CS3211 Tutorial 3
Debugging Concurrent C++ Programs

Simon

Adapted from slides made by Walter and Kingsley (OP past TAs)

Debugging Tools Scary 🥺

print(“here”)

print(f“what happen {v}”)

Never too late to learn!

Disclaimer

Debugging tools (including the mighty ASan/TSan) cannot find all bugs!

1. Protecting Shared Resources

Spot the bug(s)

```
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)

```
==1650487==ERROR: AddressSanitizer: attempting double-free on 0x602000000010 in thread T2:
    #0 0x55d462bc9a3d in operator delete(void*) /tmp/llvm-project-15.0.7.src/compiler-rt/lib/asan/asan_new_delete.cpp:152:3
    #1 0x55d462bcb7e6 in writer(int*) /tmp/compiler-explorer-compiler202318-1603447-6njzw8.04ew2/example.cpp:16:5
    #2 0x55d462bce4bb in void std::__invoke_impl<void, void (*)(int*), int*>(std::__invoke_other, void (*&&)(int*), int*&&)
    #3 0x55d462bce3fc in std::__invoke_result<void (*)(int*), int*>::type std::__invoke<void (*)(int*), int*>(void (*&&)(int
    #4 0x55d462bce3d1 in void std::thread::_Invoker<std::tuple<void (*)(int*), int*>>::_M_invoke<0ul, 1ul>(std::_Index_tuple
    #5 0x55d462bce394 in std::thread::_Invoker<std::tuple<void (*)(int*), int*>>::operator()() /usr/lib/gcc/x86_64-linux-gnu
    #6 0x55d462bce0b8 in std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)(int*), int*>>>::_M_run() /usr/li
    #7 0x7f318bb626b3  (/lib/x86_64-linux-gnu/libstdc++.so.6+0xda6b3) (BuildId: 537bd518196e70fb9620a43efdc663b946f8bd77)
    #8 0x7f318b91e608 in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x8608) (BuildId: 7b4536f41cdaa5888408e82d0836e
    #9 0x7f318b7fa132 in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x11f132) (BuildId: 1878e6b475720c7c51969e69ab2d276fae6d1d
```

`#1 0x55d462bcb7e6 in writer(int*) .../`**`example.cpp:16:5`**

1. Double Free (ASan)

```cpp
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();
}
```

2. Use After Free (ASan)

```
==1650416==ERROR: AddressSanitizer: heap-use-after-free on address 0x602000000010 at pc 0x555dafafc869 bp 0x7fcc42f
READ of size 4 at 0x602000000010 thread T2
    #0 0x555dafafc868 in writer(int*) /tmp/compiler-explorer-compiler202318-1603447-1oqmrqn.bn1q/example.cpp:15:11
    #1 0x555dafaff56b in void std::__invoke_impl<void, void (*)(int*), int*>(std::__invoke_other, void (*&&)(int*),
    #2 0x555dafaff4ac in std::__invoke_result<void (*)(int*), int*>::type std::__invoke<void (*)(int*), int*>(void
    #3 0x555dafaff481 in void std::thread::_Invoker<std::tuple<void (*)(int*), int*>>::_M_invoke<0ul, 1ul>(std::_In
    #4 0x555dafaff444 in std::thread::_Invoker<std::tuple<void (*)(int*), int*>>::operator()() /usr/lib/gcc/x86_64-
    #5 0x555dafaff168 in std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)(int*), int*>>>::_M_run(
    #6 0x7fcc46d516b3  (/lib/x86_64-linux-gnu/libstdc++.so.6+0xda6b3) (BuildId: 537bd518196e70fb9620a43efdc663b946f
    #7 0x7fcc46b0d608 in start_thread (/lib/x86_64-linux-gnu/libpthread.so.0+0x8608) (BuildId: 7b4536f41cdaa5888408
    #8 0x7fcc469e9132 in __clone (/lib/x86_64-linux-gnu/libc.so.6+0x11f132) (BuildId: 1878e6b475720c7c51969e69ab2d2
```

```
#0 0x555dafafc868 in writer(int*) .../example.cpp:15:11
```

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();
}
```

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();
}
```

See more: https://en.cppreference.com/w/cpp/language/default_initialization

2. std::shared_ptr

Reference Counting

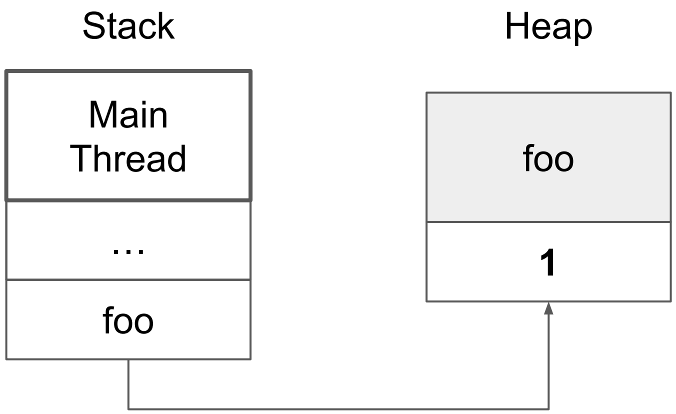
Why are we managing our own memory? 🤔

What's Reference Counting?

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

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

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

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

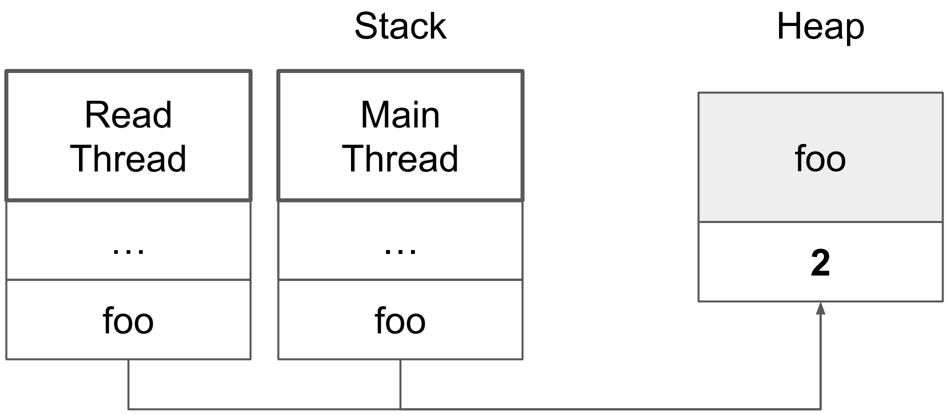
What’s Reference Counting?

```
void schedule_safe() {
    std::shared_ptr<int> foo { std::make_shared<int>(0) }; // count me!

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



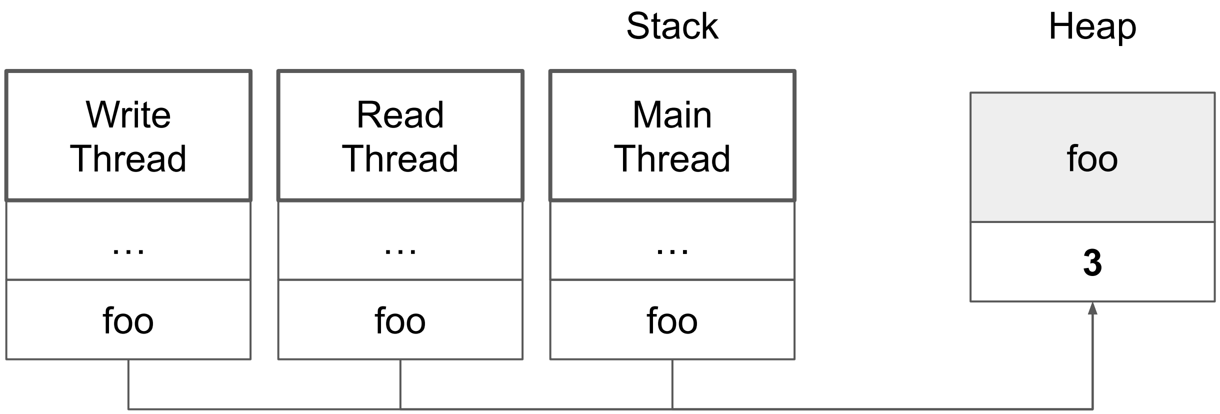
What’s Reference Counting?

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

    std::thread { reader2, foo }.detach(); // me too! => WHY +1?
    std::thread { writer2, foo }.detach();
}
```



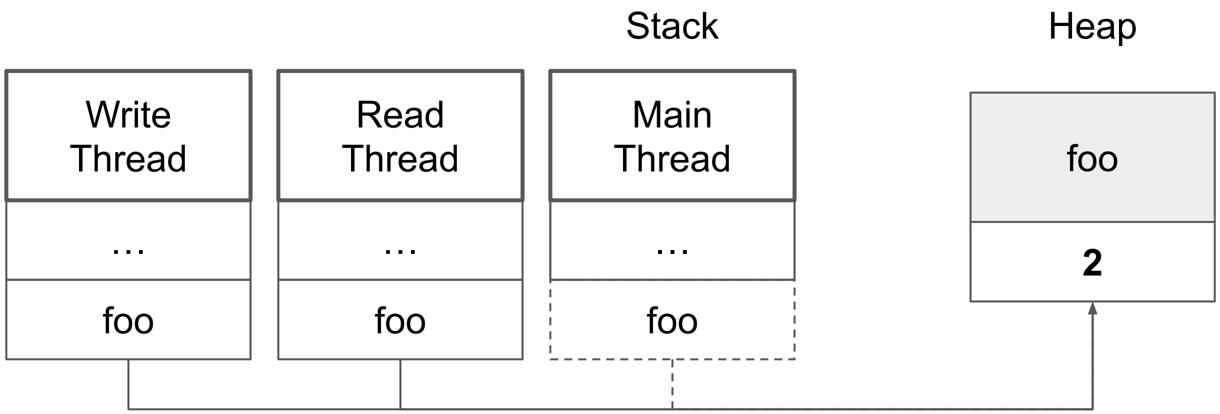
What's Reference Counting?

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

    std::thread { reader2, foo }.detach();
    std::thread { writer2, foo }.detach(); // me 3! => Another WHY +1?
}
```



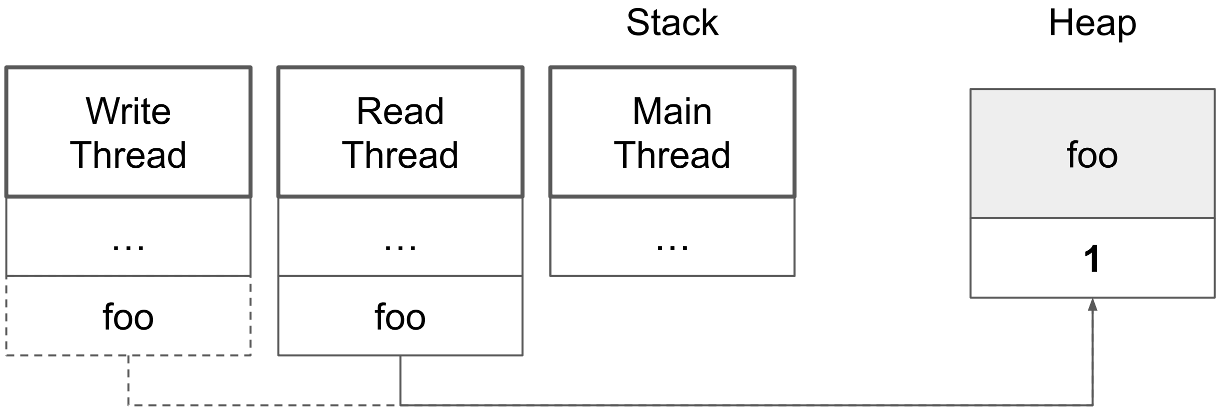
What's Reference Counting?

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

    std::thread { reader2, foo }.detach();
    std::thread { writer2, foo }.detach();
} // I'm done, count me out!
```



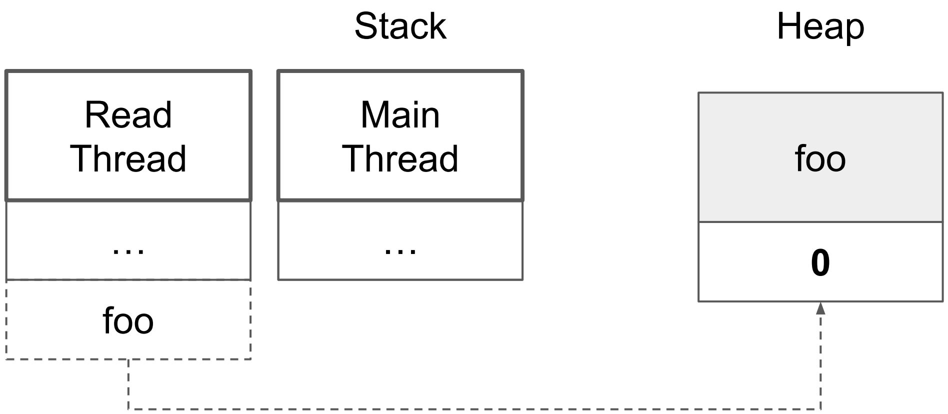
What’s Reference Counting?

```
void writer2(std::shared_ptr<int> foo) {
    foo->fetch_add(1);
} // I’m done, count me out too!
```



What’s Reference Counting?

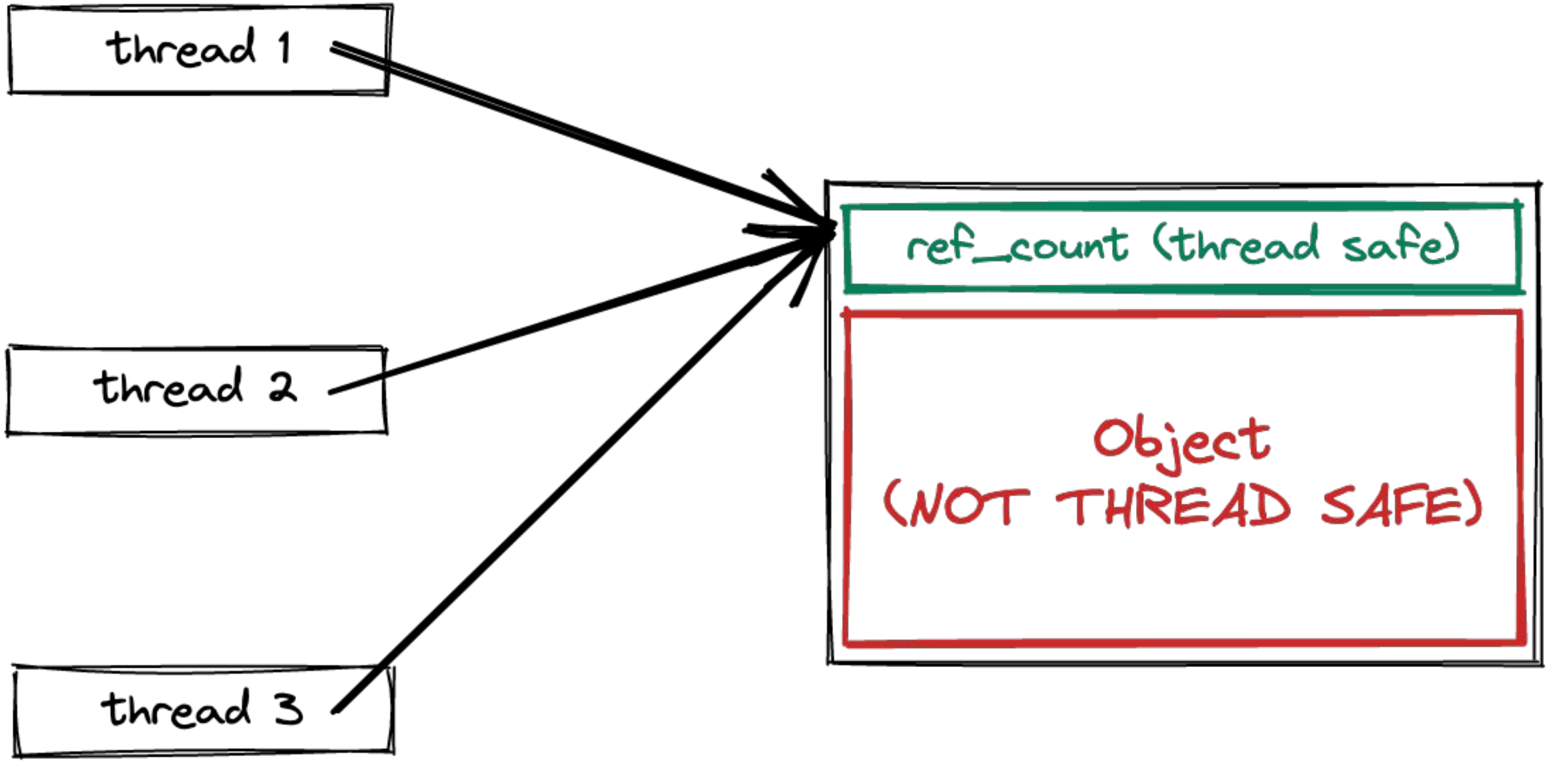
```
void reader2(std::shared_ptr<int> foo) {
    std::cout << foo->load();
} // I’m last! I'll delete foo
```



Is std::shared_ptr thread-safe?

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

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

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

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

Is std::shared_ptr thread-safe?



Is std::shared_ptr thread-safe? Proof!

From: https://en.cppreference.com/w/cpp/memory/shared_ptr

To satisfy **thread safety** requirements, the **reference counters** are typically incremented using an equivalent of [std::atomic::fetch_add](https://en.cppreference.com/w/cpp/atomic/atomic/fetch_add) ...

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

Common Mistakes

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)

```
WARNING: ThreadSanitizer: data race (pid=1651429)
  Write of size 4 at 0x7b0800000030 by thread T2:
   #0 main::$_1::operator()(std::shared_ptr<int>) const
      .../example.cpp:19:12
   ...

  Previous read of size 4 at 0x7b0800000030 by thread T1:
   #0 main::$_0::operator()(std::shared_ptr<int>) const
      .../example.cpp:13:21
   ...
```

Problem 1. Data Race on Value (TSan)

```cpp
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 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)

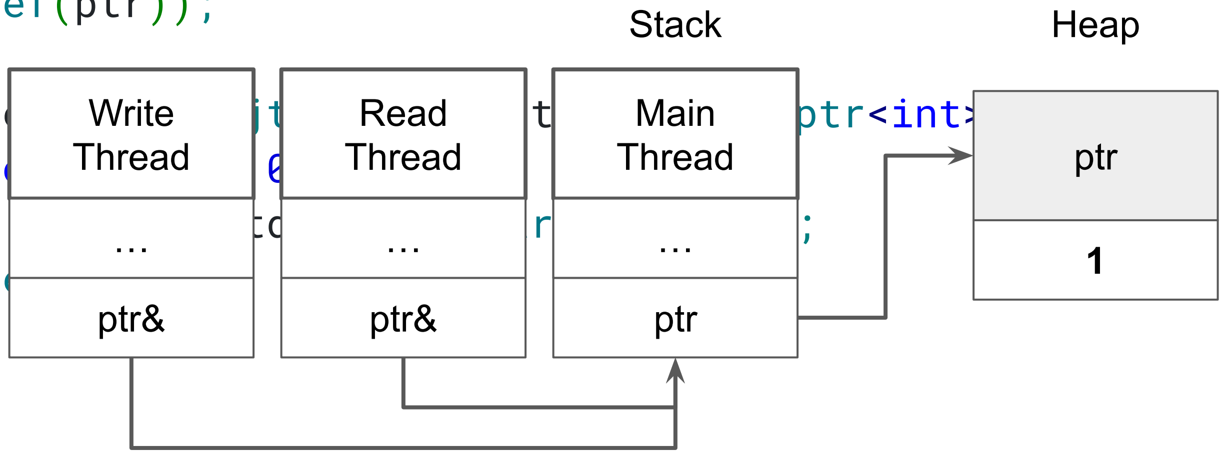
```
WARNING: ThreadSanitizer: data race (pid=1651677)
  Write of size 8 at 0x7ffff63a3b0 by thread T2:
    ...
    #3 std::shared_ptr<int>::operator=(std::shared_ptr<int>&&)
    #4 operator() .../example.cpp:21

  Previous read of size 8 at 0x7ffff63a3b0 by thread T1:
    ...
    #1 bool std::operator==<int>(std::shared_ptr<int> const&,
decltype(nullptr))
    #2 operator() .../example.cpp:13
```

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));
}
```

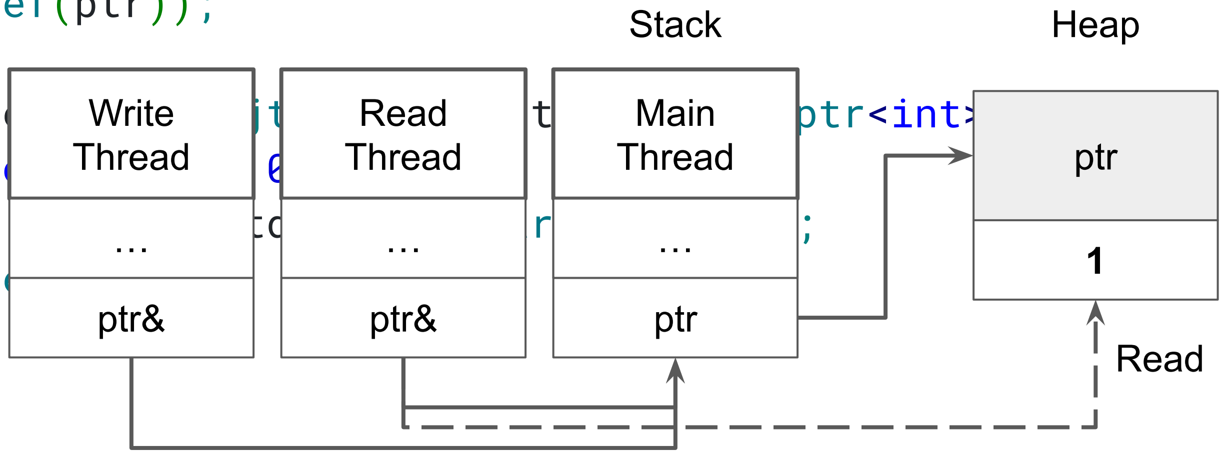
Problem 2. Data Race on Pointer (TSan)

```
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 writ
            f

    }, std::r
}
```



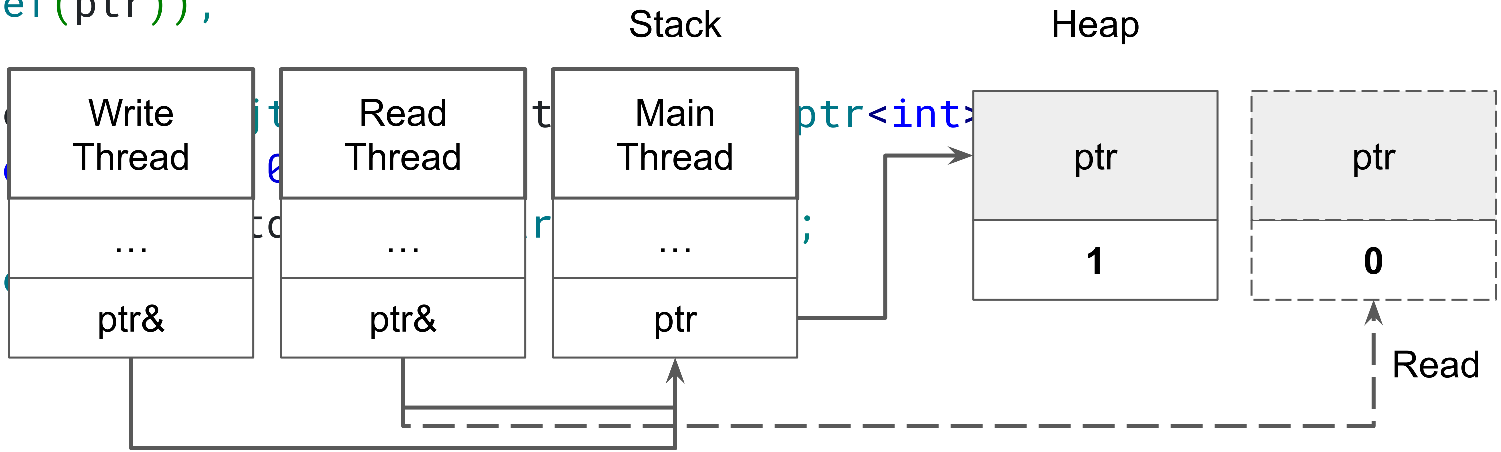
Problem 2. Data Race on Pointer (TSan)

```
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 writ
            f
    }, std::r
}
```



Problem 2. Data Race on Pointer (TSan) - Use After Free

```
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 writ
            f
    }, std::r
}
```



Problem 3. Find the bug

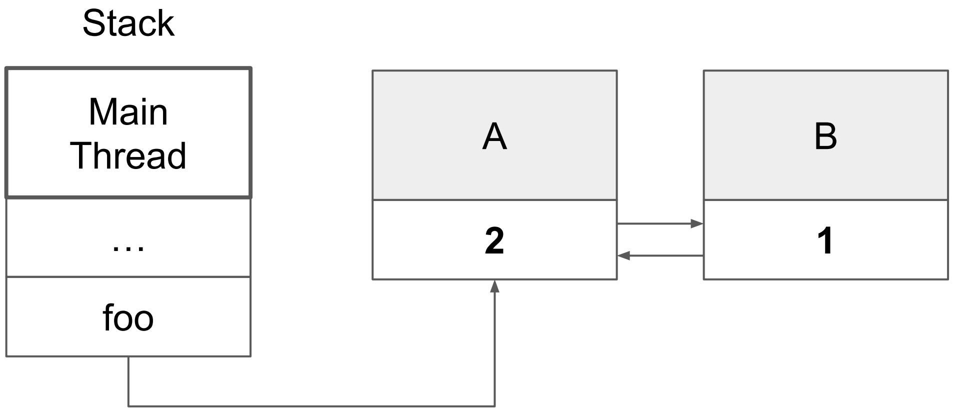
```cpp
// Doubly Linked List
struct DLLNode {
    std::shared_ptr<DLLNode> prev;
    std::shared_ptr<DLLNode> next;
};

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)

```
==1652050==ERROR: LeakSanitizer: detected memory leaks
Indirect leak of 48 byte(s) in 1 object(s) allocated from:
    #8 0x55ea0a56a81b in std::shared_ptr<DLLNode>
std::make_shared<DLLNode>()
    #9 0x55ea0a56a81b in main .../example.cpp:28:11

Indirect leak of 48 byte(s) in 1 object(s) allocated from:
    #8 0x55ea0a56a76e in std::shared_ptr<DLLNode>
std::make_shared<DLLNode>()
    #9 0x55ea0a56a76e in main .../example.cpp:27:11
```

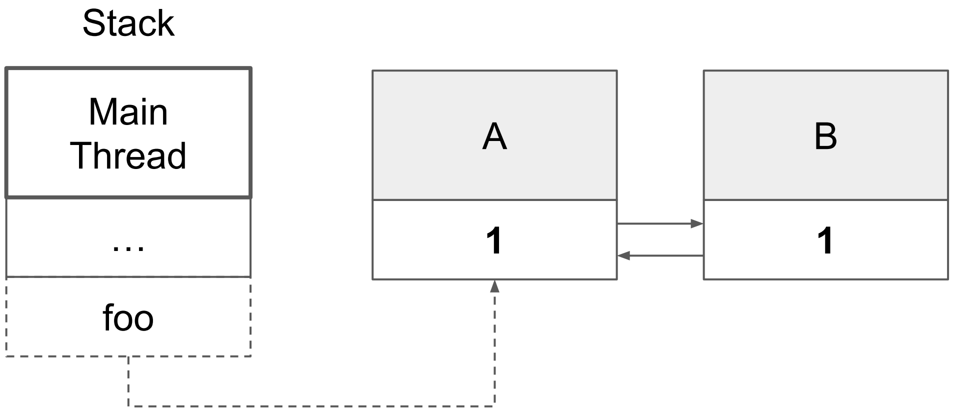
Problem 3. Circular Reference (ASan)

```
// 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();
};
```



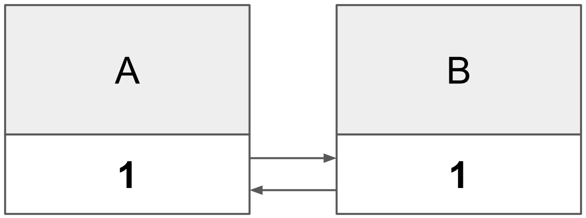
Problem 3. Circular Reference (ASan)

```cpp
// 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();
};
```



std::shared_ptr

Implementation

1. First Attempt - Any problem?

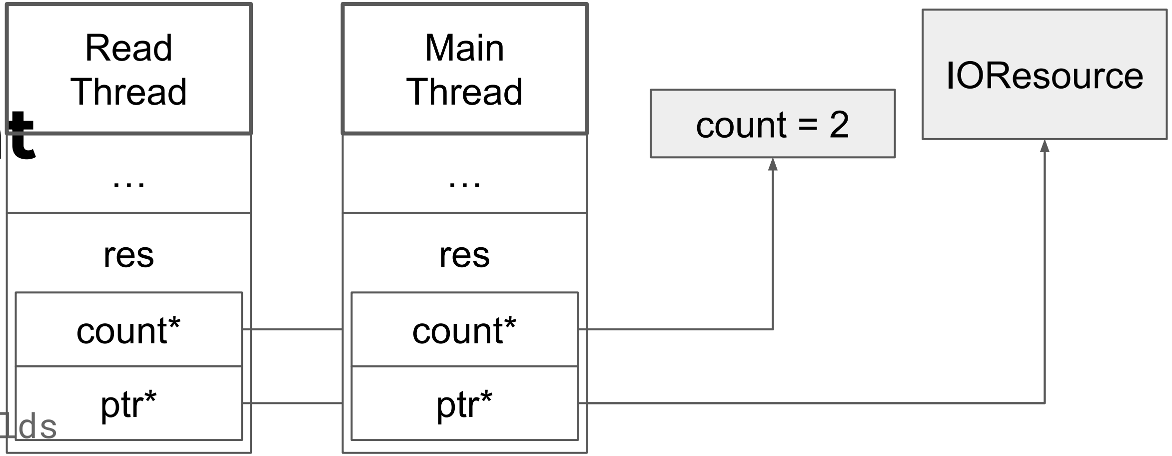
```
template <typename T>
class SharedPtr {
    size_t m_count;
    T* m_ptr;
    // TODO: add additional fields

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;
};
```

1. First Attempt

```
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;
};
```

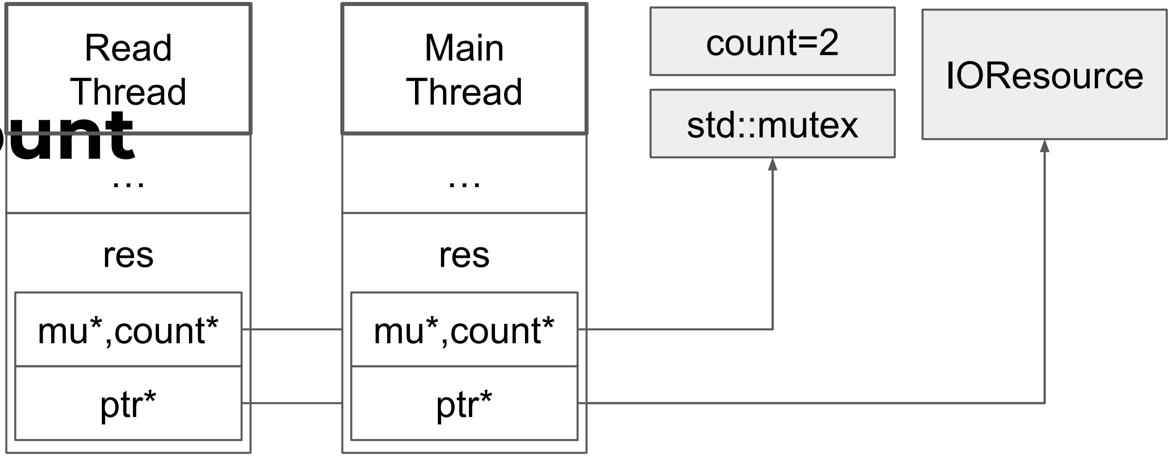
Problem: Not shared

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 size_t(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




```cpp
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. What's wrong here?

```
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. What's wrong here?

```
WARNING: ThreadSanitizer: heap-use-after-free (pid=1652364)
  Atomic read of size 1 at 0x7b0c00000000 by thread T10
(mutexes: write M0):
    #0 pthread_mutex_unlock
    ...
    #5 SharedPtr2<int>::~SharedPtr2() .../example.cpp:35:5


  Previous write of size 8 at 0x7b0c00000000 by thread T10
(mutexes: write M0):
    #0 operator delete(void*)
    #1 SharedPtr2<int>::~SharedPtr2() .../example.cpp:32:7
```

4. What's wrong here?

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

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

    }
};
```

No lock protecting this. Is this safe?

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

    }
};
```

Yes! (As long as you are using shared ptr correctly. i.e. no ref to shared ptr)

Case 1: m_count >= 2, then new_count >= 1, won't delete

Case 2: m_count = 1, no other shared ptr exists and copy constructor can't run in the middle of destructor, safe to delete

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);
    }

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

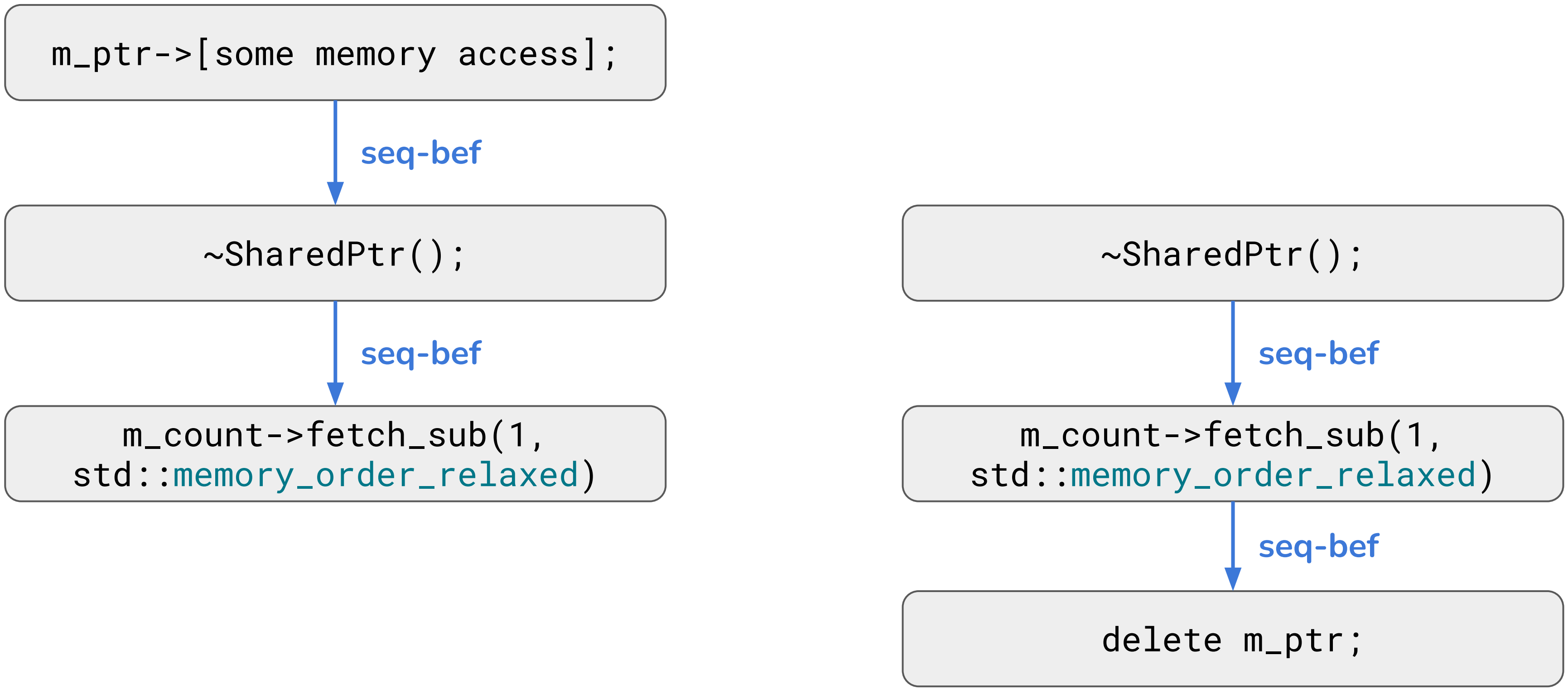
What is the weakest memory order we can use?

How about atomics (how hard can it be) - dtor?

```
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_acq_rel); // why not relaxed? I
will ask you later
    }

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

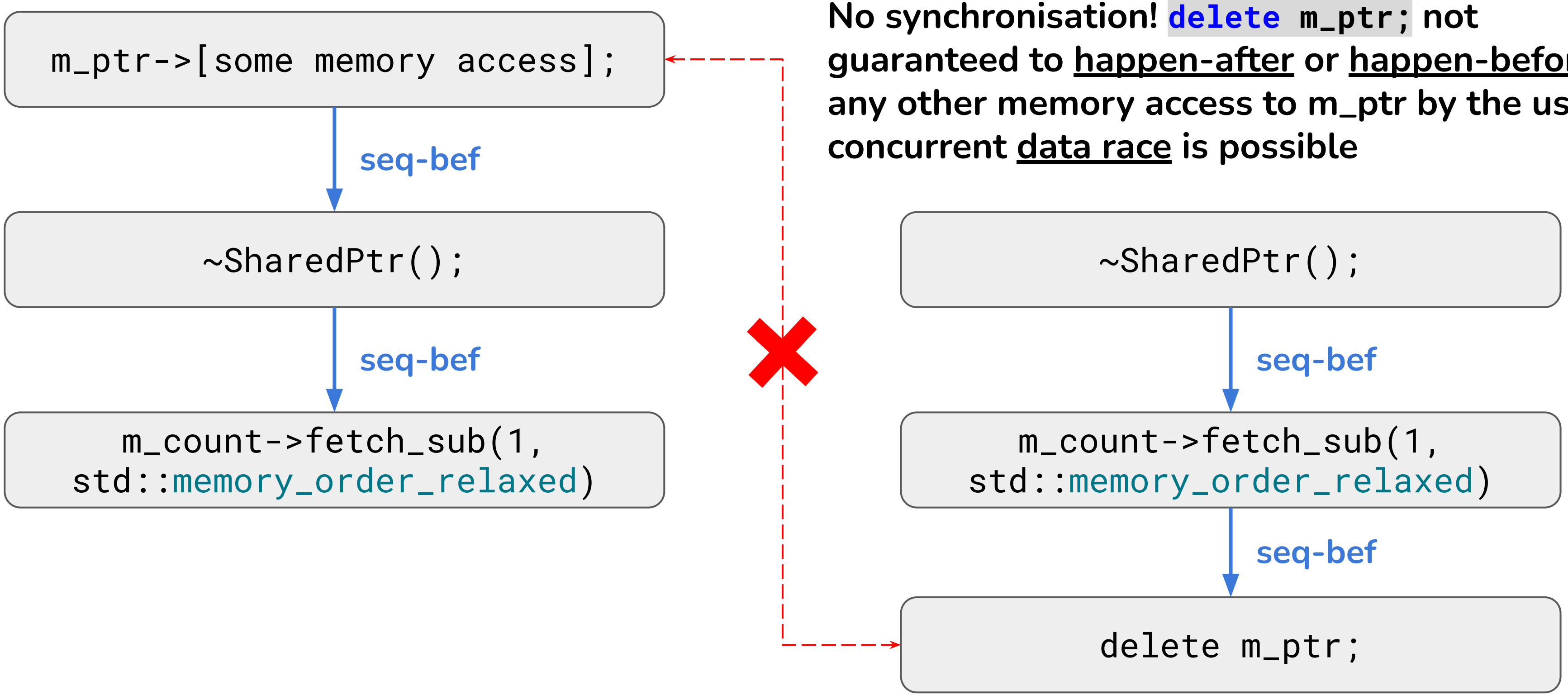
But what about m_count and m_ptr between threads?

How about atomics (how hard can it be)?



How about atomics (how hard can it be)?

No synchronisation! `delete m_ptr;` not guaranteed to happen-after or happen-before any other memory access to m_ptr by the user, concurrent data race is possible



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_acq_rel);
    }

    ~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

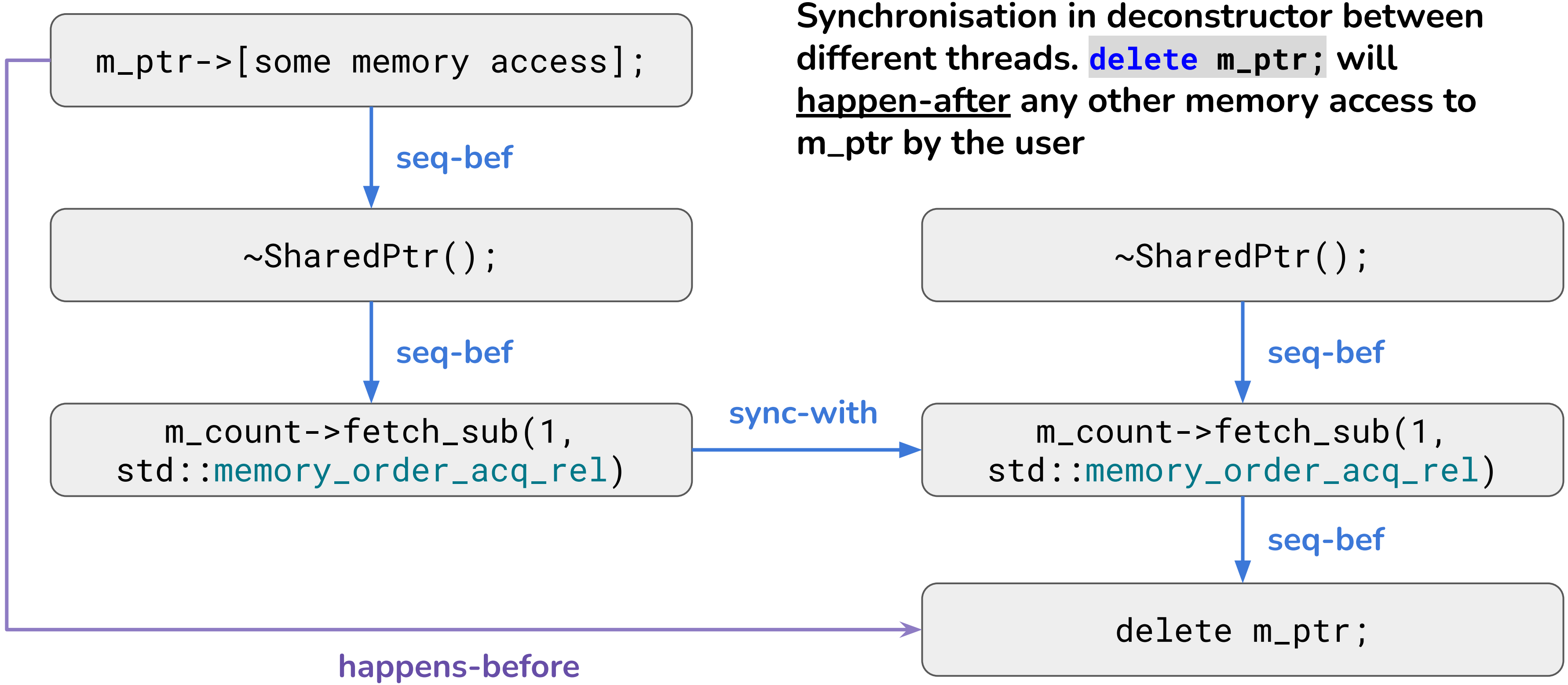
memory_order_acq_rel

!! New Stuff !!

All writes in other threads that release the same atomic variable are visible before the modification and **the modification is visible in other threads that acquire the same atomic variable. - CPP Reference**

How about atomics (how hard can it be)?

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



How about atomics the ctor?

```
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::relaxed); // or acq_rel?
    }

    ~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

Proof from CPP Reference

To satisfy thread safety requirements, the reference counters are typically incremented using an equivalent of [std::atomic::fetch_add] with [std::memory_order_relaxed] (decrementing requires stronger ordering to safely destroy the control block). - CPP Reference

To understand why memory_order_relaxed in constructor is acceptable in cpp reference, we need to understand another concept called release sequence. This is not taught in CS3211 and won't be tested.

But of course, you need to understand why **acq_release** is correct