CS3211 Tutorial 3

Smart Pointers

(AY 25/26 Semester 2)

February 12, 2026

(Compiled by Benson, thanks to all past and current TAs!)

Join our telegram group!
https://t.me/+MqCIT5ObVgxiMzRl

Admin Info

- No tutorials next week! Please don't show up – I won't either.
- You may check your attendance records on my website.

Contents

Why Smart Pointers?
- `std::unique_ptr`
- `std::shared_ptr`
- Common Pitfalls when using `std::shared_ptr`

Implementing Shared Pointer

Contents

Why Smart Pointers?
- `std::unique_ptr`
- `std::shared_ptr`
- Common Pitfalls when using `std::shared_ptr`

Implementing Shared Pointer

Smart Pointers – `std::unique_ptr`

```cpp
int main() {
  std::unique_ptr<int> foo =
      std::make_unique<int>(0);
  use_foo(foo.get()); // foo*
}
```

```cpp
int main() {
  auto deleter = [](FILE* f) {
    fclose(f);
  }
  auto bar = std::unique_ptr<FILE, decltype(deleter)> {
    fopen("file.txt", "w"), deleter
  };
  use_bar(bar.get());
}
```

Link to godbolt

Turns non-RAII types into RAII types (`delete` the resource when the pointer goes out of scope).

Unique Ownership

Smart Pointers – `std::shared_ptr`

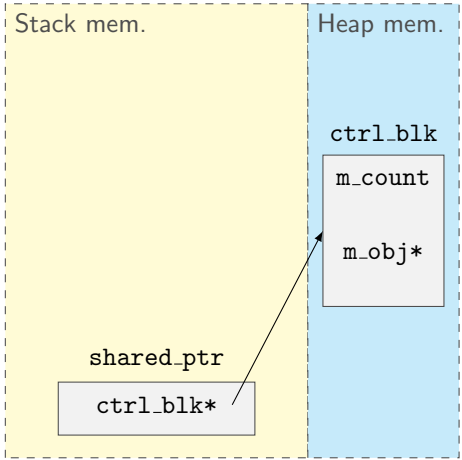
```
1  auto foo = std::make_shared<int>(0);
2  auto trader_fn = [](std::shared_ptr<int> foo) {
3      ...
4  };
5  std::thread { trader_fn, foo }.detatch();
6  std::thread { trader_fn, foo }.detatch();
```

Link to godbolt

Protects the shared resource!
`delete` only *after* the end of **all** shared pointer's lifetime.

Smart Pointers – std::shared_ptr

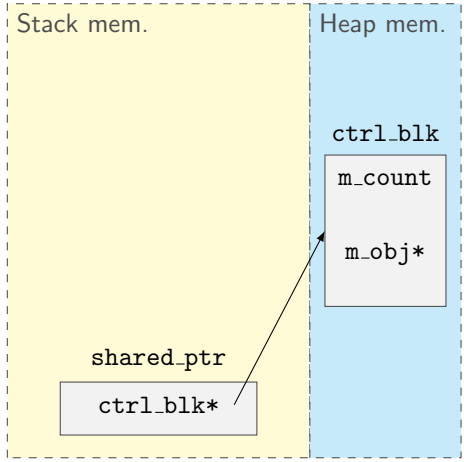
```cpp
void foo(std::shared_ptr<int> arg) {
  std::cout << *x;
}

int main() {
  auto x = std::make_shared<int>(100);
  foo(x);
}
```



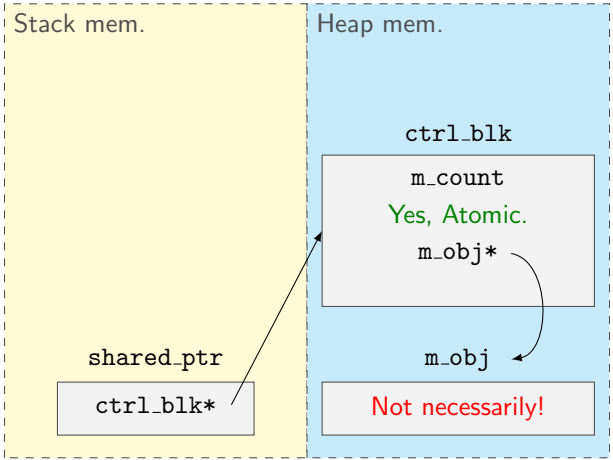
Smart Pointers – std::shared_ptr

```
1  void rc(const std::shared_ptr<int> &x) {
2    std::cout << x.use_count();
3  }
4
5  void foo(std::shared_ptr<int> arg) {
6    rc();
7    auto z = arg;
8    rc();
9  }
10
11 int main() {
12   auto x = std::make_shared<int>(100);
13   rc(x);
14   {
15     auto a = x; rc(x);
16     auto b = x; rc(x);
17     auto c = x; rc(x);
18     foo(x);     rc(x);
19   }
20   rc(x);
21 }
```

Link to godbolt



Thread Safety



Problem 1. Data Race on Object

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

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

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

reader.join();
writer.join();
```

Link to godbolt

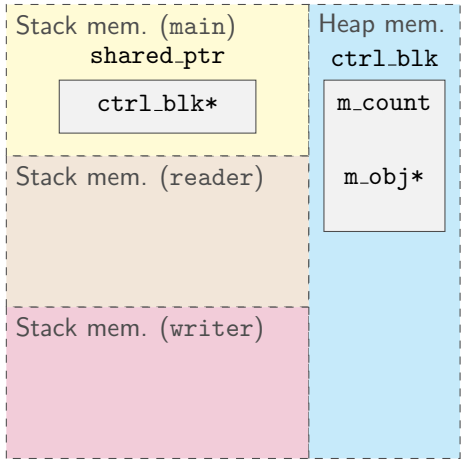
The underlying object is not thread safe!

Extra: How to fix?

Problem 1: Data Race on Object

```
1  std::shared_ptr<int> ptr =
2      std::make_shared<int>(0);
3
4  auto reader = std::thread(
5  [](std::shared_ptr<int> ptr) {
6    for(int i = 0; i < 100; i++)
7      printf("%d ", *ptr);
8    printf("\n");
9  }, ptr);
10
11 auto writer = std::thread(
12 [](std::shared_ptr<int> ptr) {
13   for(int i = 0; i < 100; i++)
14     *ptr = i;
15 }, ptr);
16
17 reader.join();
18 writer.join();
```

Link to godbolt



Problem 2. Data Race on Pointer

```
1  std::shared_ptr<int> ptr;
2
3  auto reader = std::thread(
4  [](std::shared_ptr<int>& ptr) {
5    while(ptr == nullptr);
6    printf("%d\n", *ptr);
7  }, std::ref(ptr));
8
9  auto writer = std::thread(
10 [](std::shared_ptr<int>& ptr) {
11   for(int i = 0; i < 100; i++)
12     ptr = std::make_shared<int>(i);
13 }, std::ref(ptr));
14
15 reader.join();
16 writer.join();
```

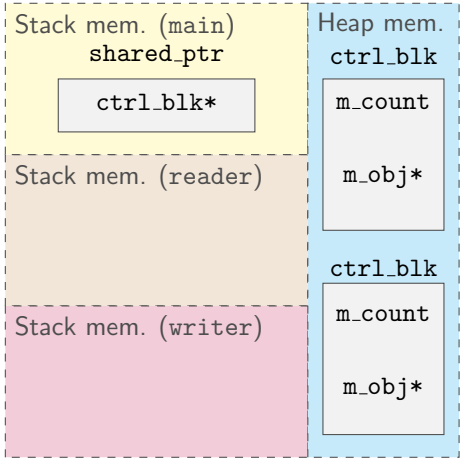
Link to godbolt

Data race on `ptr` itself!

Problem 2: Data Race on Pointer

```
1  std::shared_ptr<int> ptr;
2
3  auto reader = std::thread(
4  [](std::shared_ptr<int>& ptr) {
5    while(ptr == nullptr);
6    printf("%d\n", *ptr);
7  }, std::ref(ptr));
8
9  auto writer = std::thread(
10 [](std::shared_ptr<int>& ptr) {
11   for(int i = 0; i < 100; i++)
12     ptr = std::make_shared<int>(i);
13 }, std::ref(ptr));
14
15 reader.join();
16 writer.join();
```

Link to godbolt



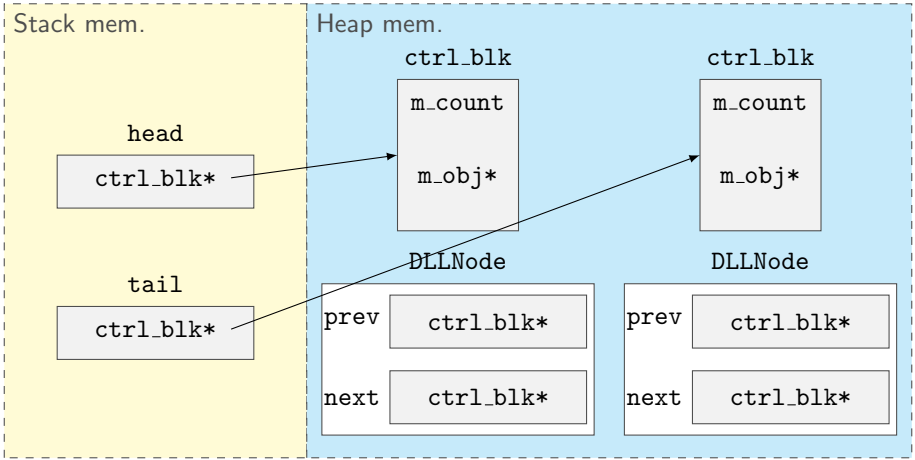
Problem 3. Circular Reference

```
1  struct DLLNode
2  {
3    std::shared_ptr<DLLNode> prev;
4    std::shared_ptr<DLLNode> next;
5  };
6
7  struct DLL
8  {
9    std::shared_ptr<DLLNode> head {};
10   std::shared_ptr<DLLNode> tail {};
11
12   void push_front(std::shared_ptr<DLLNode>);
13   void push_back(std::shared_ptr<DLLNode>);
14
15   std::shared_ptr<DLLNode> front();
16   std::shared_ptr<DLLNode> back();
17 };
18
```

Link to godbolt

Run `a->next = b` and `b->prev = a`.
Extra: How to fix?

Problem 3. Circular Reference



Contents

Why Smart Pointers?
- `std::unique_ptr`
- `std::shared_ptr`
- Common Pitfalls when using `std::shared_ptr`

Implementing Shared Pointer

Attempt 1: Using a Mutex

```
1  template <typename T>
2  struct SharedPtr {
3  private:
4    size_t m_count; T* m_ptr;
5
6  public:
7    SharedPtr(T* ptr) : m_count(1), m_ptr(ptr) {}
8
9    SharedPtr(const SharedPtr& other)
10       : m_count(other.m_count) , m_ptr(other.m_ptr) {
11     ++m_count;
12   }
13
14   ~SharedPtr() {
15     if((--m_count) == 0)
16       delete m_ptr;
17   }
18
19   T* get() { return m_ptr; }
20   const T* get() const { return m_ptr; }
21 };
```

Link to godbolt

Use-after-free Situation!
We must not `unlock` the mutex after we free it.

Attempt 2: Atomic Reference Counter

```
1  template <typename T>
2  struct SharedPtr {
3  private:
4    size_t m_count; T* m_ptr;
5
6  public:
7    SharedPtr(T* ptr) : m_count(1), m_ptr(ptr) {}
8
9    SharedPtr(const SharedPtr& other)
10       : m_count(other.m_count) , m_ptr(other.m_ptr) {
11     ++m_count;
12   }
13
14   ~SharedPtr() {
15     if((--m_count) == 0)
16       delete m_ptr;
17   }
18
19   T* get() { return m_ptr; }
20   const T* get() const { return m_ptr; }
21 };
```

Link to godbolt

Can we weaken the memory order?

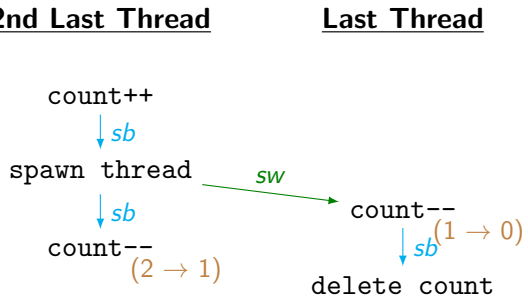
Attempt 3: Relaxed Reference Counting

Is there a data race?

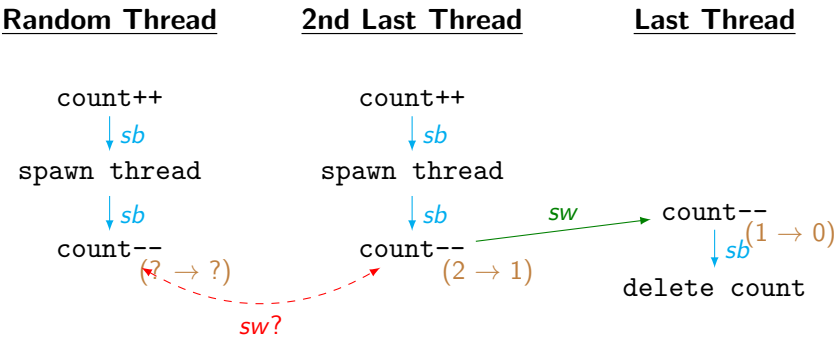
```
SharedPtr(const SharedPtr& other)
    : m_count(other.m_count)
    , m_ptr(other.m_ptr) {
  m_count->fetch_add(1,
      std::memory_order::relaxed);
}
```

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

Attempt 3: Relaxed Refernece Counting



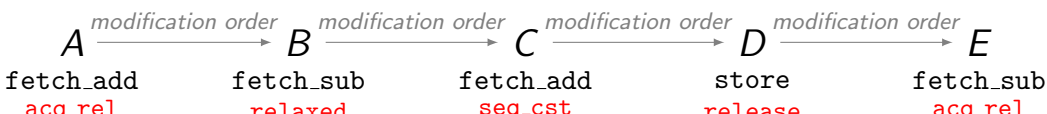
What happens if we have 3 threads?



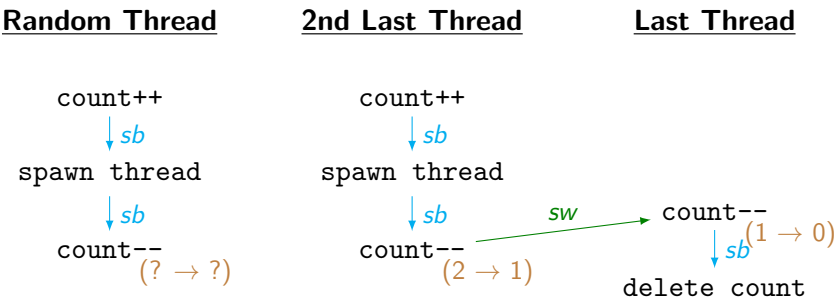
Release Sequences

Release Sequence: Maximal contiguous subsequence of atomic read-modify-write (RMW) operations that follow in the modification order of the memory location.

Guarantee: An atomic release *A synchronizes with* an atomic acquire *B* that reads from the release sequence headed by *A*.



Release Sequences in Action



Summary

- `std::unique_ptr`: Turns non-RAII types into RAII types. Enforces unique ownership.
- `std::shared_ptr`: Performs reference counting to support shared ownership.
- We implemented `std::shared_ptr` using mutex and atomics!
  - Mutex: Avoid **use-after-free** situation.
  - Atomics: **Release sequences**.

Attendance

Session ID: 1939881
CS3211 Tutorial 3
12 February 2026 14:00-16:00
BIZ2-02-02 - SEMINAR ROOM 2-2

That's it!

Anonymous Feedback (throughout the semester):

https://forms.gle/6a9T4t88wNwYxG4G8

The link to the tutorial slides will be posted in telegram.