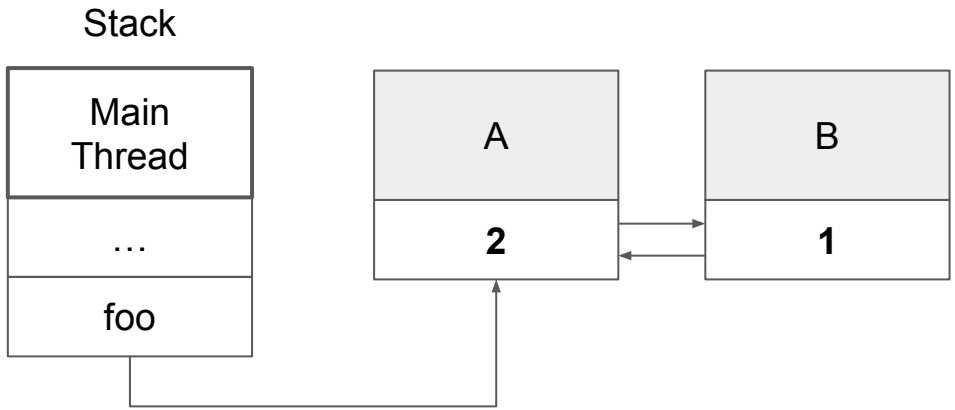
```
==================
WARNING: ThreadSanitizer: data race (pid=48675)
  Write of size 8 at 0x7ffc4917a598 by thread T2:
    (truncated)
    #3 std::shared_ptr<int>::operator=(std::shared_ptr<int>&&)
    (truncated)

  Previous read of size 8 at 0x7ffc4917a598 by thread T1:
    (truncated)
    #1 bool std::operator==<int>(std::shared_ptr<int> const&, std::nullptr_t)
    (truncated)
```

Problem 3. Find the bug

```
// Doubly Linked List
struct DLLNode {
    std::shared_ptr<DLLNode> prev;
    std::shared_ptr<DLLNode> next;
};

struct DLL {
    std::shared_ptr<DLLNode> head {};
    std::shared_ptr<DLLNode> tail {};

    void push_front(std::shared_ptr<DLLNode>);
    void push_back(std::shared_ptr<DLLNode>);

    std::shared_ptr<DLLNode> front();
    std::shared_ptr<DLLNode> back();
};
```

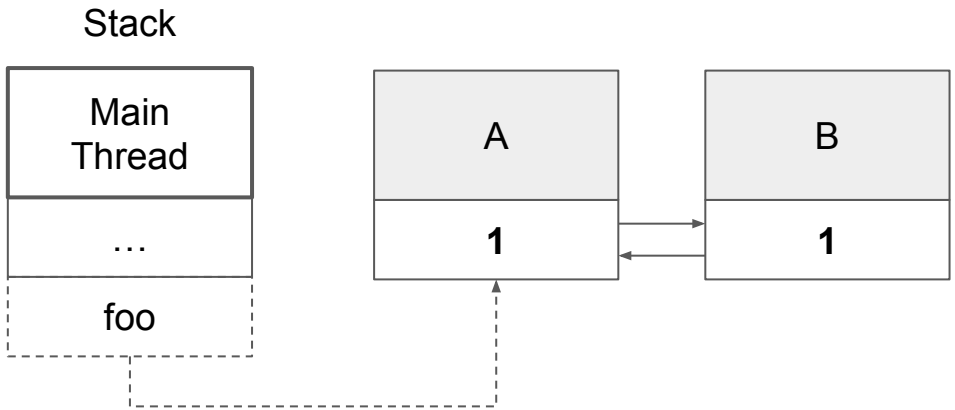
Problem 3. Circular Reference (ASan)

```
// Doubly Linked List
struct DLLNode {
    std::shared_ptr<DLLNode> prev;
    std::shared_ptr<DLLNode> next;
};

struct DLL {
    std::shared_ptr<DLLNode> head {};
    std::shared_ptr<DLLNode> tail {};

    void push_front(std::shared_ptr<DLLNode>);
    void push_back(std::shared_ptr<DLLNode>);

    std::shared_ptr<DLLNode> front();
    std::shared_ptr<DLLNode> back();
};
```



Problem 3. Circular Reference (ASan)

```
// Doubly Linked List
struct DLLNode {
    std::shared_ptr<DLLNode> prev;
    std::shared_ptr<DLLNode> next;
};

struct DLL {
    std::shared_ptr<DLLNode> head {};
    std::shared_ptr<DLLNode> tail {};

    void push_front(std::shared_ptr<DLLNode>);
    void push_back(std::shared_ptr<DLLNode>);

    std::shared_ptr<DLLNode> front();
    std::shared_ptr<DLLNode> back();
};
```



Problem 3. Circular Referenc

```
// Doubly Linked List
struct DLLNode {
    std::shared_ptr<DLLNode> prev;
    std::shared_ptr<DLLNode> next;
};

struct DLL {
    std::shared_ptr<DLLNode> head {};
    std::shared_ptr<DLLNode> tail {};

    void push_front(std::shared_ptr<DL
    void push_back(std::shared_ptr<DLL

    std::shared_ptr<DLLNode> front();
    std::shared_ptr<DLLNode> back();
};
```

```
=================================================================
==1==ERROR: LeakSanitizer: detected memory leaks
Indirect leak of 48 byte(s) in 1 object(s) allocated from:
    #0 0x5639046f60bd in operator new(unsigned long) /root/llvm-pro
    #1 0x5639046f87c0 in std::__new_allocator<std::_Sp_counted_ptr_
    #2 0x5639046f87c0 in std::allocator_traits<std::allocator<std::
    #3 0x5639046f87c0 in std::__allocated_ptr<std::allocator<std::_
    #4 0x5639046f87c0 in std::__shared_count<(__gnu_cxx::_Lock_poli
    #5 0x5639046f87c0 in std::__shared_ptr<DLLNode, (__gnu_cxx::_Lo
    #6 0x5639046f87c0 in std::shared_ptr<DLLNode>::shared_ptr<std::
    #7 0x5639046f87c0 in std::shared_ptr<std::enable_if<!is_array<D
    #8 0x5639046f87c0 in main /app/example.cpp:28:11
    #9 0x7fd5fa270082 in __libc_start_main (/lib/x86_64-linux-gnu/l

Indirect leak of 48 byte(s) in 1 object(s) allocated from:
    #0 0x5639046f60bd in operator new(unsigned long) /root/llvm-pro
    #1 0x5639046f8710 in std::__new_allocator<std::_Sp_counted_ptr_
    #2 0x5639046f8710 in std::allocator_traits<std::allocator<std::
    #3 0x5639046f8710 in std::__allocated_ptr<std::allocator<std::_
    #4 0x5639046f8710 in std::__shared_count<(__gnu_cxx::_Lock_poli
    #5 0x5639046f8710 in std::__shared_ptr<DLLNode, (__gnu_cxx::_Lo
    #6 0x5639046f8710 in std::shared_ptr<DLLNode>::shared_ptr<std::
    #7 0x5639046f8710 in std::shared_ptr<std::enable_if<!is_array<D
    #8 0x5639046f8710 in main /app/example.cpp:27:11
    #9 0x7fd5fa270082 in __libc_start_main (/lib/x86_64-linux-gnu/l

SUMMARY: AddressSanitizer: 96 byte(s) leaked in 2 allocation(s).
```

Summary of Pitfalls

1. Data race on pointer
2. Data race on value
3. Memory leak due to circular reference

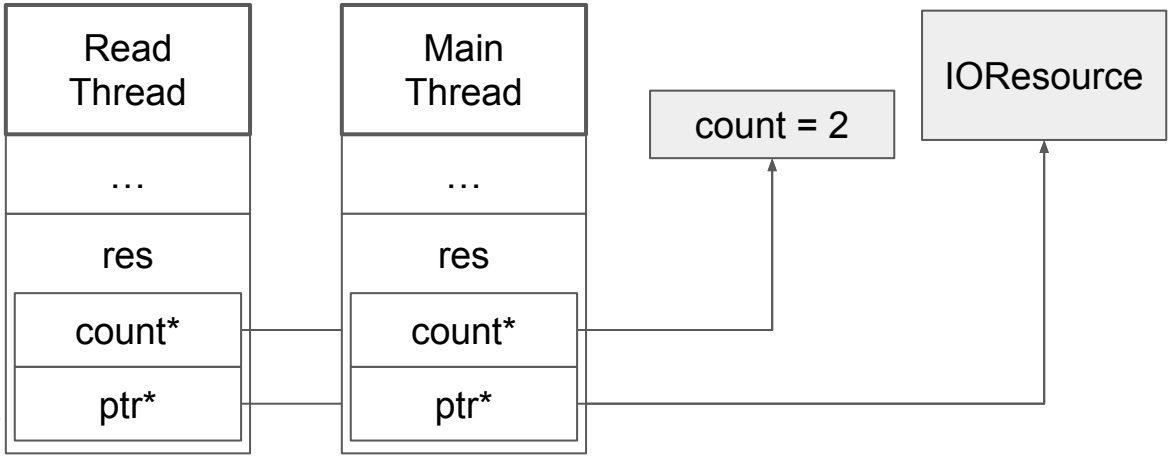
std::shared_ptr

Implementation

1. Start with sequential impl

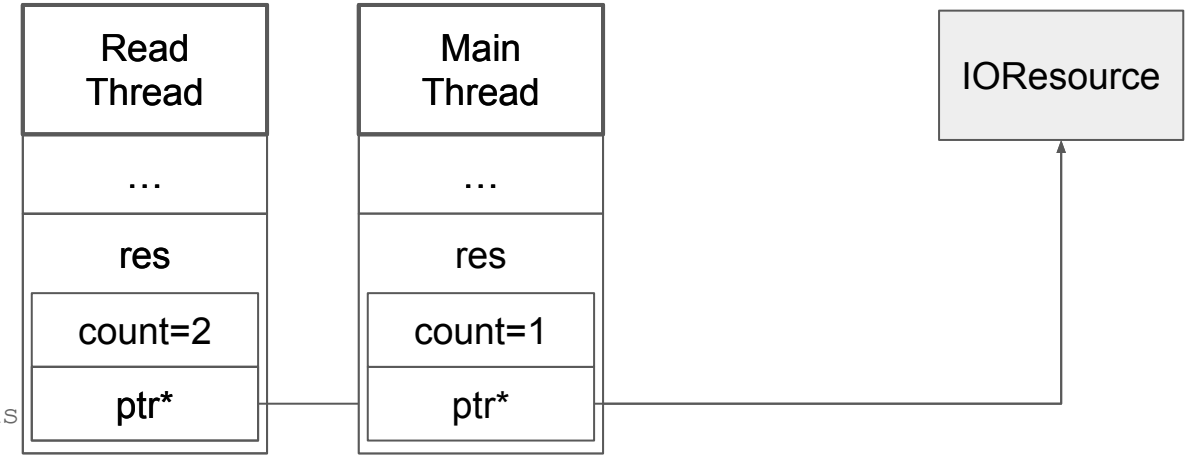
```
template <typename T>
class SharedPtr {
    size t m count;
    T* m ptr;
    // TODO: add additional fields

public:
    SharedPtr(T* ptr) : m_count(1), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        // TODO: synchronise this
        ++m_count;
    }

    ~SharedPtr() {
        // TODO: synchronise this
        if((--m count) == 0)
            delete m_ptr;
};
```

2. Share the count



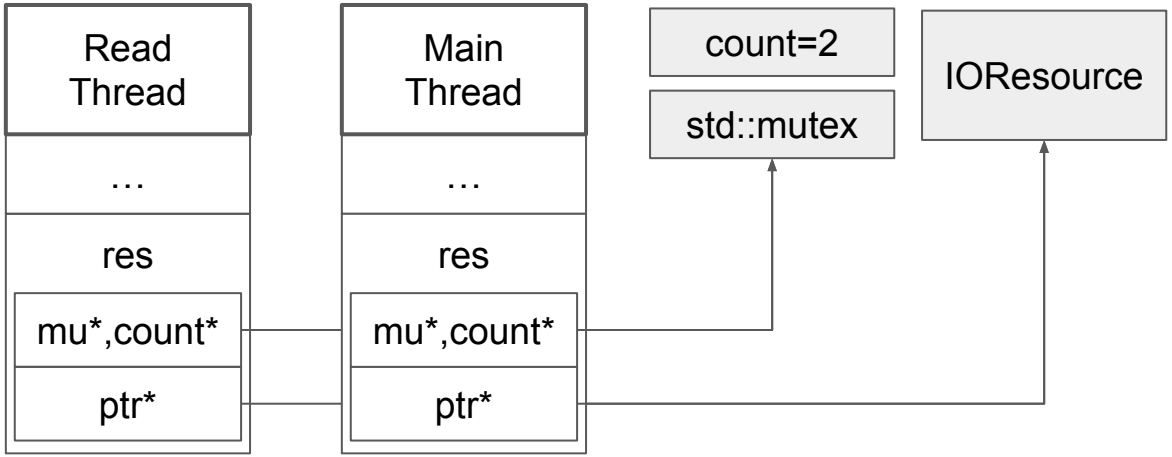

```
template <typename T>
class SharedPtr {
    size_t* m_count;
    T* m_ptr;
    // TODO: add additional fields

public:
    SharedPtr(T* ptr) : m_count(new int(1)), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        // TODO: synchronise this
        ++(*m_count);
    }

    ~SharedPtr() {
        // TODO: synchronise this
        if((--(*m_count)) == 0)
            delete m_ptr;
};
```

2. …vs copy (attempt #1)




```
template <typename T>
class SharedPtr {
    size_t* m_count;
    T* m_ptr;
    // TODO: add additional fields

public:
    SharedPtr(T* ptr) : m_count(new int(1)), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        // TODO: synchronise this
        ++(*m_count);
    }

    ~SharedPtr() {
        // TODO: synchronise this
        if((--(*m_count)) == 0)
            delete m_ptr;
};
```

3. Synchronize count (attempt #2)




```
template <typename T>
class SharedPtr {
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(1)), m_mutex(new std::mutex()), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_mutex(other.m_mutex),
m_ptr(other.m_ptr) {
        auto lk = std::unique_lock { *m_mutex };
        ++(*m_count);
    }

    ~SharedPtr() {
        auto lk = std::unique_lock { *m_mutex };
        if(--(*m_count) == 0) {
            delete m_ptr; delete m_mutex; delete m_count;
        }
    }
};
```

4. Safety?

```
template <typename T>
class SharedPtr {
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(1)), m_mutex(new std::mutex()), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_mutex(other.m_mutex),
m_ptr(other.m_ptr) {
        auto lk = std::unique_lock { *m_mutex };
        ++(*m_count);
    }

    ~SharedPtr() {
        auto lk = std::unique lock { *m_mutex };
        if(--(*m_count) == 0) {
            delete m_ptr; delete m_mutex; delete m_count;
        }
    }
};
```

4. Safety?

```
template <typename T>
class SharedPtr {
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(1)), m_mutex(new std::mutex()), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_mutex(other.m_mutex),
m_ptr(other.m_ptr) {
        auto lk = std::unique_lock { *m_mutex };
        ++(*m_count);
    }

    ~SharedPtr() {
        auto lk = std::unique_lock { *m_mutex };
        if(--(*m_count) == 0) {
            delete m_ptr; delete m_mutex; delete m_count;
        }
    } // m_mutex.unlock()
};
```

Use after free!

4. Safety?

```
template <typename T>
class SharedPtr {
    size t* m count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(

    SharedPtr(const SharedPtr& other) : m_c
m_ptr(other.m ptr) {
        auto lk = std::unique_lock { *m_m
        ++(*m_count);
    }

    ~SharedPtr() {
        auto lk = std::unique_lock { *m_m
        if(--(*m count) == 0) {
            delete m_ptr; delete m_mutex; delete m_count;
        }
    } // m_mutex.unlock()
};
```

```
==================
WARNING: ThreadSanitizer: heap-use-after-free (pid=1)
  Atomic read of size 1 at 0x7b0c00000030 by thread T9 (mutexes: write M0):
    #0 <null> <null> (output.s+0x709fa)
    #1 <null> <null> (output.s+0xd3add)
    #2 <null> <null> (output.s+0xd387e)
    #3 <null> <null> (libstdc++.so.6+0xe0a3b) (BuildId: 563574e8434b929f2596a1

  Previous write of size 8 at 0x7b0c00000030 by thread T9 (mutexes: write M0):
    #0 <null> <null> (output.s+0xd2b6e)
    #1 <null> <null> (output.s+0xd3ab0)
    #2 <null> <null> (output.s+0xd387e)
    #3 <null> <null> (libstdc++.so.6+0xe0a3b) (BuildId: 563574e8434b929f2596a1

  Mutex M0 (0x7b0c00000030) created at:
    #0 <null> <null> (output.s+0x7085a)
    #1 <null> <null> (output.s+0xd3762)
    #2 <null> <null> (libc.so.6+0x24082) (BuildId: 1878e6b475720c7c51969e69ab2

  Thread T9 (tid=11, running) created by thread T8 at:
    #0 <null> <null> (output.s+0x52b7d)
    #1 <null> <null> (libstdc++.so.6+0xe0d3b) (BuildId: 563574e8434b929f2596a1
    #2 <null> <null> (output.s+0xd39ae)
    #3 <null> <null> (libstdc++.so.6+0xe0a32) (BuildId: 563574e8434b929f2596a1

SUMMARY: ThreadSanitizer: heap-use-after-free (/app/output.s+0x709fa)
==================
```

Use after free!

4. Unlock before Delete

```
template <typename T>
class SharedPtr {
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(1)), m_mutex(new std::mutex()), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_mutex(other.m_mutex), m_ptr(other.m_ptr) {
            auto lk = std::unique_lock { *m_mutex };
            ++(*m_count);
    }

    ~SharedPtr() {
            size_t new_count = [this]() {
                  auto lk = std::unique_lock { *m_mutex };
                  return --(*m_count);
            }(); // m_mutex.unlock()
            if(new_count == 0) {
                delete m_ptr; delete m_mutex; delete m_count;
            }

    }
};
```

Can we do better?

```
template <typename T>
class SharedPtr {
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(1)), m_mutex(new std::mutex()), m_ptr(ptr) {} // 2 alloc

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_mutex(other.m_mutex), m_ptr(other.m_ptr) {
            auto lk = std::unique_lock { *m_mutex }; // 1st cache miss + syscall
            ++(*m_count);                            // 2nd cache miss
    }                                                // 2nd syscall

    ~SharedPtr() {
            size_t new_count = [this]() {
                  auto lk = std::unique_lock { *m_mutex }; // 1st cache miss + syscall
                  return --(*m_count);                     // 2nd cache miss
            }();                                           // 2nd syscall
            if(new_count == 0) {
                delete m_ptr; delete m_mutex; delete m_count; // whee!
            }

    }
};
```

Can we do better?

```
template <typename T>
class SharedPtr {
    size_t* m_count;
    std::mutex* m_mutex;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new size_t(1)), m_mutex(new std::mutex()), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_mutex(other.m_mutex), m_ptr(other.m_ptr) {
            auto lk = std::unique_lock { *m_mutex };
            ++(*m_count);
    }

    ~SharedPtr() {
            size_t new_count = [this]() {
                  auto lk = std::unique_lock { *m_mutex };
                  return --(*m_count);
            }();
            if(new_count == 0) {
                delete m_ptr; delete m_mutex; delete m_count;
            }

    }
};
```

When m_count = 1, only 1 shared ptr exists, no concurrency i.e. no copy constructor run concurrently in the middle of destructor. Hence it is safe to delete

Rewrite It In Atomic 😜

```
template <typename T>
class SharedPtr {
    std::atomic<size_t>* m_count;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new std::atomic<size_t>(1)), m_ptr(ptr) {}

    SharedPtr(const SharedPtr4& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        m_count->fetch_add(1);
    }

    ~SharedPtr4() {
        size t old count = m_count->fetch_sub(1);
        if(old_count == 1) {
            delete m_ptr; delete m_count;
        }
    }
};
```

What is the weakest memory order we can use for fetch_add and fetch_sub?

How about atomics (how hard can it be)?

```
template <typename T>
class SharedPtr {
    std::atomic<size_t>* m_count;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new std::atomic<size_t>(1)), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        m_count->fetch_add(1, std::memory_order_relaxed);
    }

    ~SharedPtr() {
        size_t old_count = m_count->fetch_sub(1, std::memory_order_relaxed);
        if(old_count == 1) {
            delete m_ptr; delete m_count;
        }
    }
};
```

relaxed?

All threads will agree on the modification order of each individual variable

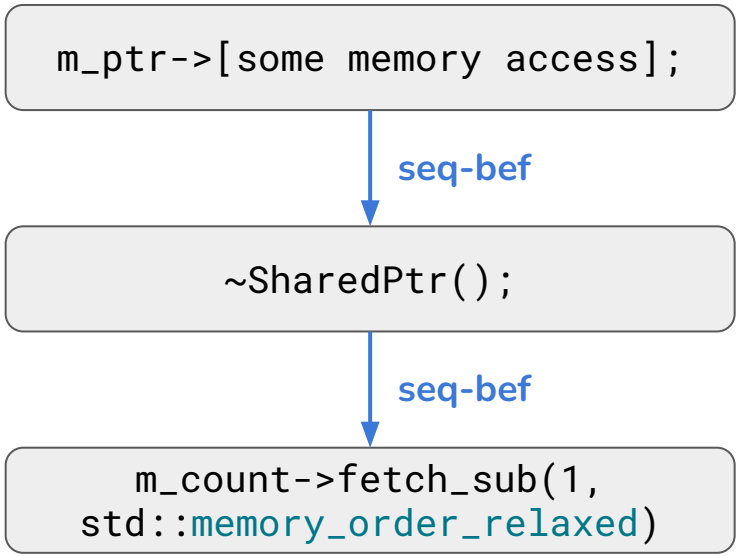
How about atomics (how hard can it be)?

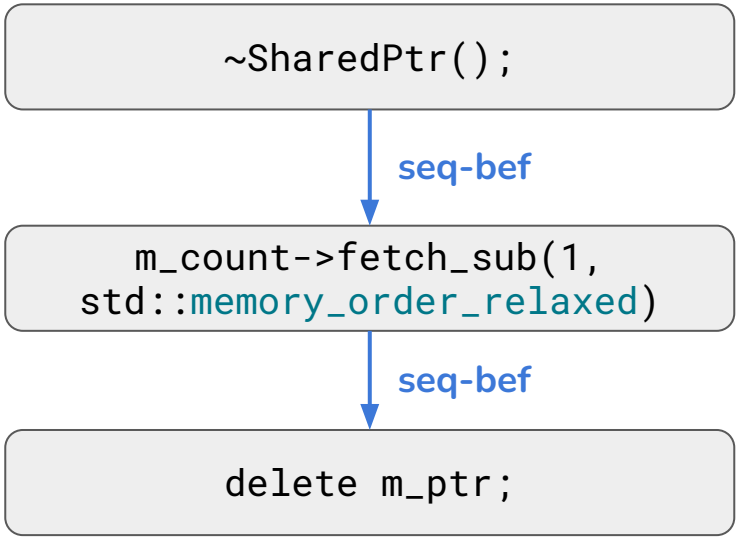
```
template <typename T>
class SharedPtr {
    std::atomic<size_t>* m_count;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new std::atomic<size_t>(1)), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        m_count->fetch_add(1, std::memory_order_relaxed);
    }

    ~SharedPtr() {
        size_t old_count = m_count->fetch_sub(1, std::memory_order_relaxed);
        if(old_count == 1) {
            delete m_ptr; delete m_count;
        }
    }
};
```

relaxed?

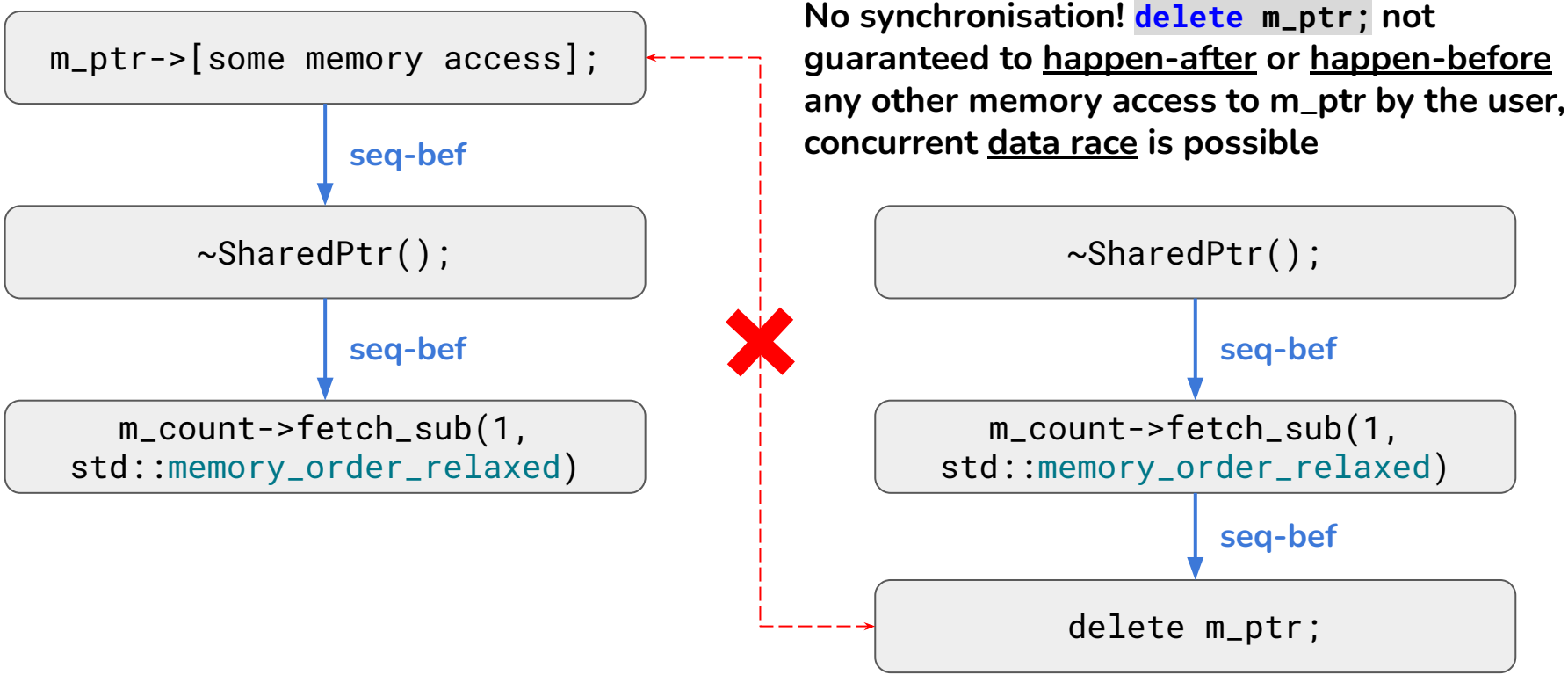
But what about m_count and m_ptr between threads?

How about atomics (how hard can it be)?





How about atomics (how hard can it be)?

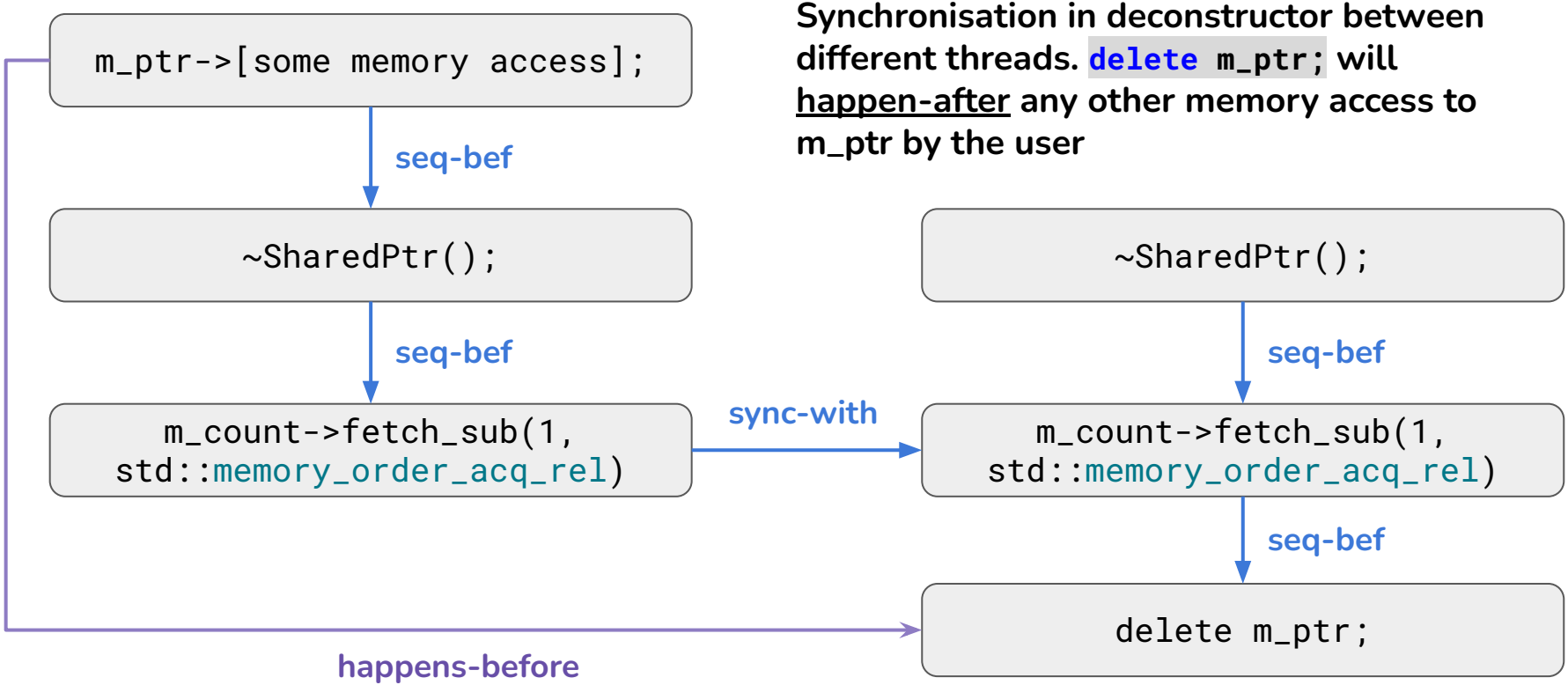


How about atomics (how hard can it be)?

```
template <typename T>
class SharedPtr {
    std::atomic<size_t>* m_count;
    T* m_ptr;

public:
    SharedPtr(T* ptr) : m_count(new std::atomic<size_t>(1)), m_ptr(ptr) {}

    SharedPtr(const SharedPtr& other) : m_count(other.m_count), m_ptr(other.m_ptr) {
        m_count->fetch_add(1, std::memory_order_relaxed);
    }

    ~SharedPtr() {
        size_t old_count = m_count->fetch_sub(1, std::memory_order_acq_rel);
        if(old_count == 1) {
            delete m_ptr; delete m_count;
        }
    }
};
```

Combined acquire release

How about atomics (how hard can it be)?



Synchronisation in deconstructor between different threads. `delete m_ptr;` will happen-after any other memory access to m_ptr by the user

Have a great weekend ahead!