

CS 105C: Lecture 9

Last Time...

Copying is expensive, stealing is cheap!*

```
1 int main(){
2     X x, x2;
3     ...
4     x2 = create_an_x();
5     x = x2;
6 }
```

Wherever possible, we'd like to move data around instead of making copies of it.

But we can't always steal: stealing the result of `create_an_x()` is okay, while stealing `x2` is not okay.

*Please do not attempt to use this as a defense in court.

Rvalues are about to be lost anyways



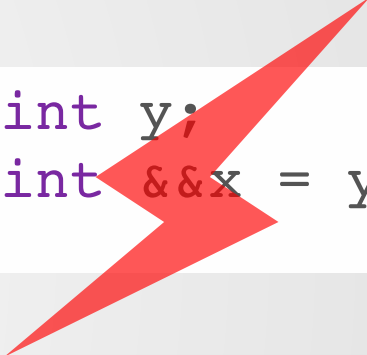
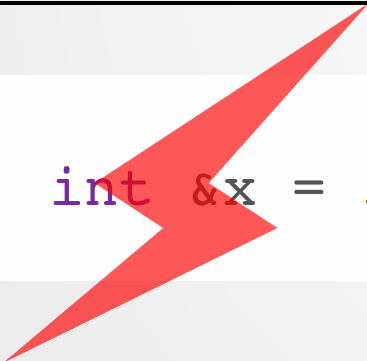


Anything which must go on the right hand side of an assignment operator cannot be reused unless we bind its name.

```
1 string s1, s2;  
2 ...  
3 interleave(s1, s2);
```



Unless we assign this to something, the return value from `interleave()` will expire after this line is executed.

Use rvalue references, which can only bind to rvalues

Can I bind a... / ...to a	Lvalue Reference	const Lvalue Reference	Rvalue Reference
Lvalue	<pre>1 int y; 2 int &x = y;</pre> 	<pre>1 int y; 2 const int &x = y;</pre> 	<pre>1 int y; 2 int &&x = y;</pre> 
Rvalue	<pre>1 2 int &x = 5;</pre> 	<pre>1 2 const int &x = 5;</pre> 	<pre>1 2 int &&x = 5;</pre> 

Now, instead of making copies all the time, we can steal data when we know the other object is about to expire!

```
1 // Expensive to Copy!
2 struct ETC{
3     int* data;
4     int size;
5     ETC() = something;
6     ETC(const ETC& other) = something;
7     ETC& operator=(const ETC& other) = something;
8     ETC(ETC&& other) noexcept : data{nullptr}, size{0} {
9         std::swap(data, other.data);
10        std::swap(size, other.size);
11    }
12    ETC& operator=(ETC&& other) noexcept {
13        std::swap(data, other.data);
14        std::swap(size, other.size);
15    }
16 };
```

Use looks exactly like before, but with +efficiency

```
1 ETC gen_etc();  
2  
3 int main(){  
4     ETC e1 = gen_etc(); // Constructed by move  
5     ETC e2 = e1;       // Constructed by copy  
6 }
```

Points of Confusion

Use `std::move` to force move construction.

```
1 ETC a1;  
2 ...  
3 ETC a2 = a1; // Copied  
4 ETC a3 = std::move(a1); // Moved
```

But **`std::move` doesn't actually move anything!** It just converts its operand into an rvalue reference

After move, programmer may not assume anything about the state of `a1`.

A named rvalue reference is an lvalue!

```
1 ETC&& a = gen_etc();  
2 ETC b = a; // Constructed by copy!
```

Rvalues are about to be lost, so we can steal all their stuff--once you bind to an rvalue reference, the rvalue's lifetime is extended, so this is no longer the case.

CS 105C: Lecture 9

Smart Pointers and Perfect Forwarding

But first, a redux!

RValues and LValues

The Ownership Problem

Episode 5: RAII Strikes Back!

Let's write an RAII wrapper class!

Last time we discussed RAII, I showed you classes that managed an object's lifetime *and* did something else (write to a file, allow memory access, control a lock).

Now that we have template tools available to us, let's try to write a templated RAII class that *only manages lifetimes* (we'll leave the functionality to the managed class)

Let's write an RAII wrapper class!

```
1  template <typename T>
2  class Manager {
3      T* managed;
4
5      Manager()      : managed(nullptr) { };
6      Manager(T* t) : managed(t)      { }
7      ~Manager()    { delete managed; }
8  };
```

```
1  int main(){
2      Dog* d = new Dog();
3      Manager managed_dog(d);
4
5      /* Really complicated logic */
6
7      ...
8      // Now we can't forget to free the memory
9      // because it'll automatically be deleted when
10     // managed_dog goes out of scope!
11 }
```

```
1  template <typename T>
2  class Manager {
3      T* managed;
4
5      Manager() = delete;
6      Manager(T* t) : managed(t) { }
7      ~Manager(){ delete managed; }
8  };
```

```
1  template <typename T>
2  void compute_manager(Manager<T> m) {
3      // Do something
4  }
5
6  int main(){
7      Dog* d = new Dog();
8      Manager managed_dog(d);
9      compute_manager(m);
10
11     //...oh dangnabbit
12 }
```

```

1  template <typename T>
2  class Manager {
3      T* managed;
4
5      Manager() = delete;
6      Manager(T* t) : managed(t) { }
7      ~Manager(){ delete managed; }
8  };

```

```

1  template <typename T>
2  void compute_manager(Manager<T> m) { ←
3      // Do something
4  } ←
5
6  int main(){
7      Dog* d = new Dog();
8      Manager managed_dog(d);
9      compute_manager(m);
10     do_other_stuff(m); ←
11     //...oh dangnabbit
12 } ←

```

Copy is made here...
...and deleted here

Use-after-free!

Double free!

Many more issues!!

Fundamental Problem

Having multiple managers who are...

- 1. Managing the same entity**
- 2. Not talking to each other**

...is a terrible idea.

This is true in pretty much any situation ever.

How do we fix this?

Only one manager can ever exist for a managed object!



std::unique_ptr

Let the managers talk to each other!



std::shared_ptr

std::unique_ptr

std::unique_ptr

A built-in way of dynamically managing the *lifetime* of an object.

- Wraps a pointer to a heap-allocated object
- **Only one** unique_ptr can ever refer to a given heap object
- Once the unique_ptr goes out of scope, the object is deleted.

How do we enforce the requirement that only one manager exists at once?

Enforcing Uniqueness

There are two ways that a unique pointer could become non-unique:

```
1 Dog* dog = new Dog()  
2 std::unique_ptr<Dog> p1(dog);  
3 std::unique_ptr<Dog> p2(p1);
```

By copying an existing unique_ptr

```
1 Dog* dog = new Dog()  
2 std::unique_ptr<Dog> p1(dog);  
3 std::unique_ptr<Dog> p2(dog);
```

By reusing a raw pointer

Eliminating Copies of unique_ptr

How do you make a class uncopyable?

```
unique_ptr {
    T* ptr;
    ...
    unique_ptr(const unique_ptr& other) = delete;
    unique_ptr& operator=(const unique_ptr& other) = delete;

    unique_ptr(unique_ptr&& other){
        std::swap(this->ptr, other.ptr)
    }
    unique_ptr& operator=(unique_ptr&& other){
        std::swap(this->ptr, other.ptr);
    }
    ~unique_ptr(){ delete this->ptr; }
```

Now this class can only be moved, not copied.

How do we solve the problem of someone reusing a pointer?

```
1 Dog* dog = new Dog()  
2 std::unique_ptr<Dog> p1(dog);  
3 std::unique_ptr<Dog> p2(dog);
```

Can't really `_(ツ)_/`

The solution would be to use rvalue references, but there are reasons why we can't do this.

std::make_unique

However, we *can* encourage the use of a factory function which disallows this issue!

```
1 struct X{  
2     X();  
3     X(int, int);  
4 };
```

```
1 std::unique_ptr<X> xp = std::make_unique<X>(2, 3);
```

The arguments to `make_unique` are passed through to the constructor of `X`.

Example

```
1 struct Dog {
2     Dog(std::string x);
3 };
4
5 void pet_dog(std::unique_ptr<Dog> d){
6     // Blah blah blah
7 }
8
9 int main(){
10     auto x = std::make_unique<Dog>("Spot");
11
12     pet_dog(x); // Illegal, attempts to copy construct!
13
14     pet_dog(std::move(x)); // Okay!
15
16     pet_dog(make_unique<Dog>("Toby")); // Also okay!
17 }
18
```

What about the other solution?

Managers talk to each other?

std::shared_ptr

std::shared_ptr

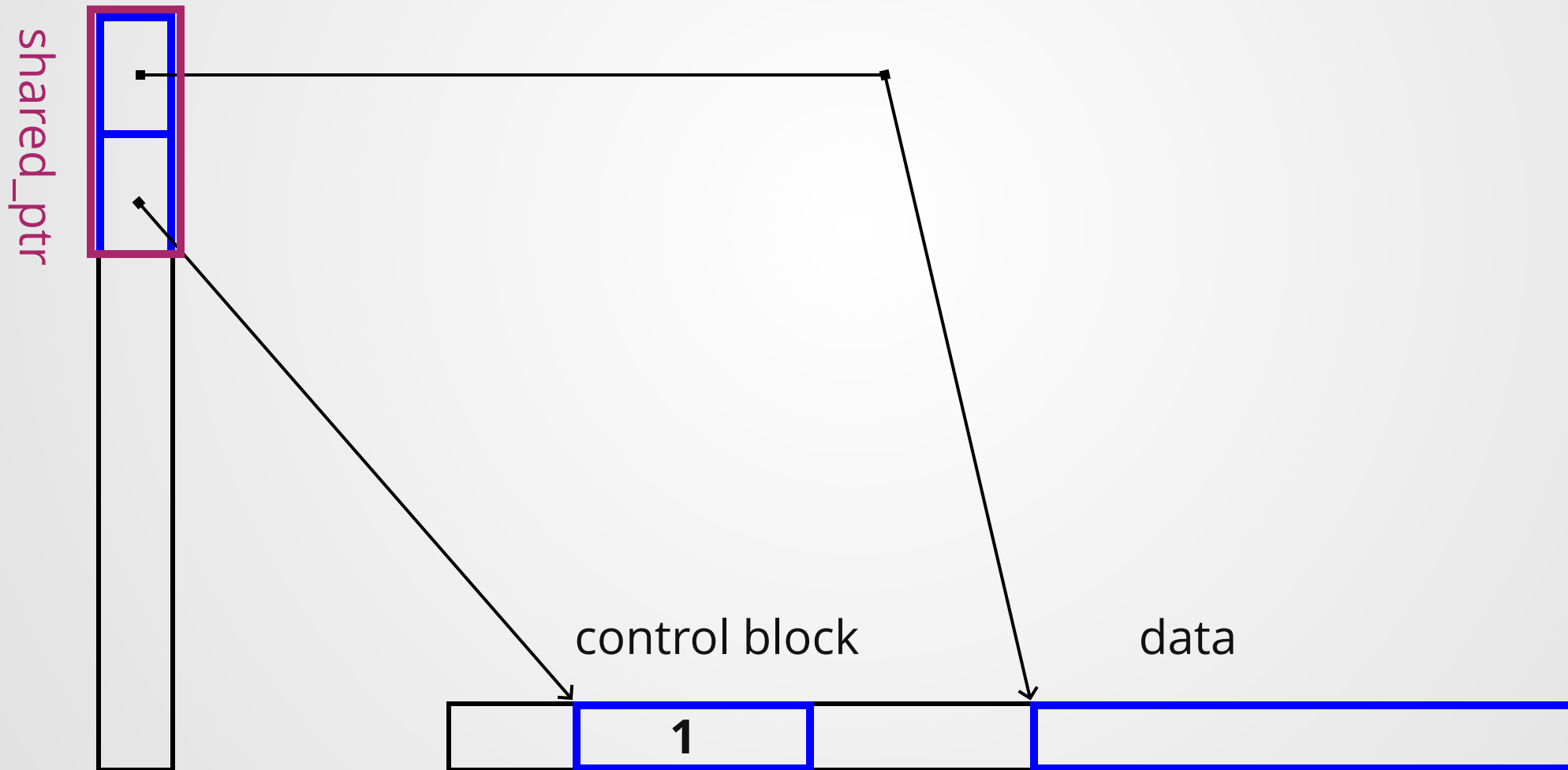
A built-in way of dynamically managing the *lifetime* of an object.

- Wraps a pointer to a heap-allocated object
- **Multiple** shared_ptrs can exist to the same object.
- Once the ***last*** shared_ptr goes out of scope, the object is deleted.

How do we enforce the requirement that the object can only be deleted when the **last** shared pointer is gone?

Anatomy of a shared_ptr

The simplest implementation of shared_ptr is a pair of pointers: one points to the *data*, and one points to a *control block*



What's in a control block?

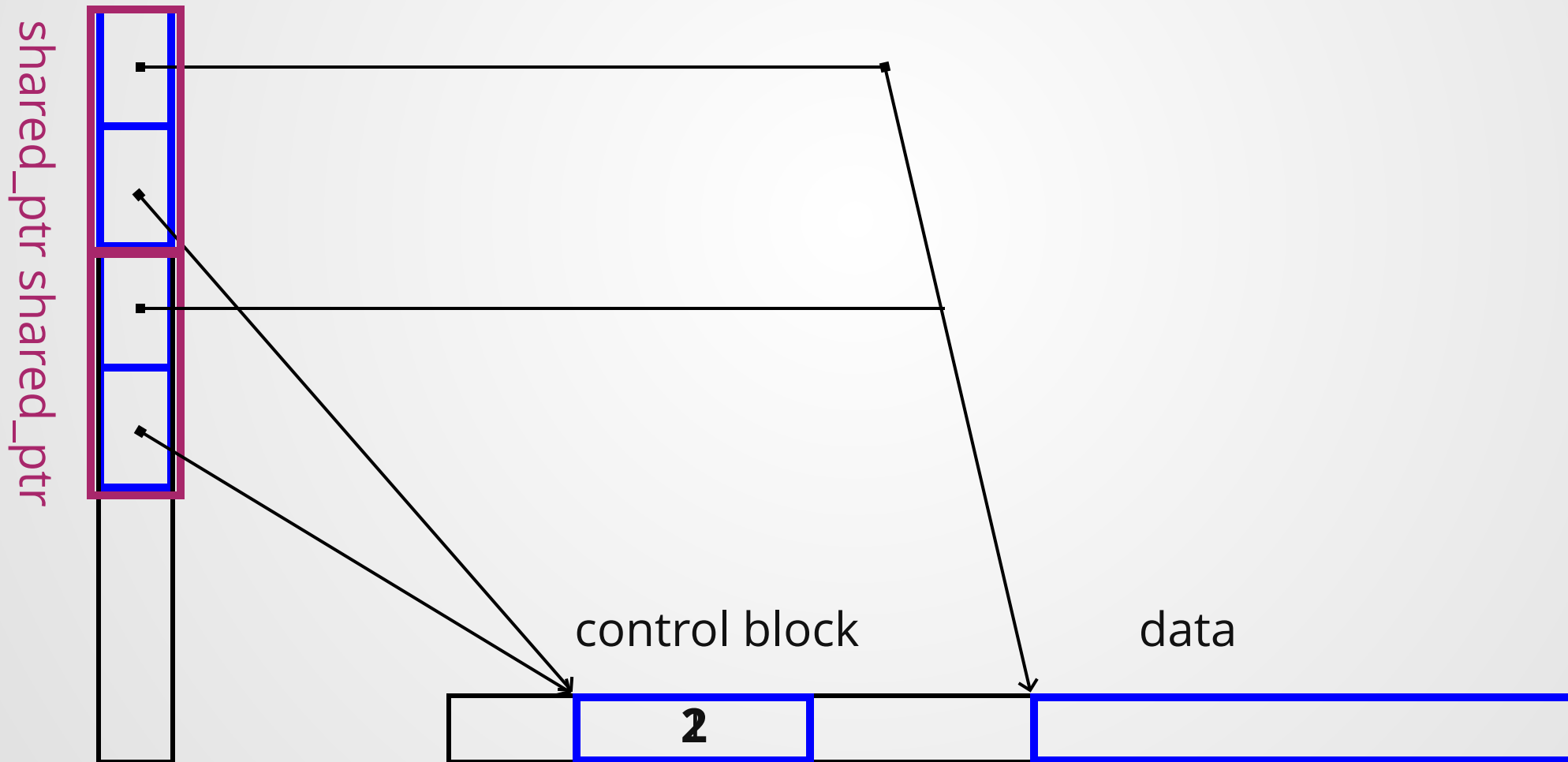
Lots of things:

- Pointer to data
- Memory Allocator
- Memory Deleter
- **Number of shared pointers referring to the object**



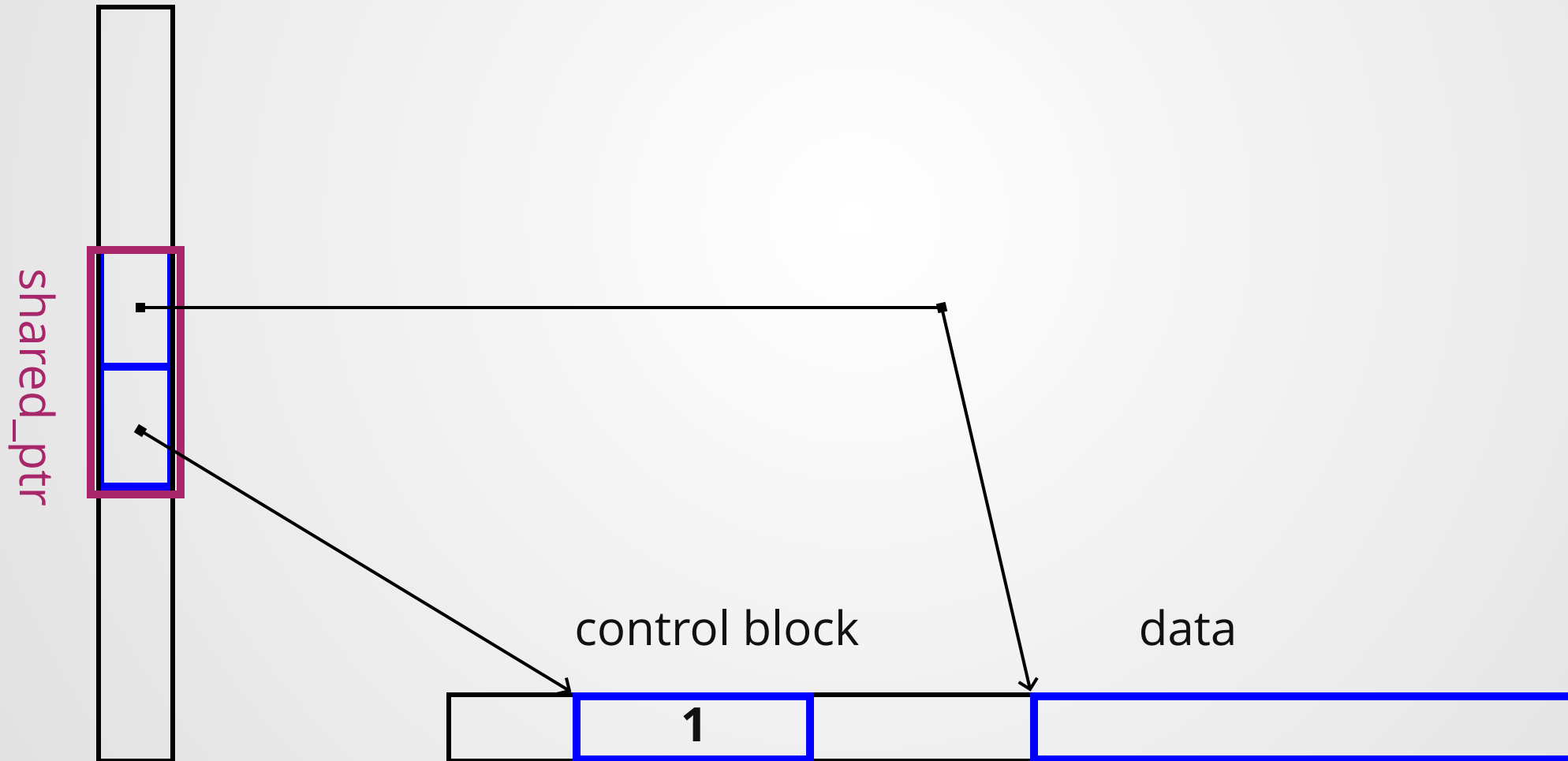
Anatomy of a shared_ptr

When a new shared_ptr is copy constructed, increase the control block counter (first copy the control block pointer, then use it to increment counter)



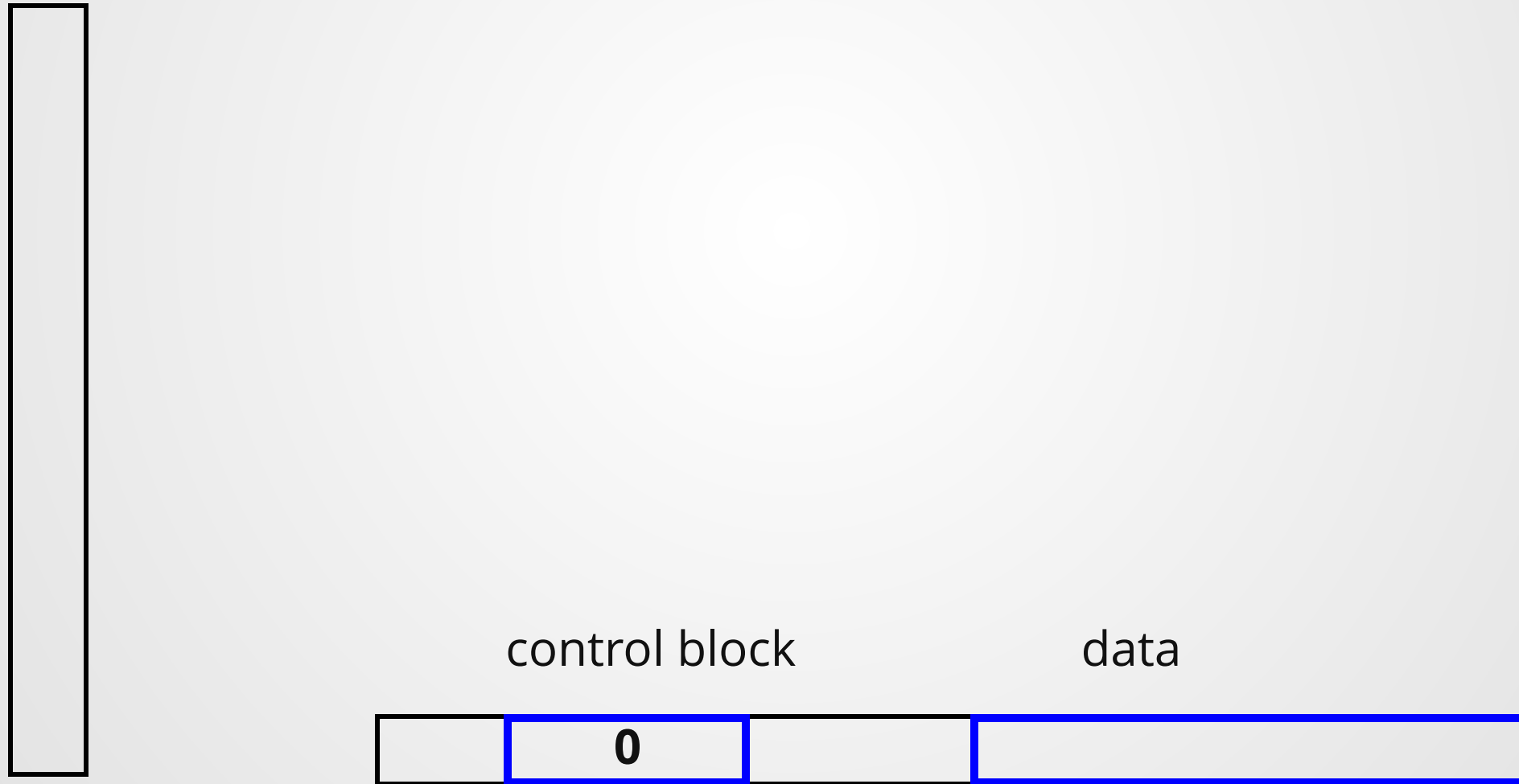
Anatomy of a shared_ptr

When a shared_ptr is deleted, decrement the control block counter in the destructor.



Anatomy of a shared_ptr

When the counter reaches zero, delete the control block and the data!



shared_ptr *should* be copied!

The whole point of this class is that once there are no copies of the shared_ptr left, the managed memory cleans itself up!

Copy shared pointers around all you want!

Also, read the [ISO C++ guidelines](#), which has lots of great advice on how to use smart pointers (see section "R").

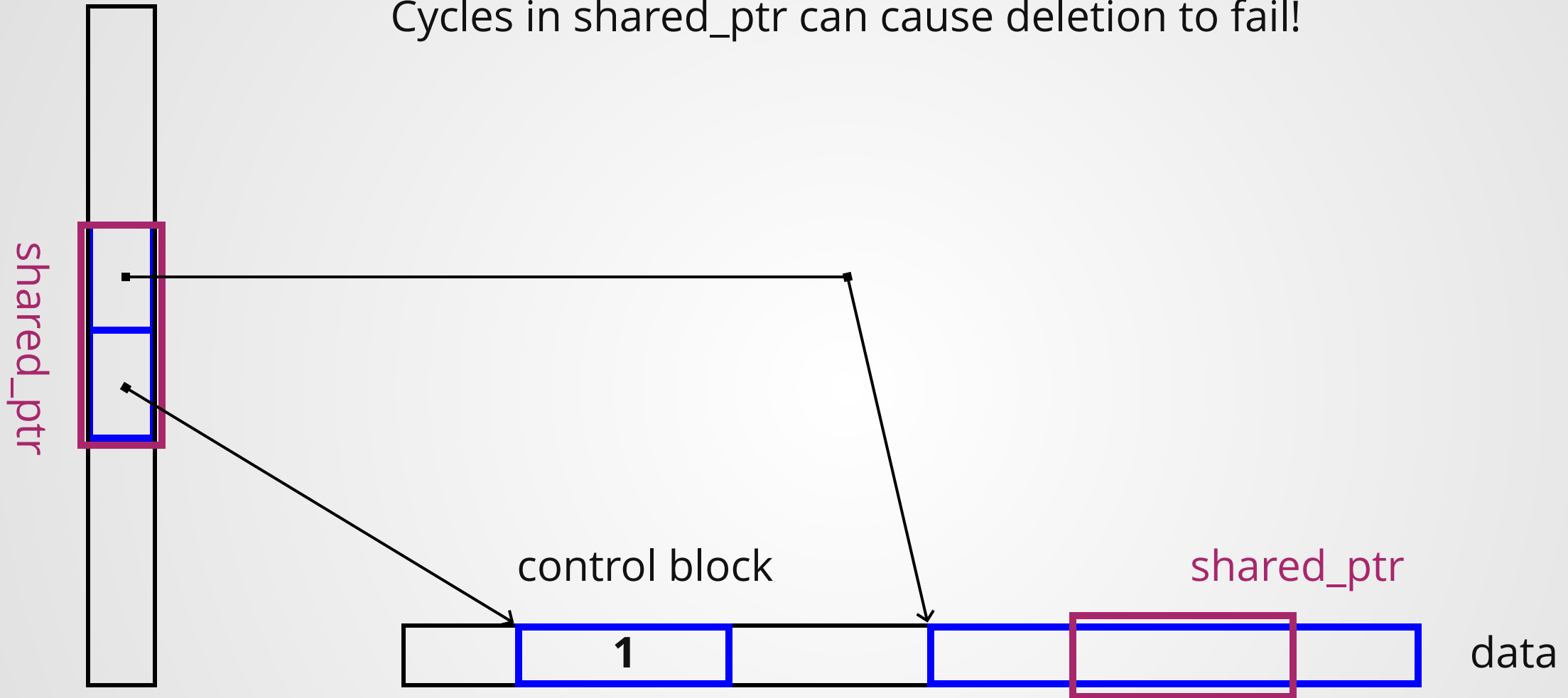
std::make_shared

Works just like std::make_unique

```
1 struct Dog {
2     Dog(std::string x);
3 };
4
5 void pet_dog(std::shared_ptr<Dog> d){
6     // Blah blah blah
7 }
8
9 int main(){
10     auto x = std::make_shared<Dog>("Spot");
11
12     pet_dog(x); // Okay! In fact, shared pointers should be copied!
13
14     pet_dog(std::move(x)); // Also okay!
15
16     pet_dog(make_shared<Dog>("Toby")); // Also okay!
17 }
18
```

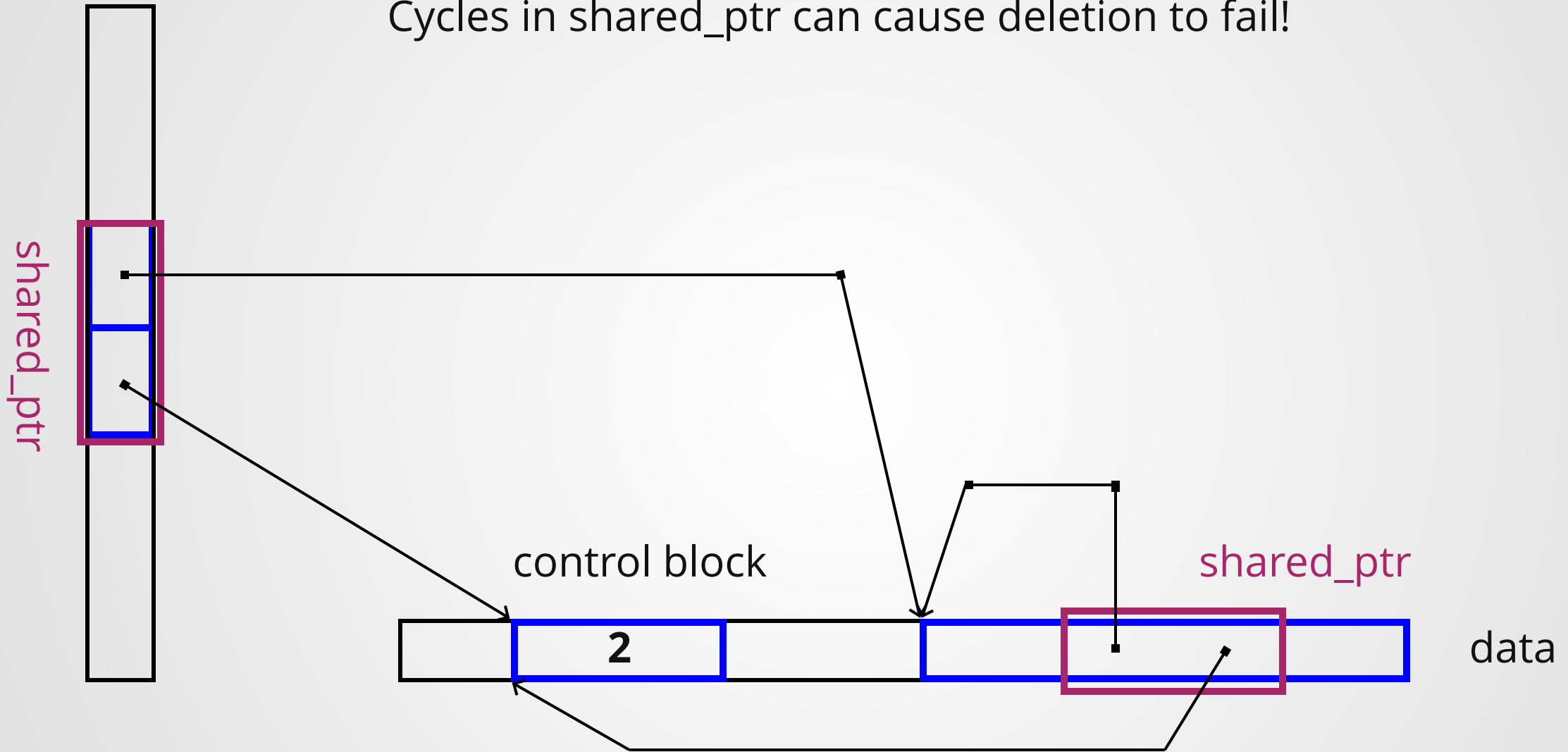
Reference Cycles

Cycles in shared_ptr can cause deletion to fail!



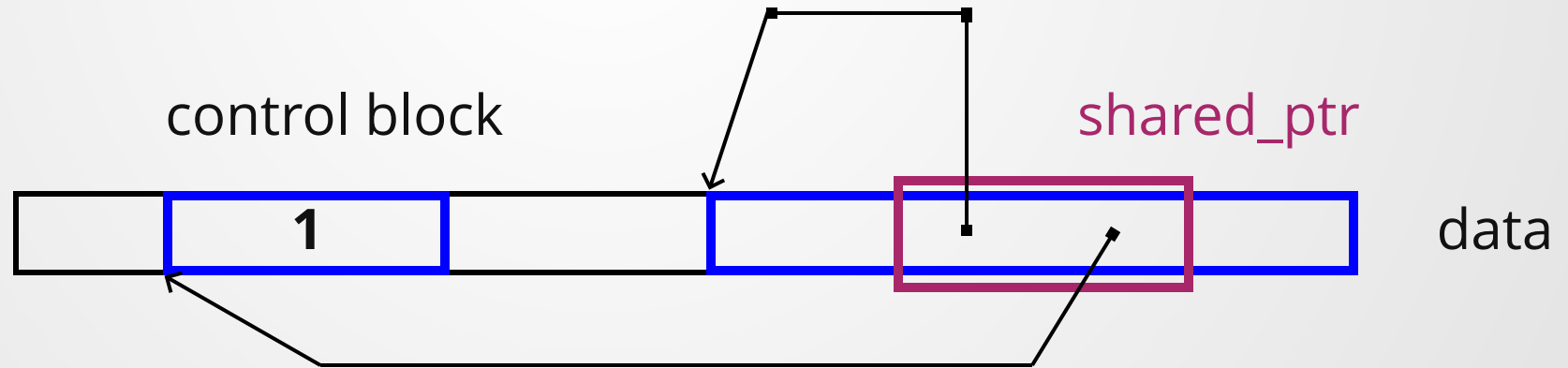
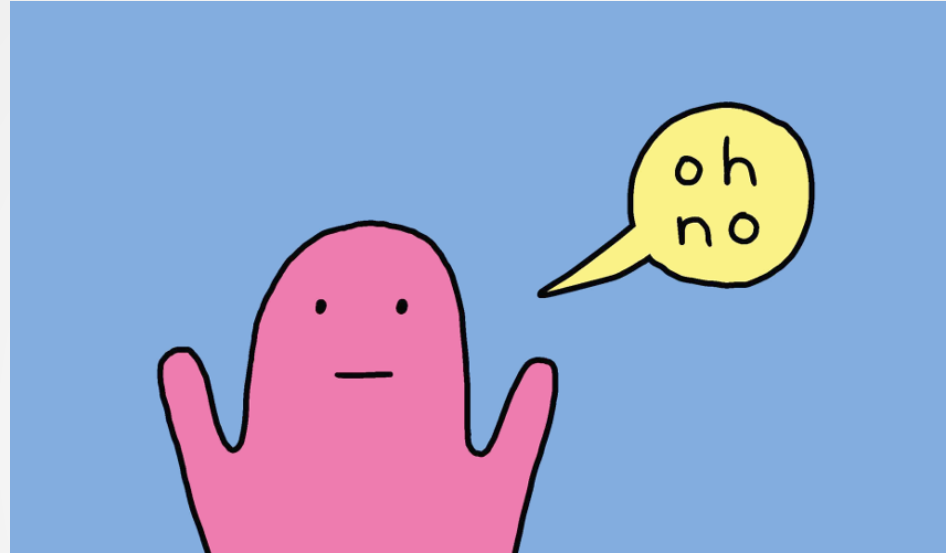
Reference Cycles

Cycles in shared_ptr can cause deletion to fail!



Reference Cycles

This can happen with any cycle, not just self-cycles!



Any cycle of `shared_ptr`s can cause memory leaks.

To fix this, use a `weak_ptr` somewhere in the cycle.

For those of you that have been asking about garbage collection in C++: the `shared_ptr` effectively implements reference-counting GC

The Forwarding Problem

What does `make_shared` look like?

```
1  template <typename T, typename U>
2  std::shared_ptr<T> make_shared(U&& arg){
3      T* obj = new T(arg);
4      return std::shared_ptr<T>(obj);
5  }
```

NB: this technically could result in violation of the "two shared pointers" rule, but since it's a very short function, it's relatively safe.

However, it breaks on very important functionality!

Trying to write factory functions like `make_shared` in the presence of the rvalue/lvalue distinction can be tricky!

```
1 class MoveOnly {
2     MoveOnly() = default;
3     MoveOnly(const MoveOnly&) = delete;
4     MoveOnly& operator=(const MoveOnly&) = delete;
5     MoveOnly(MoveOnly&&) = default;
6     MoveOnly& operator=(MoveOnly&&) = delete;
7 };
```

```
1 template <typename T, typename U>
2 std::shared_ptr<T> make_shared(U&& arg){
3     T* obj = new T(arg);
4     return std::shared_ptr<T>(obj);
5 }
```

```
1 int main(){
2     MoveOnly a;
3     make_shared<MoveOnly>(a);
4     make_shared<MoveOnly>(MoveOnly());
5 }
```

The first call is obviously illegal.
What about the second?

Let's say we can solve that problem

```
1 class MoveOrCopy {
2     MoveOnly() = default;
3     MoveOnly(const MoveOnly&) = default;
4     MoveOnly& operator=(const MoveOnly&) = default;
5     MoveOnly(MoveOnly&&) = default;
6     MoveOnly& operator=(MoveOnly&&) = default;
7 };
```

```
1 template <typename T, typename U>
2 std::shared_ptr<T> make_shared(U&& arg){
3     T* obj = new T(arg);
4     return std::shared_ptr<T>(obj);
5 }
```

```
1 int main(){
2     MoveOnly a;
3     make_shared<MoveOnly>(a);
4     make_shared<MoveOnly>(MoveOnly());
5 }
```

The first call is still illegal! The call is on an lvalue, and U&& is an rvalue reference.

Solution: Forwarding References*

```
1 template <typename T>
2 std::shared_ptr<T> make_shared(const Animal& in){
3     T* obj = new T(in);
4     return std::shared_ptr<T>(obj);
5 }
```

When the form `T&&` appears in a type and T is subject to type inference (i.e. is `auto` or a template parameter), then T&& is *not* an rvalue reference.

It is a special type of reference called a *forwarding reference* that preserves the type it was initialized with:

- If it is initialized from an rvalue, then it is an rvalue reference
- If it is initialized from an lvalue, then it is an lvalue reference.

*Sometimes called "universal references"

Example: Forwarding References*

```
1 struct Animal {
2     Animal(int& x){ std::cout << "Calling lvalue ref constructor\n";}
3     Animal(int&& x){ std::cout << "Calling rvalue ref constructor\n";}
4 };
5
6 template <typename T, typename U>
7 std::shared_ptr<T> make_shared(U&& in){
8     T* obj = new T(in);
9     return std::shared_ptr<T>(obj);
10 }
```

```
1 int main(){
2     int i = 3;
3     auto x = make_shared<Animal>(i);
4 }
5
6 /*
7 > ./a.out
8 > Calling lvalue ref constructor
9 */
```

```
1 int main(){
2     auto x = make_shared<Animal>(3);
3 }
4
5 /*
6 > ./a.out
7 > Calling lvalue ref constructor
8 */
```

...wait, what?

Remember: *rvalue references are lvalues!*

We can preserve the lvalue/rvalue-ness of a forwarding reference by using `std::forward`

```
1 struct Animal {
2     Animal(int& x){ std::cout << "Calling lvalue ref constructor\n";}
3     Animal(int&& x){ std::cout << "Calling rvalue ref constructor\n";}
4 };
5
6 template <typename T, typename U>
7 std::shared_ptr<T> make_shared(U&& in){
8     T* obj = new T(std::forward<U>(in));
9     return std::shared_ptr<T>(obj);
10 }
```

```
1 int main(){
2     int i = 3;
3     auto x = make_shared<Animal>(i);
4 }
5
6 /*
7 > ./a.out
8 > Calling lvalue ref constructor
9 */
```

```
1 int main(){
2     auto x = make_shared<Animal>(3);
3 }
4
5 /*
6 > ./a.out
7 > Calling rvalue ref constructor
8 */
```

At some point in project 3, if you haven't made some weird design decisions, you may find that some of your code does not work when passed rvalues, and that fixing it makes it not work when passed lvalues.

If that happens...look back over this section.

Summary

Having multiple non-communicating managers of an entity is almost always a bad idea!

```
1 template <typename T>
2 class Manager {
3     T* managed;
4
5     Manager() = delete;
6     Manager(T* t) : managed(t) { }
7     ~Manager(){ delete managed; }
8 };
```

```
1 template <typename T>
2 void compute_manager(Manager<T> m){
3     // Do something
4 }
5
6 int main(){
7     Dog* d = new Dog();
8     Manager managed_dog(d);
9     compute_manager(m);
10
11     //...oh dangnabbit
12 }
```

We can manage objects (lifetimes) in C++ using two smart pointer classes, which solve the non-communicating manager problem .

Only one manager can ever exist for a managed object!



std::unique_ptr

Let the managers talk to each other!



std::shared_ptr

std::unique_ptr

unique_ptr solves the problem by only letting one unique_ptr own an object.

unique_ptr cannot be copied, only moved!

```
unique_ptr {  
    T* ptr;  
    ...  
    unique_ptr(const unique_ptr& other) = delete;  
    unique_ptr& operator=(const unique_ptr& other) = delete;
```

Use std::make_unique to create a unique_ptr without danger of double-owning a raw pointer.

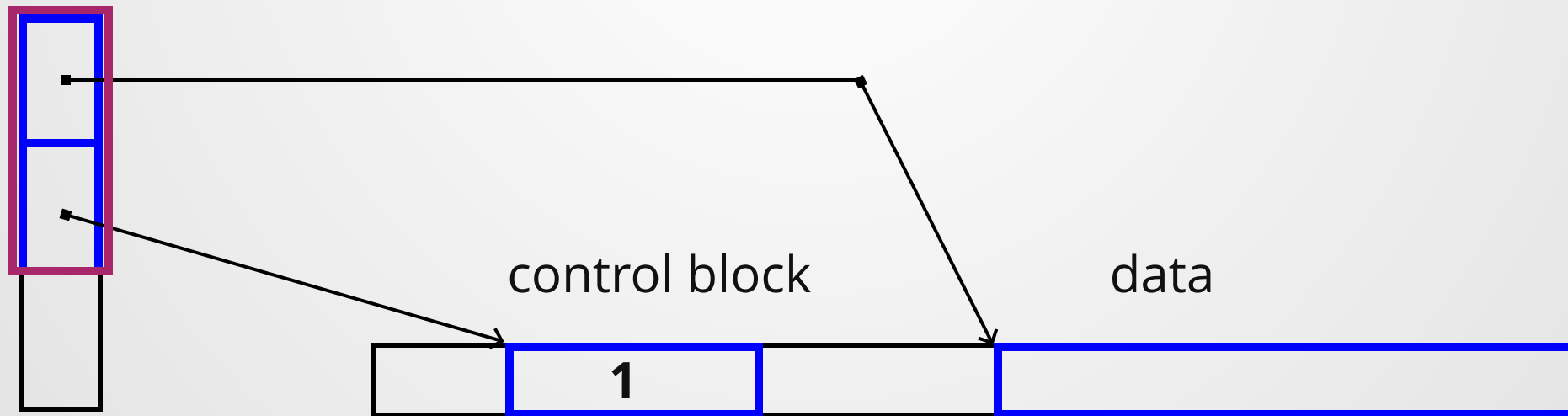
std::shared_ptr

shared_ptr solves the problem by counting how many copies of the shared_ptr exist.

When the shared_ptr count hits zero, the object auto-destructs.

Reference cycles may prevent shared_ptrs from destructing, leaking memory.

Use std::make_shared to create a shared_ptr without danger of double-owning a raw pointer.



Forwarding

There are two problems when trying to pass rvalue/lvalue references through intermediate functions to their appropriate constructors:

- RValue references are lvalues
- Sometimes we want to take both rvalues and lvalues

To solve this, we have a *forwarding reference*: a reference declared with T&& where T is type-deduced.

Forwarding references can be passed through a function while retaining their original "value category" (i.e. rvalue/lvalue-ness) with `std::forward`

The Rest of the Class

You now have all the information you need to complete the projects and

Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material. **If you leave this blank, you will be docked points.**

If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.

Additional Resources

Universal References
and Perfect Forwarding

An extended guide on rvalue
references, their motivation, and
the forwarding problem

An explanation of reference
cycles using Python as an
example (python is refcounted,
similar to shared_ptr)

The wikipedia page on smart pointers is
surprisingly good