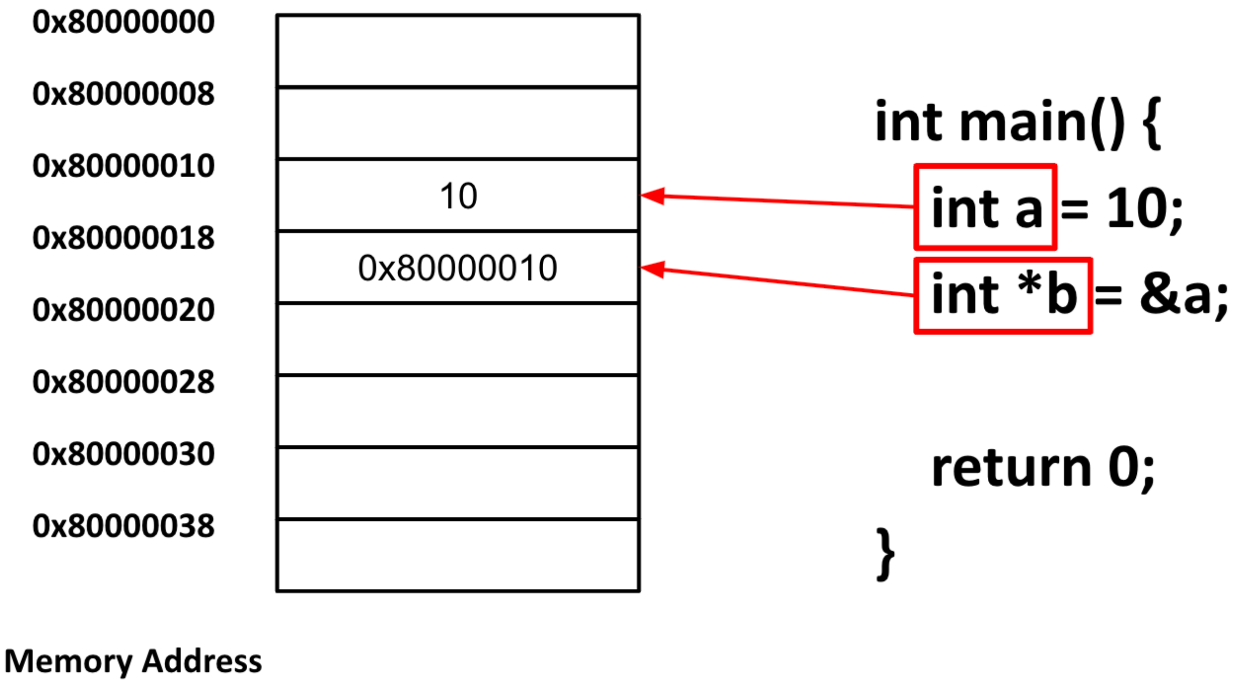
Smart Pointers

Based on "Using C++11's Smart Pointers": http://umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf
Based on https://devblogs.microsoft.com/cppblog/stdoptional-how-when-and-why

Recap: Raw Pointer

- Pointer itself is a **variable**
  - But, have a specific purpose - stores the memory address



Recap: Variable Lifetime

- From cppreference.com
  - For any objects or reference, there is a point of execution a program when its lifetime begins and there is a moment when it ends

- Example
  - Local variable: bracket scope (inside “{” and “}”)

Recap: Memory Allocation

- Stack Memory Allocation

- Heap Memory Allocation (often called Dynamic Memory Allocation)

Problem: Memory Deallocation

- For a stack memory, usually we do not need to care about the memory deallocation
  - Memory is deallocated when the code goes out of the variable scope

- However, for a heap memory, we especially need to care about the memory deallocation
  - If we do not deallocate the heap memory properly, it critically causes the program performance, security, etc.

Solution via Design Pattern: RAII

- Resource Acquisition is Initialization

- 자원 관리를 스택에 할당한 “객체”를 통해서 진행
  - 스택에 할당한 객체이므로, 반드시 자원이 회수된다.

Combining RAII into the pointers

- We want to ensure that when we use the pointer,
  - Memory deallocates when it’s lifecycle is ended
  - If multiple pointer points the same object, determine which pointer deallocates the object

- C++ introduces the smart pointers (std::unique_ptr, std::shared_ptr)
  - Pointer wrapper **class** following the RAII design pattern
  - Introduces the ownership concept

Object Ownership

- Object Ownership
  - If you have multiple code using a certain object, which one should be freeing the object?
    - Typical issues: memory leak, dangling pointers, use-after-free, double free
  - Object ownership: “which code” is responsible for deleting the object

- Smart pointers make this ownership implementation easy

Smart Pointers

- **A smart pointer is a class object**
  - Behaves like built-in pointers
  - Manage objects that you created with “new”
    - So you don’t need to explicitly “delete” those

- **unique_ptr**
  - Implements unique ownership: only one smart pointer owns the object at a time
- **shared_ptr**
  - Implements shared ownership: any number of smart pointers can jointly own the object
  - weak_ptr is used together with shared_ptr

Unique Ownership

```
unique_ptr<Thing> p1 (new Thing); // p1 owns the Thing
unique_ptr<Thing> p2(p1);  // error - copy construction is not allowed.
unique_ptr<Thing> p3;      // an empty unique_ptr;
p3 = p1;                   // error, copy assignment is not allowed.
```

Transferring Ownership

```
//create a Thing and return a unique_ptr to it:
unique_ptr<Thing> create_Thing()
{
    unique_ptr<Thing> local_ptr(new Thing);
    return local_ptr; // local_ptr will surrender ownership
}
void foo()
{
    unique_ptr<Thing> p1(create_Thing()); // move ctor from returned rvalue
    // p1 now owns the Thing

    unique_ptr<Thing> p2; // default ctor'd; owns nothing
    p2 = create_Thing(); // move assignment from returned rvalue
    // p2 now owns the second Thing
}
```

```
unique_ptr<Thing> p1(new Thing); // p1 owns the Thing
unique_ptr<Thing> p2; // p2 owns nothing
// invoke move assignment explicitly
p2 = std::move(p1); // now p2 owns it, p1 owns nothing
// invoke move construction explicitly
unique_ptr<Thing> p3(std::move(p2)); // now p3 owns it, p2 and p1 own nothing
```

unique_ptr

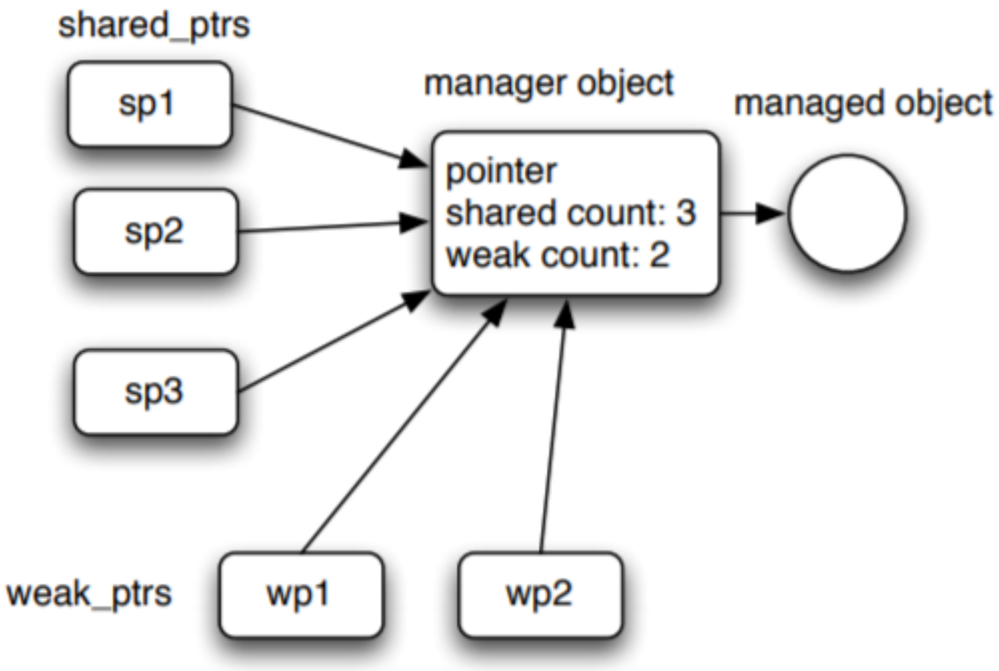
- **unique_ptr can be thought as a special case of shared_ptr**
  - The shared count is always <= 1

- **Difference b/w unique_ptr and shared_ptr**
  - unique_ptr has no runtime costs
    - unique_ptr does not need manager object
  - unique_ptr implements a unique ownership concept

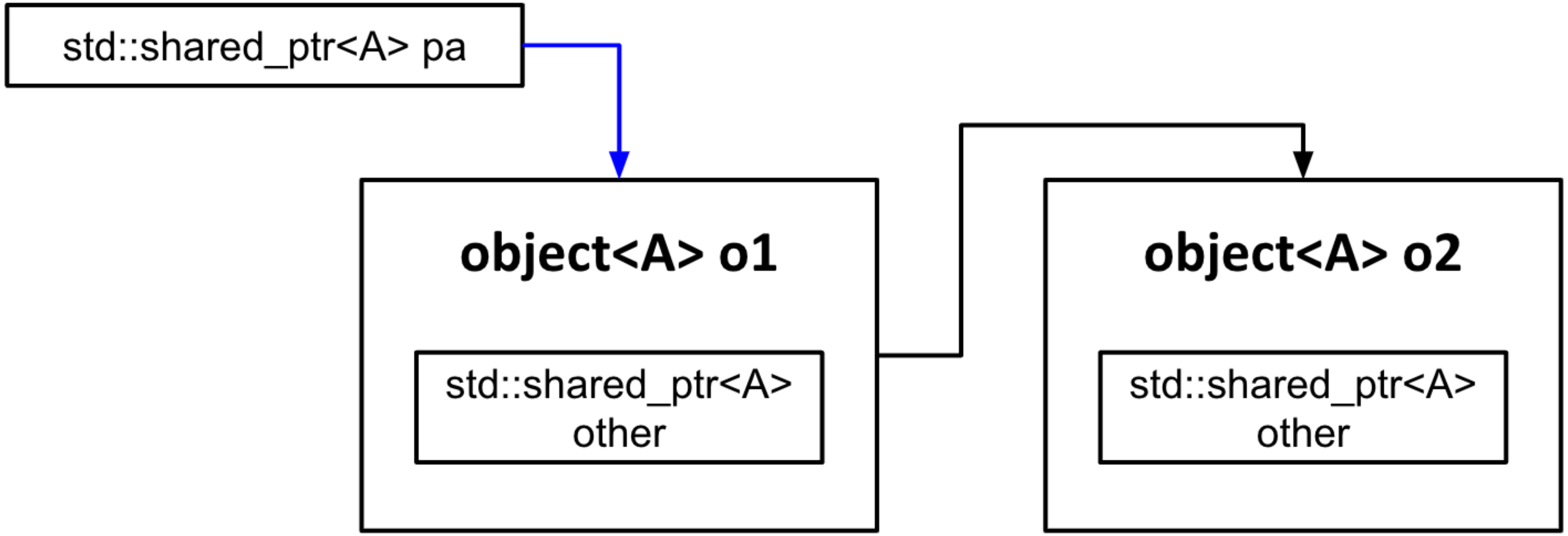
shared_ptr

- If shared count becomes zero, the object is deleted

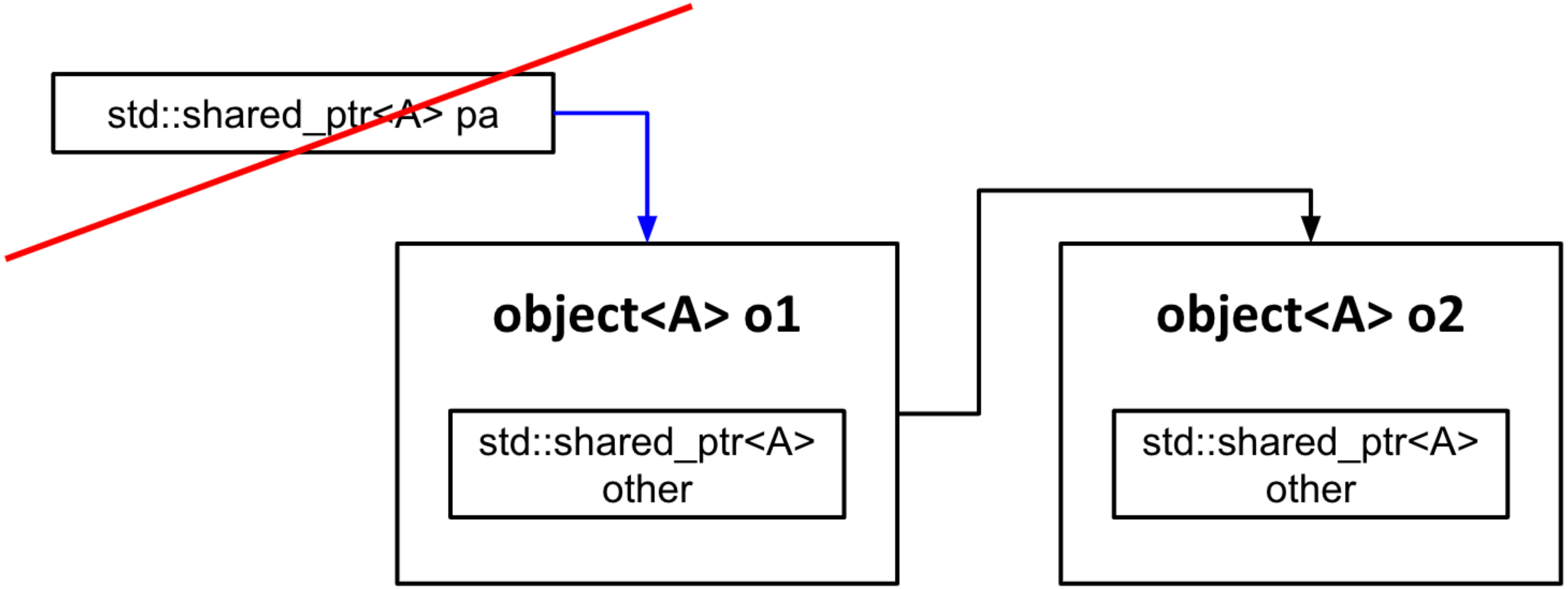
- If weak count becomes zero, the manager object is deleted



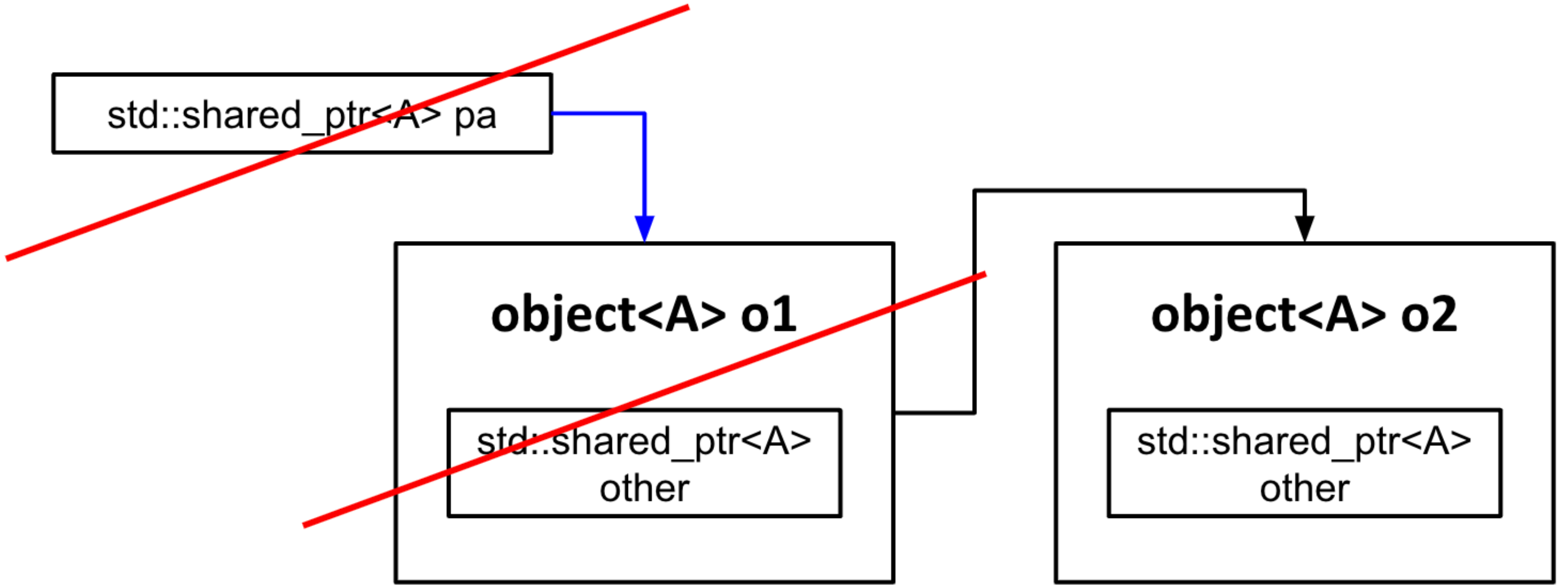
shared_ptr: example



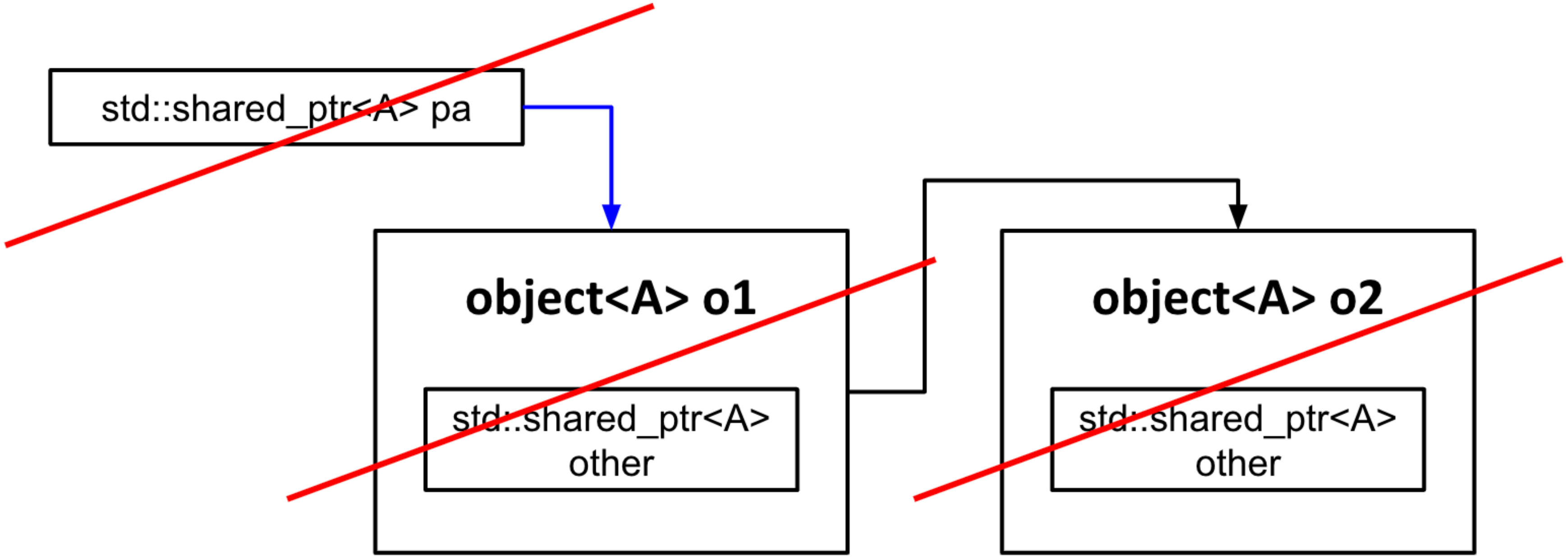
shared_ptr: example



shared_ptr: example



shared_ptr: example



How to use shared_ptr

```
class Thing {
 public:
    void defrangulate();
};
ostream& operator<< (ostream&, const Thing&);

// a function can return a shared_ptr
shared_ptr<Thing> find_some_thing();
// a function can take a shared_ptr parameter by value;
shared_ptr<Thing> do_something_with(shared_ptr<Thing> p);

void foo()
{
    // the new is in the shared_ptr constructor expression:
    shared_ptr<Thing> p1(new Thing);
    // ...
    shared_ptr<Thing> p2 = p1; // p1 and p2 now share ownership of the Thing
    // ...
    shared_ptr<Thing> p3(new Thing); // another Thing

    p1 = find_some_thing(); // p1 may no longer point to first Thing
    do_something_with(p2);
    p3->defrangulate(); // call a member function like built-in pointer
    cout << *p2 << endl; // dereference like built-in pointer

    // reset with a member function or assignment to nullptr:
    p1.reset(); // decrement count, delete if last
    p2 = nullptr; // convert nullptr to an empty shared_ptr, and decrement count;
}
// p1, p2, p3 go out of scope, decrementing count, delete the Things if last
```

make_shared

- Using shared_ptr requires two allocations when creating an object
  - a manager object and a managed object

```
shared_ptr<Thing> p(new Thing); // ouch - two allocations
```

- make_shared allows you to only allocate once

```
shared_ptr<Thing> p(make_shared<Thing>()); // only one allocation!
```

  - Magic behind: “a manger object” and “a managed object” are merged into a single object if using “make_shared<>”

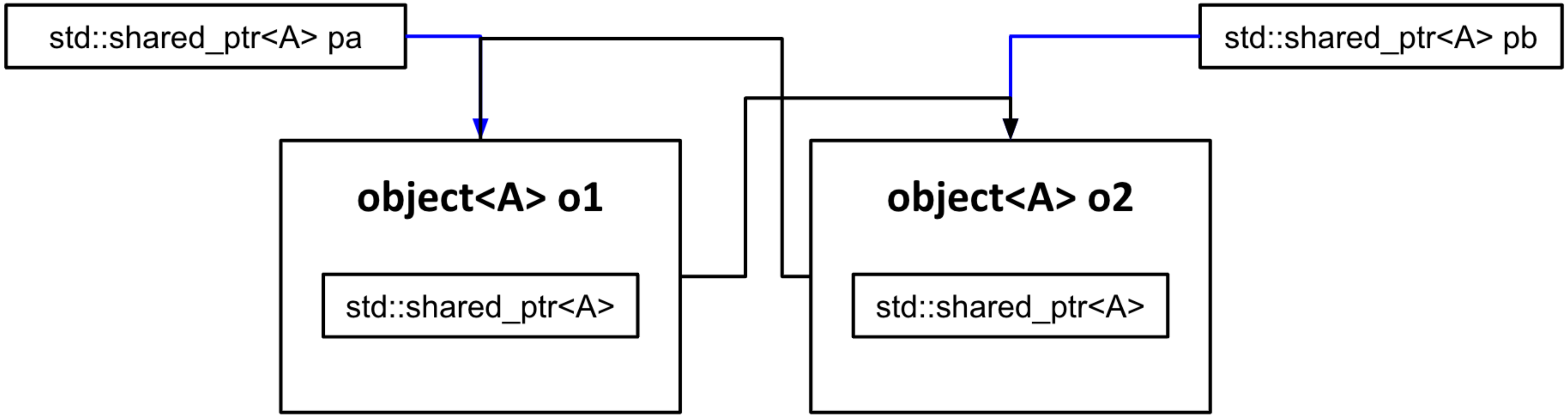
Circular Dependency

- What happens if two shared_ptr points to each other's objects?
  - No one can be destructed

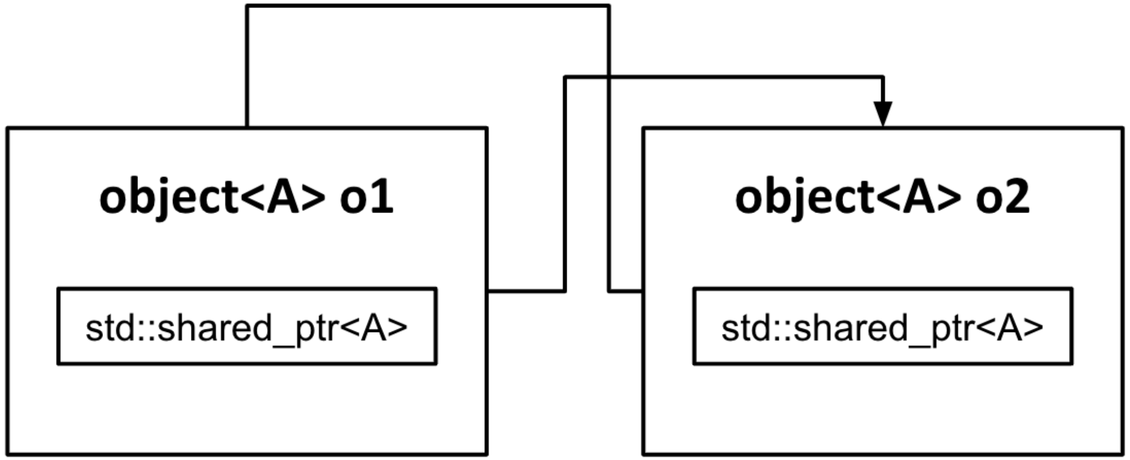
```cpp
#include <iostream>
#include <memory>

class A {
  int *data;
  std::shared_ptr<A> other;

 public:
  A() {
    data = new int[100];
  }

  ~A() {
    delete[] data;
  }

  void set_other(std::shared_ptr<A> o) { other = o; }
};

int main() {
  std::shared_ptr<A> pa = std::make_shared<A>();
  std::shared_ptr<A> pb = std::make_shared<A>();

  pa->set_other(pb);
  pb->set_other(pa);
}
```

Code snippet from https://modoocode.com/252

Circular Dependency



Circular Dependency



Solution: weak_ptr

- weak_ptr does not affect the reference count
- If you want to access the object using the weak_ptr, you must convert weak_ptr to shared_ptr

```
std::shared_ptr<FibonacciNode<T>> right;
// NOTE: If you set left pointer to share
// So, left pointer should be set to weak
std::shared_ptr<FibonacciNode<T>> child;
std::weak_ptr<FibonacciNode<T>> left;
std::weak_ptr<FibonacciNode<T>> parent;
```

Motivation: Why std::optional<>

- **How to optionally accept or return an object?**

```
void maybe_take_an_int(int value = -1); // an argument of -1 means "no value"
int maybe_return_an_int(); // a return value of -1 means "no value"
```

  - Error-prone: you will need a pre-defined constant value, or you may not even be able to pre-define such a constant value

- **Possible solution?**

```
void maybe_take_an_int(int value = -1, bool is_valid = false);
void or_even_better(pair<int,bool> param = std::make_pair(-1, false));
pair<int, bool> maybe_return_an_int();
```

  - Awkward, hard to use

std::optional<> comes to the rescue

```
optional<int> o = maybe_return_an_int();
if (o.has_value()) { /* ... */ }
if (o) { /* ... */ } // "if" converts its condition to bool
```

```
if (o) { cout << "The value is: " << *o << '\n'; }
```

```
cout << "The value is: " << o.value() << '\n';
```

```
cout << "The value might be: " << o.value_or(42) << '\n';
```