Smart/Managed Pointers
(A light introduction)

Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan

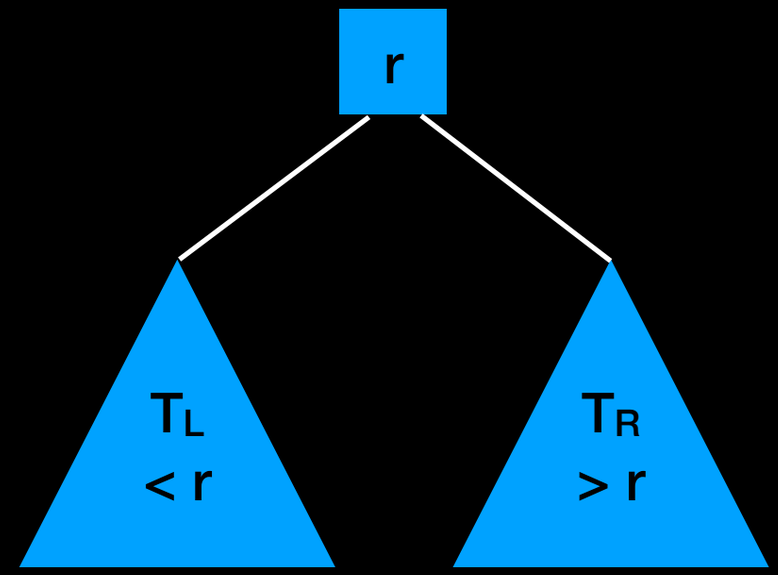
Recap

Motivation

Managed Pointers (light)

Recap: Binary Search Tree

Structural Property:
For each node n
n > all values in T_L
n < all values in T_R




```
search(bs_tree, item)
{
    if (bs_tree is empty) //base case
        item not found
    else if (item == root)
        return root
    else if (item < root)
        search(T_L , item)
    else // item > root
        search(T_R , item)
}
```

Recap: Efficiency of BST

Searching is key to most operations

Think about the structure and height of the tree

O(h)

What is the maximum height?

What is the minimum height?

Managed Pointers Motivation

What happens when program that dynamically allocated memory relinquishes control in the middle of execution because of an exception?

Managed Pointers Motivation

What happens when program that dynamically allocated memory relinquishes control in the middle of execution because of an exception?

Dynamically allocated memory never released!!!

Managed Pointers Motivation

Whenever using dynamic memory allocation and exception handling together must consider ways to prevent memory leaks

Memory Leak

```
template<class T>
T List<T>::getItem(size_t position) const
{
    Node<T>* pos_ptr = getPointerTo(position);
    if(pos_ptr == nullptr)
        throw(std::out_of_range("getItem called with empty list or invalid position"));
    else
        return pos_ptr->getItem();

}
```

out_of_range exception thrown here

```
T someFunction(const List<T>& some_list)
{
    //code here that dynamically allocates memory
    T an_item;
    //code here
    an_item = some_list.getItem(n);
    // delete dynamically allocated memory

}
```

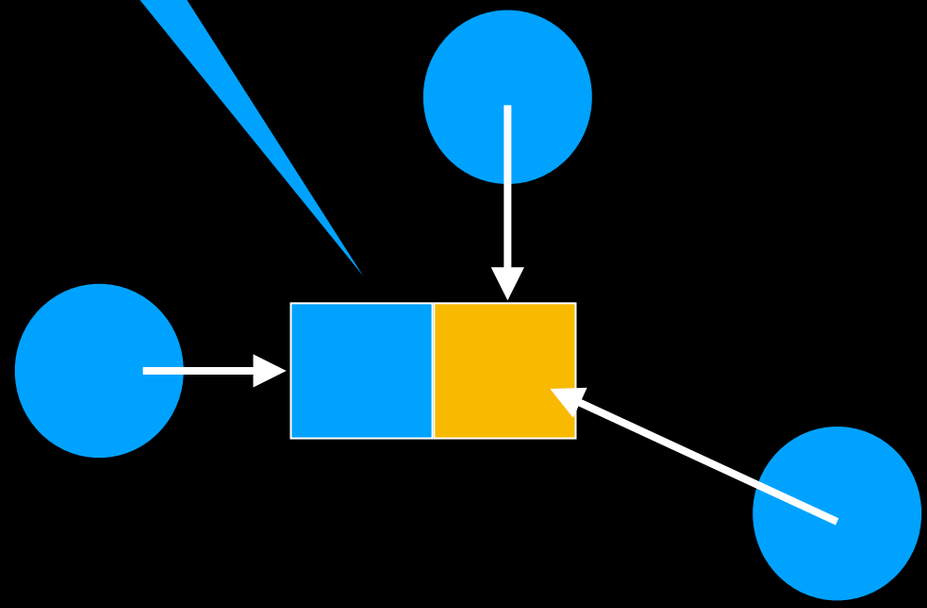
out_of_range exception not handled here

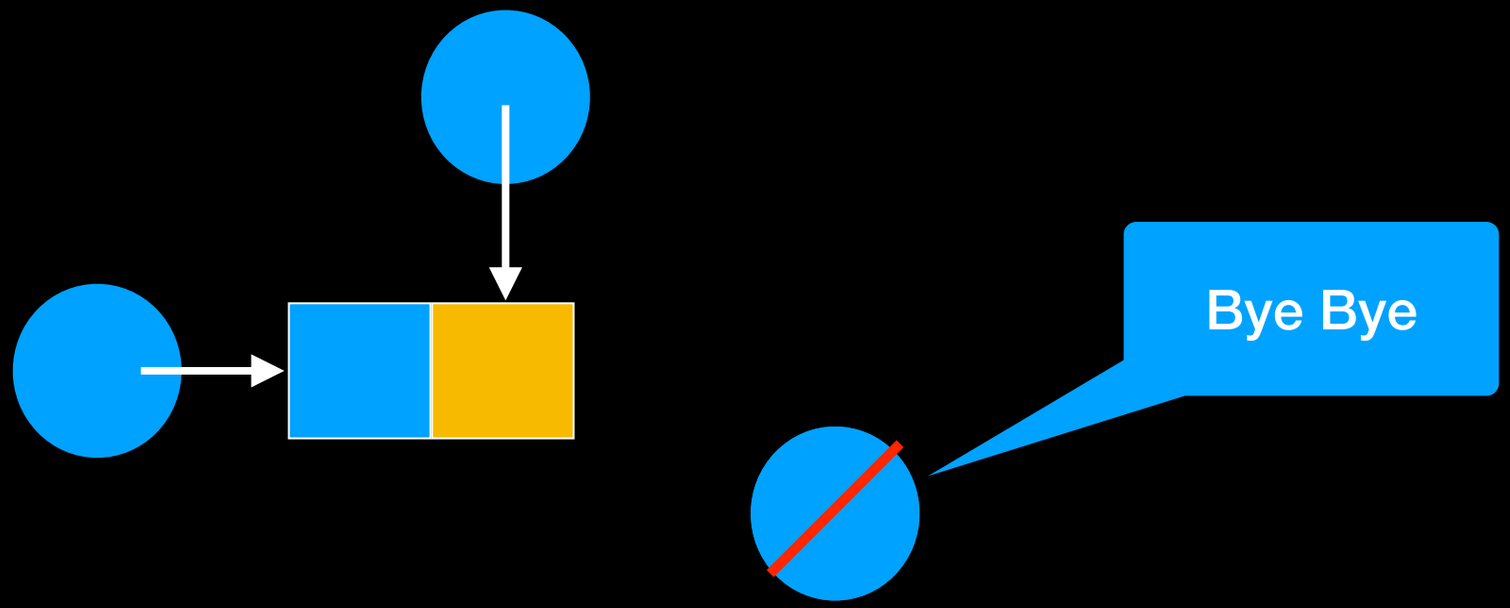
```
int main()
{
    List<string> my_list;
    try
    {
        std::string some_string = someFunction(my_list);
    }
    catch(const std::out_of_range& problem)
    {
        //code to handle exception here
    }
    //more code here
    return 0;
}
```

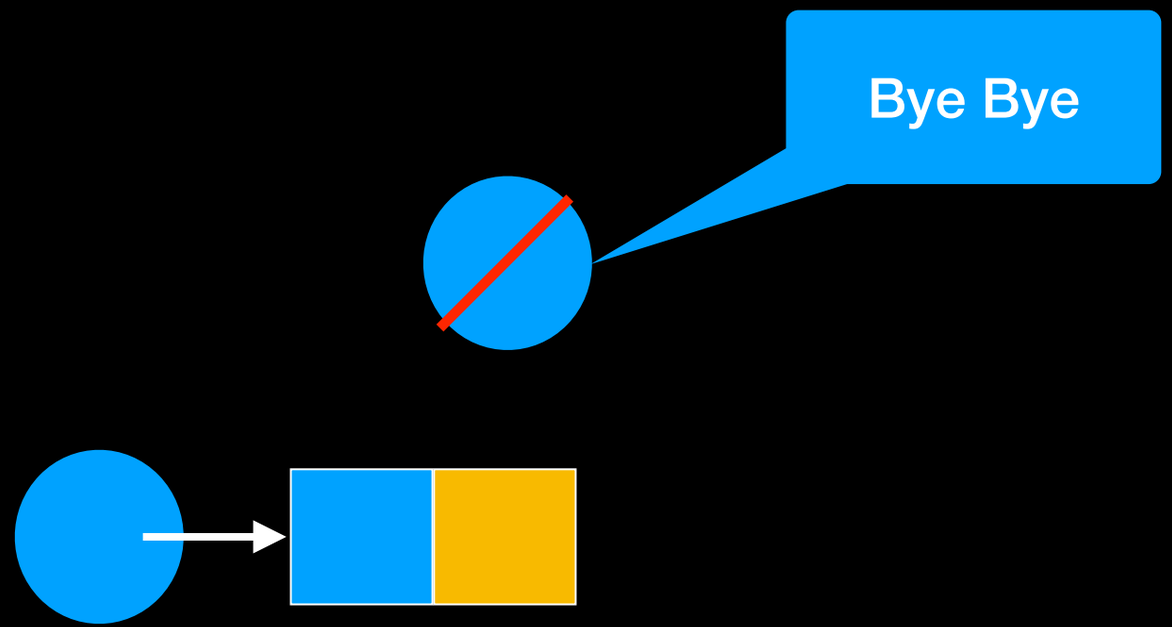
out_of_range exception handled here

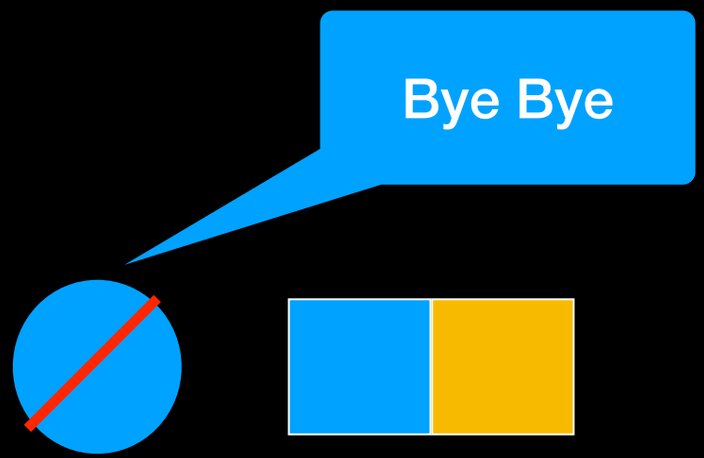
Dynamically allocated object

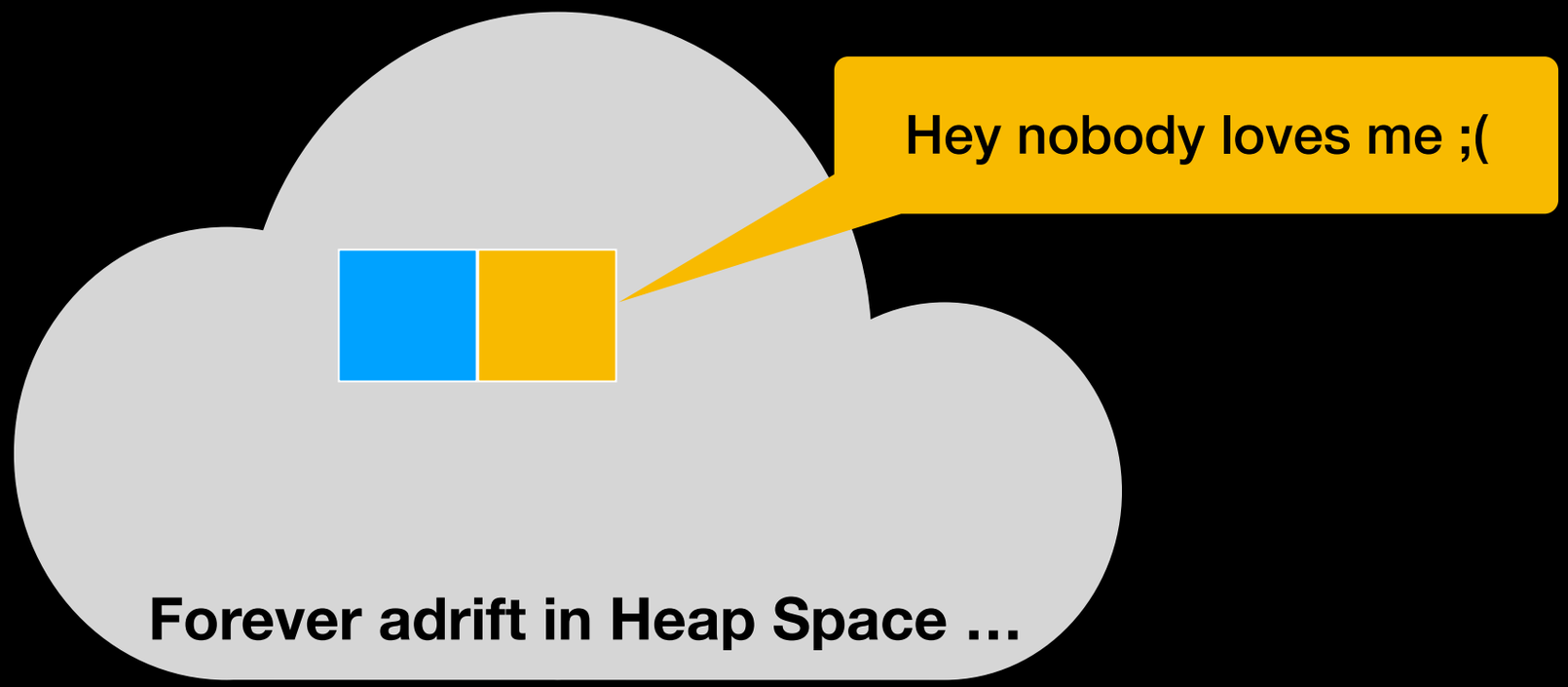
Pointers are not aware of each other

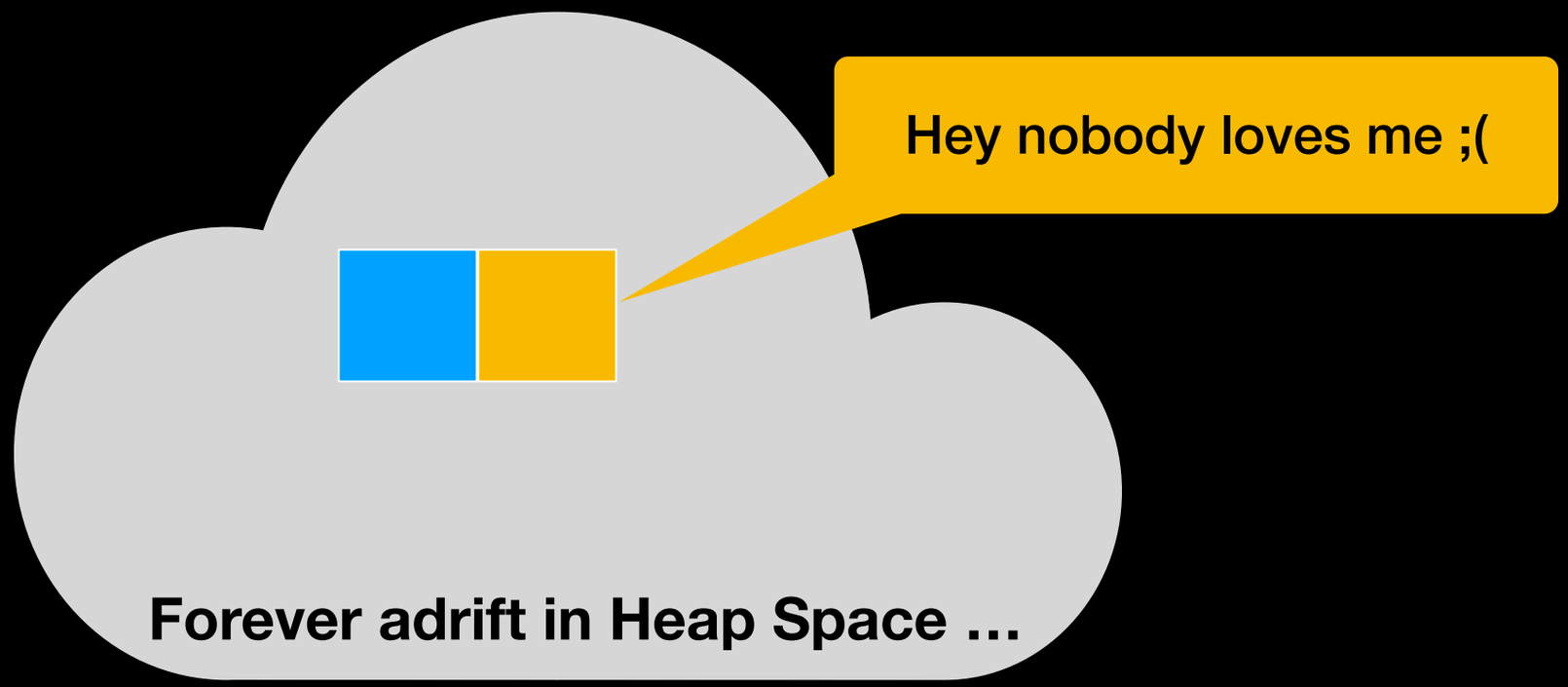



Bye Bye


Bye Bye


Bye Bye


Hey nobody loves me ;(
Forever adrift in Heap Space ...



Programmer responsible for keeping track

Ownership

A pointer is said to **own** a dynamically allocated object if it is responsible for deleting it

If any node is disconnected it is lost on heap

Nodes must be deleted before disconnecting from chain

If multiple pointers point to same node it can be hard to keep track who is responsible for deleting it

Smart/Managed Pointer

A Light Introduction

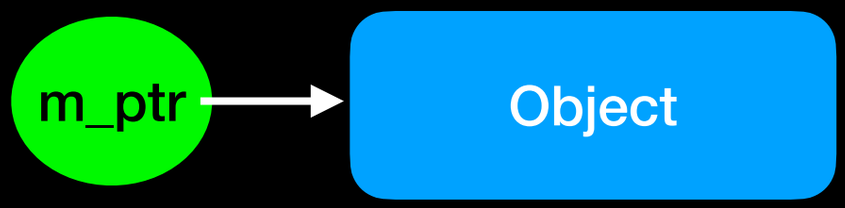
Smart/Managed Pointer

Smart pointer:

- An object
- Acts like a **raw pointer**
-Provides automatic memory management
  (at some performance cost)

Distinguish from"smart"

C++14



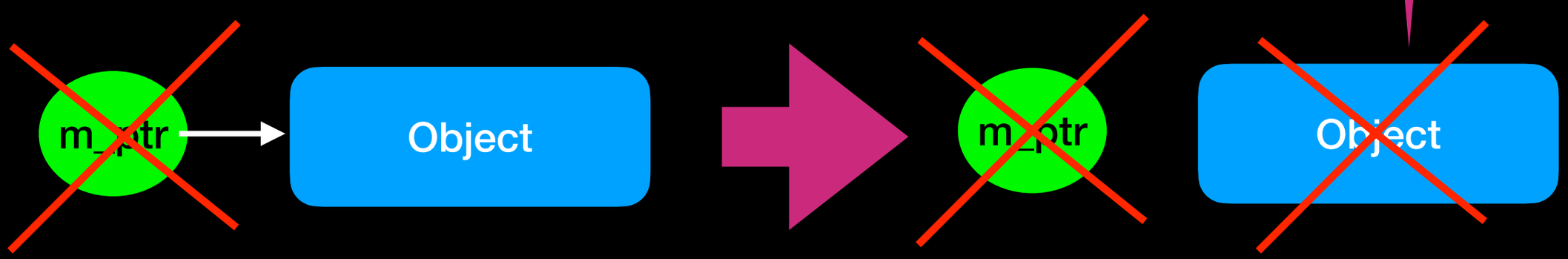
A non-trivial sentence but we will leave it at that

Smart/Managed Pointer

Smart pointer:

- An object
- Acts like a **raw pointer**
-Provides automatic memory management
(at some performance cost)

Smart Pointer destructor automatically invokes destructor of object it points to



Smart/Managed Pointers

Smart pointer ownership = object's destructor automatically invoked when pointer goes out of scope or set to `nullptr`

3 types:

- `shared_ptr`
- `unique_ptr`
- `weak_ptr`

Shared ownership: keeps track of # of pointers to one object. The last one must delete object

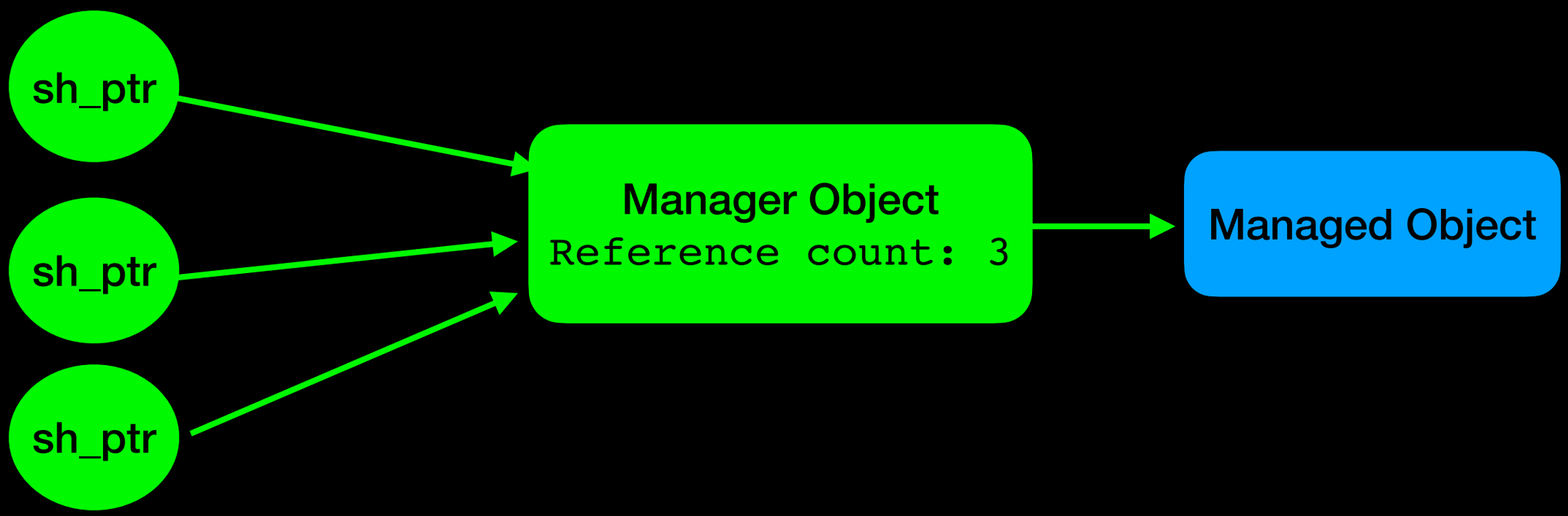
Unique ownership: only smart pointer allowed to point to the object

Points but **does not own**

Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object



Smart/Managed Pointers

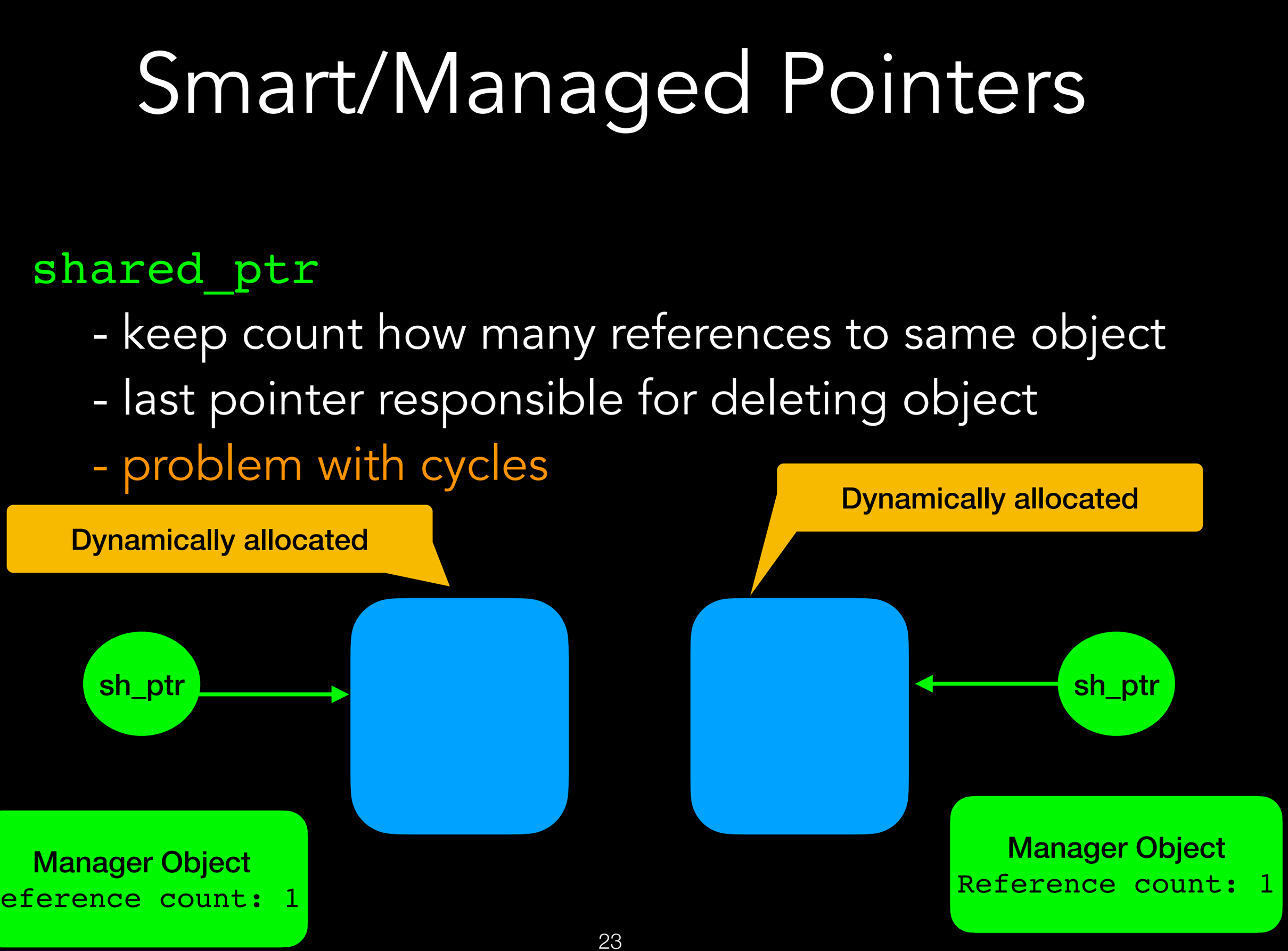
`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

Smart/Managed Pointers

`shared_ptr`

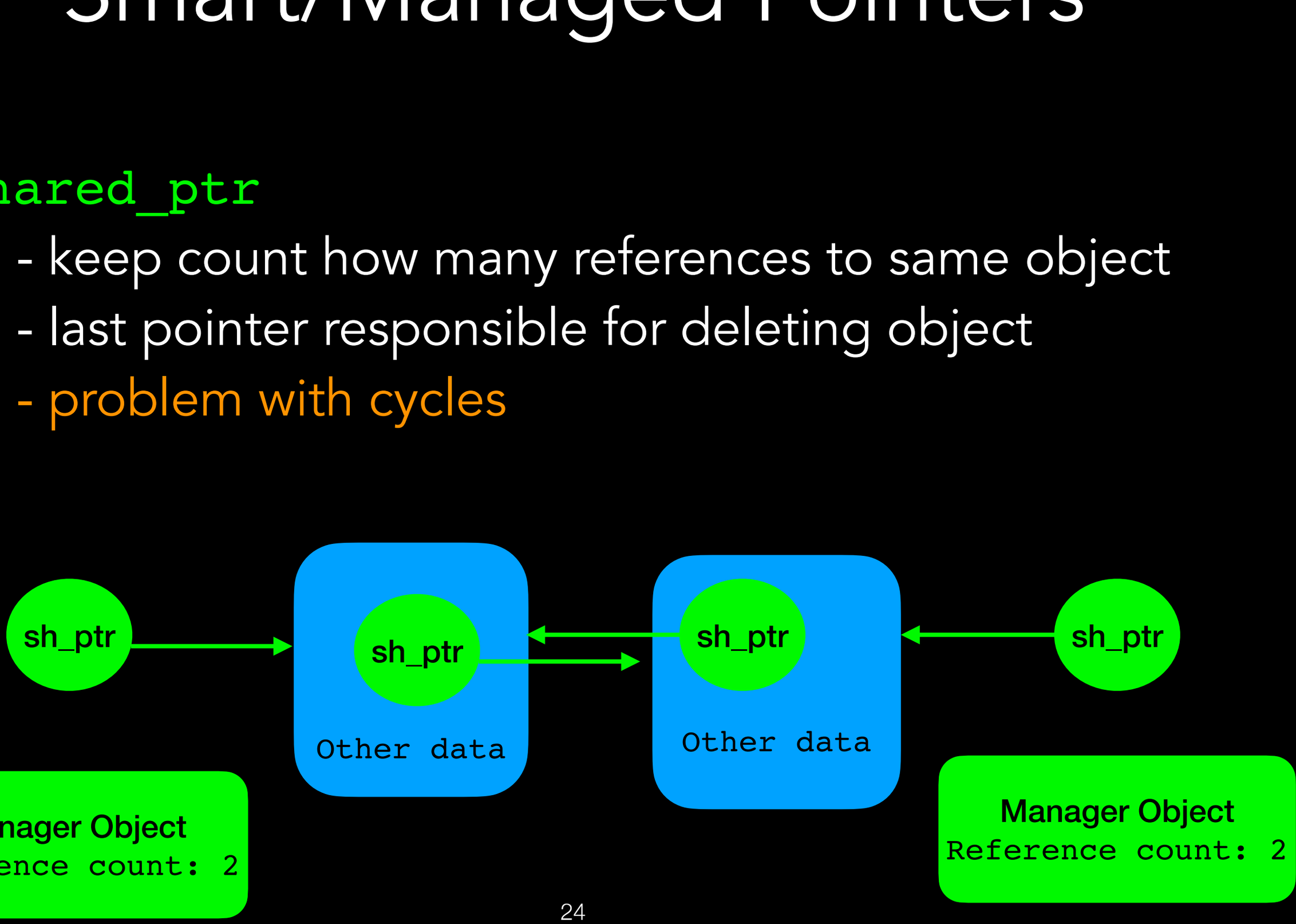
- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles



Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

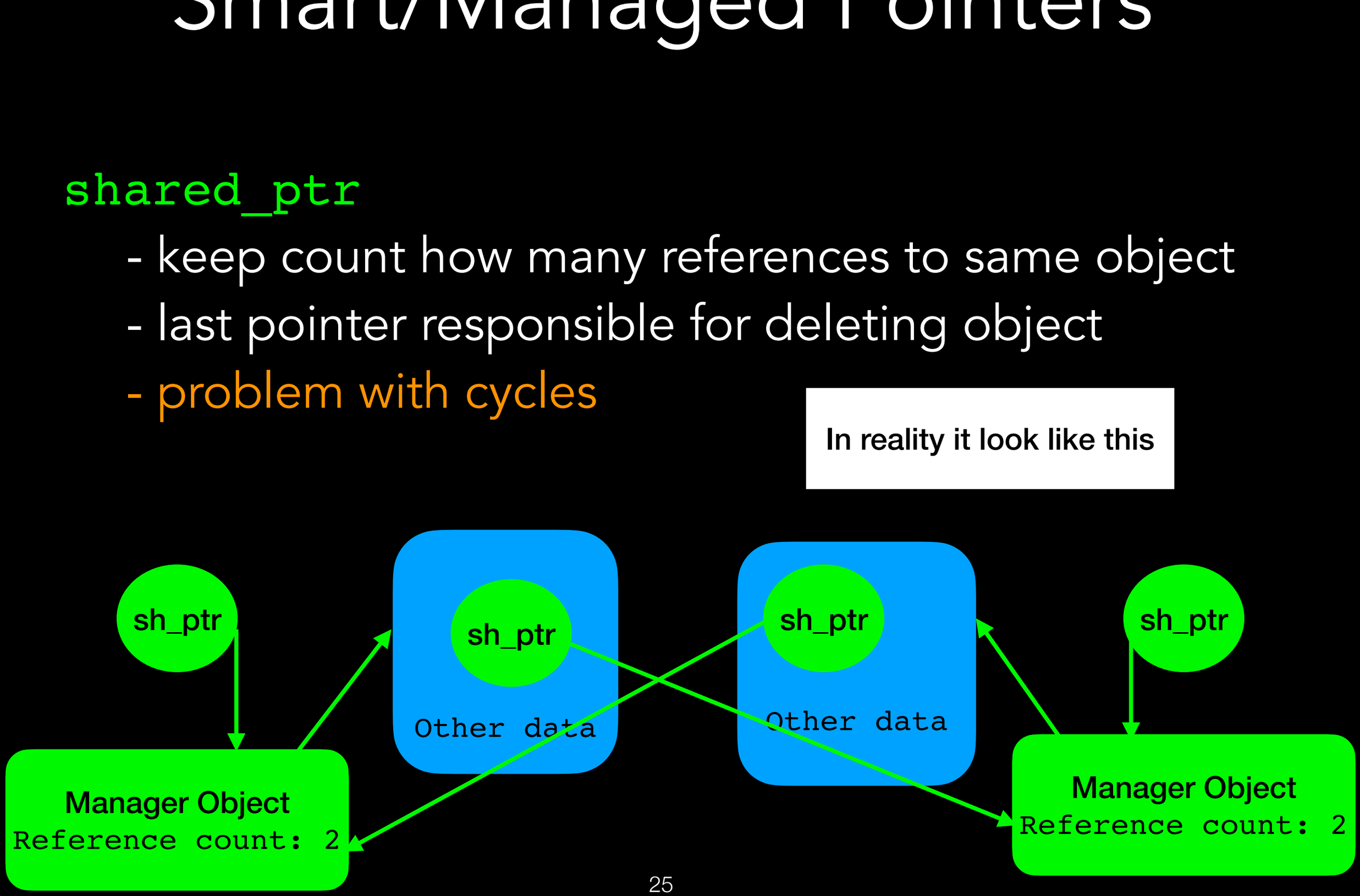


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

In reality it look like this

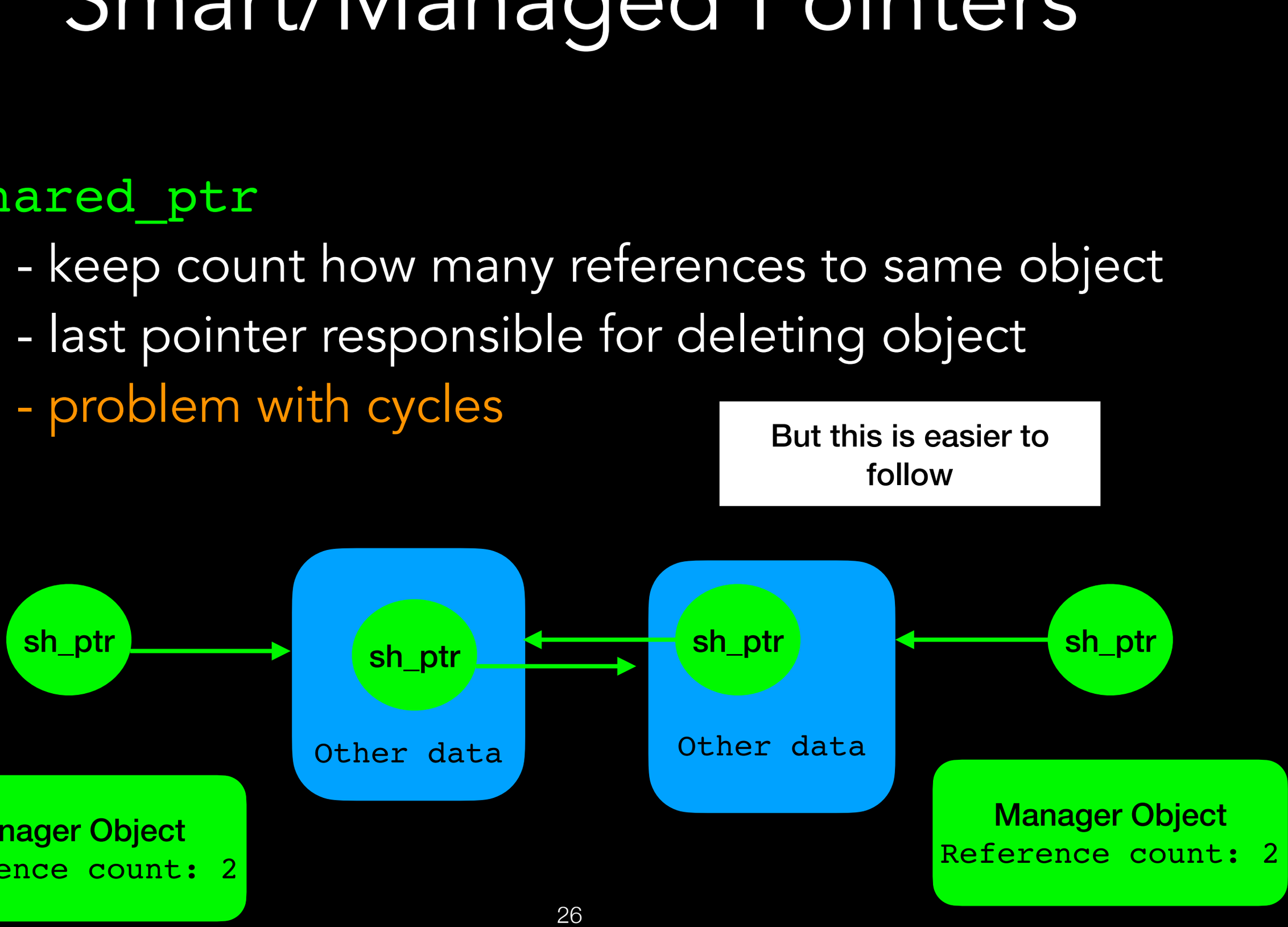


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

But this is easier to follow

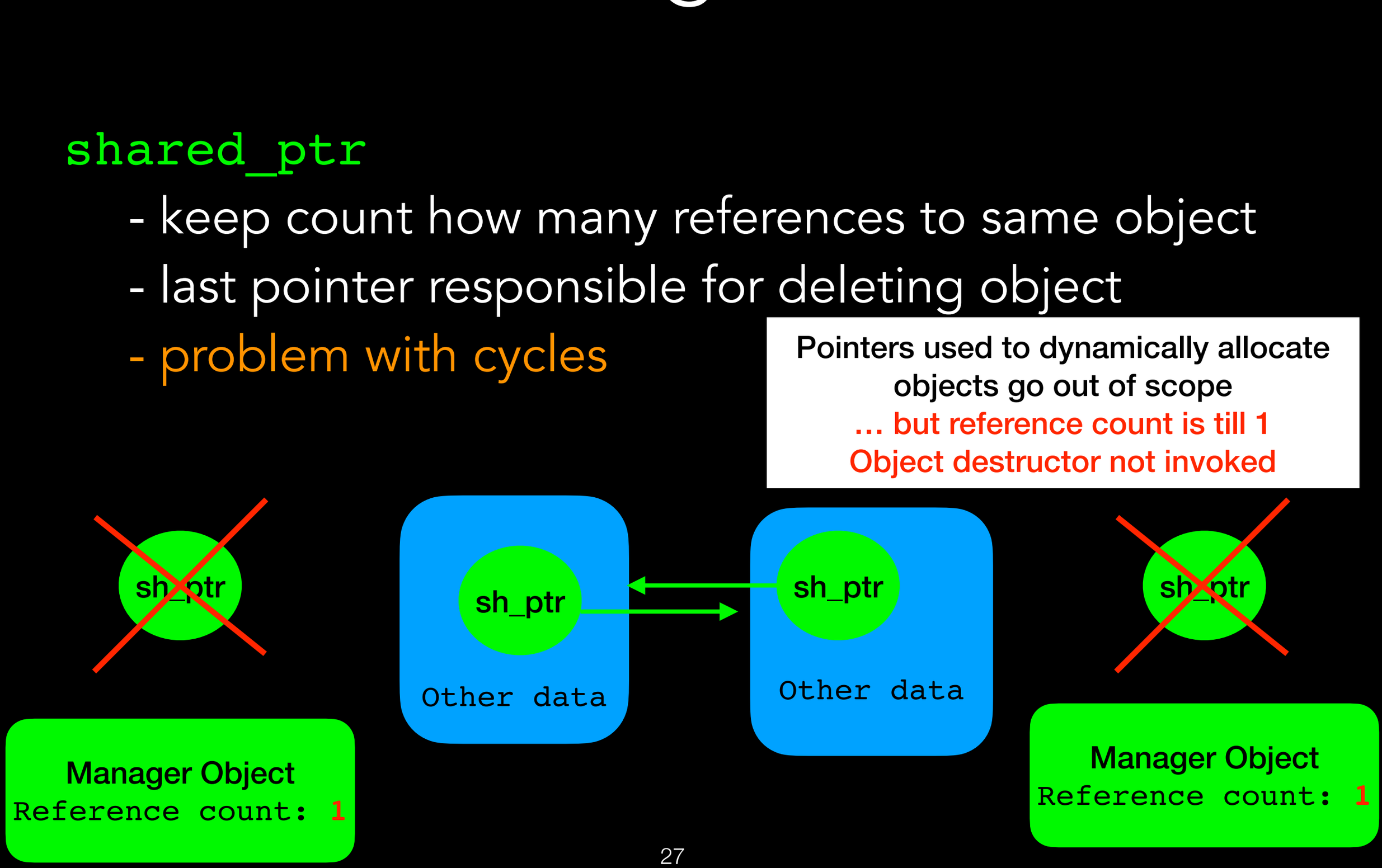


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

Pointers used to dynamically allocate objects go out of scope
… but reference count is till 1
Object destructor not invoked

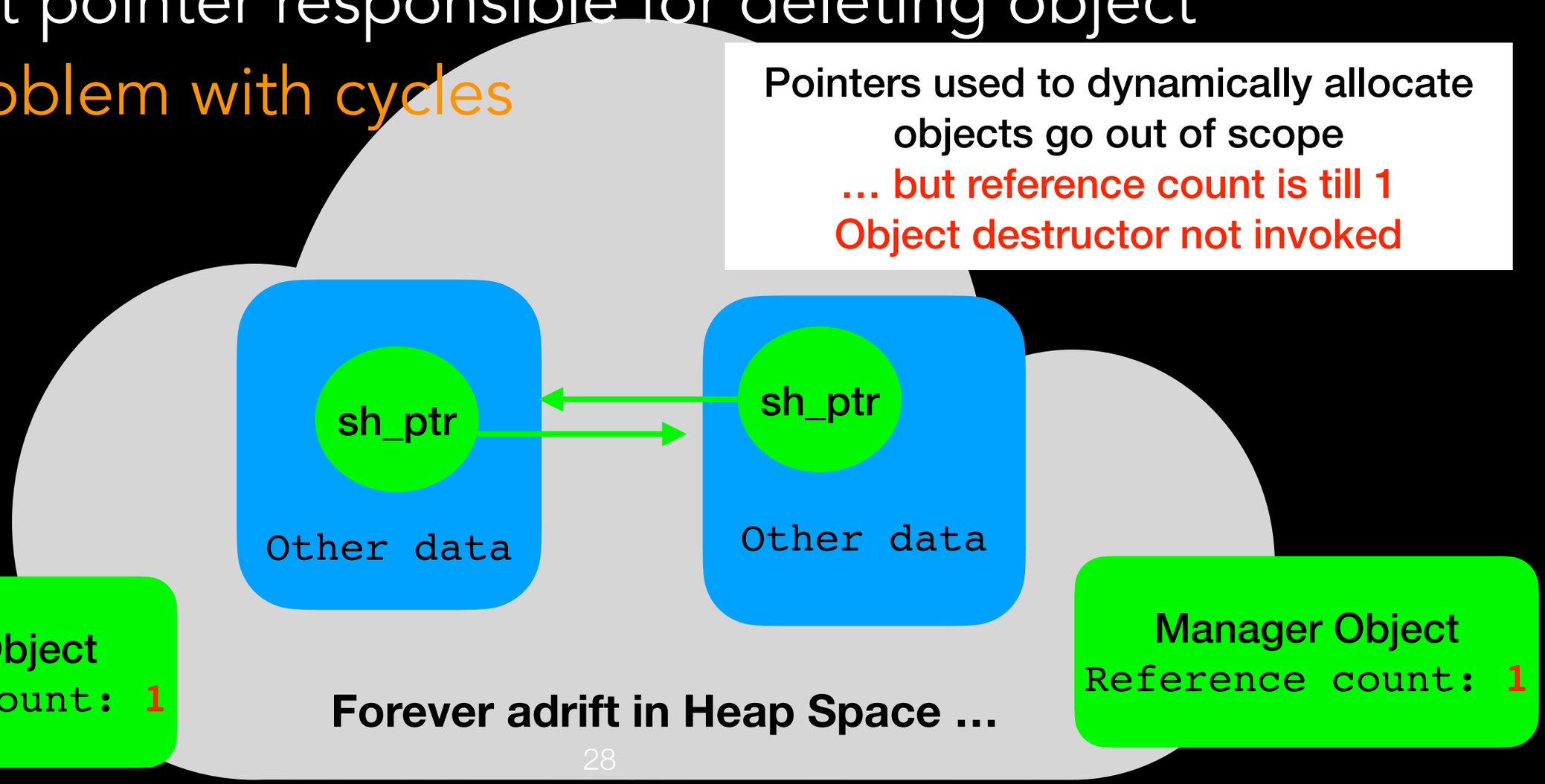


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

Pointers used to dynamically allocate objects go out of scope
… but reference count is till 1
Object destructor not invoked

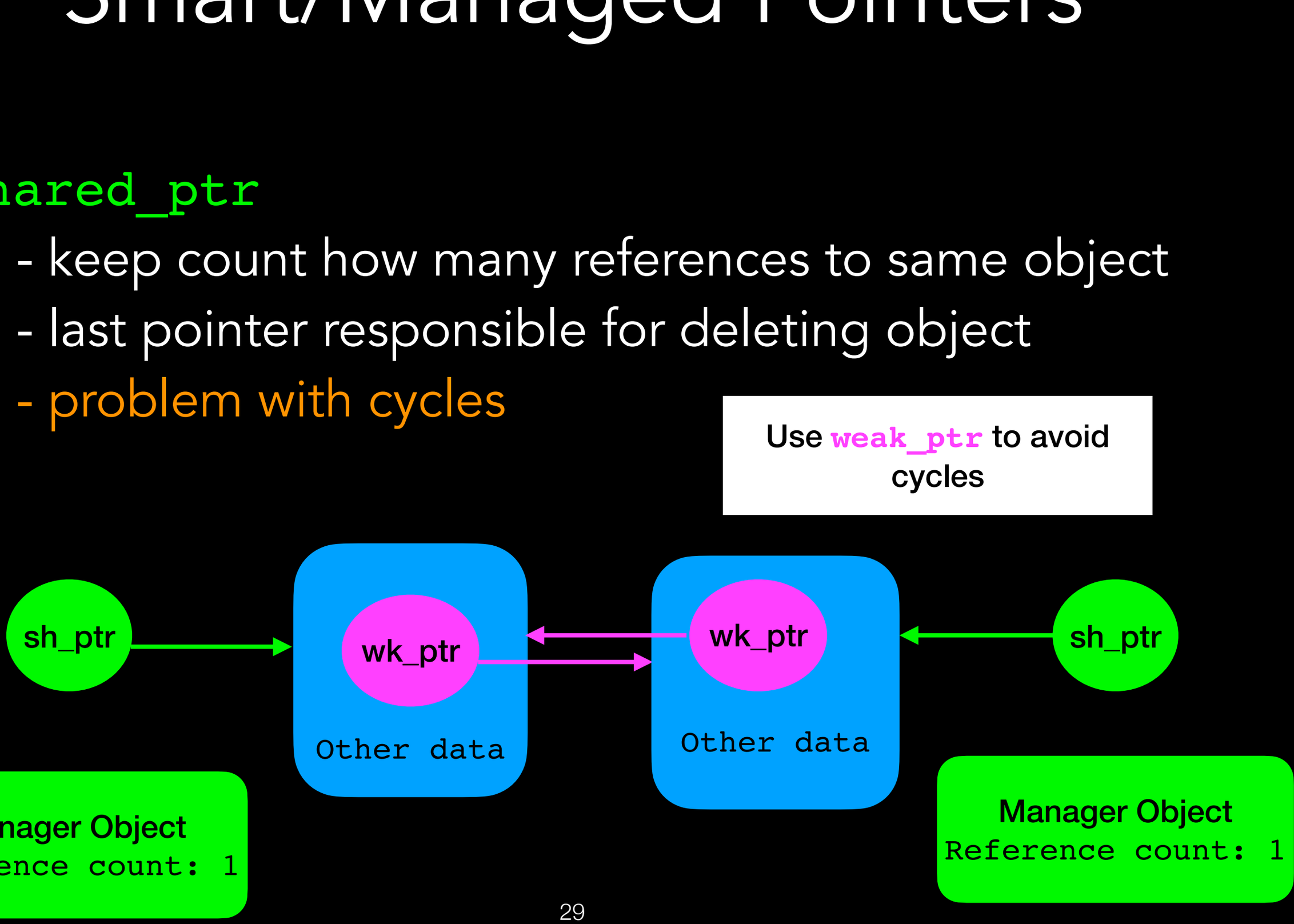


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

Use `weak_ptr` to avoid cycles

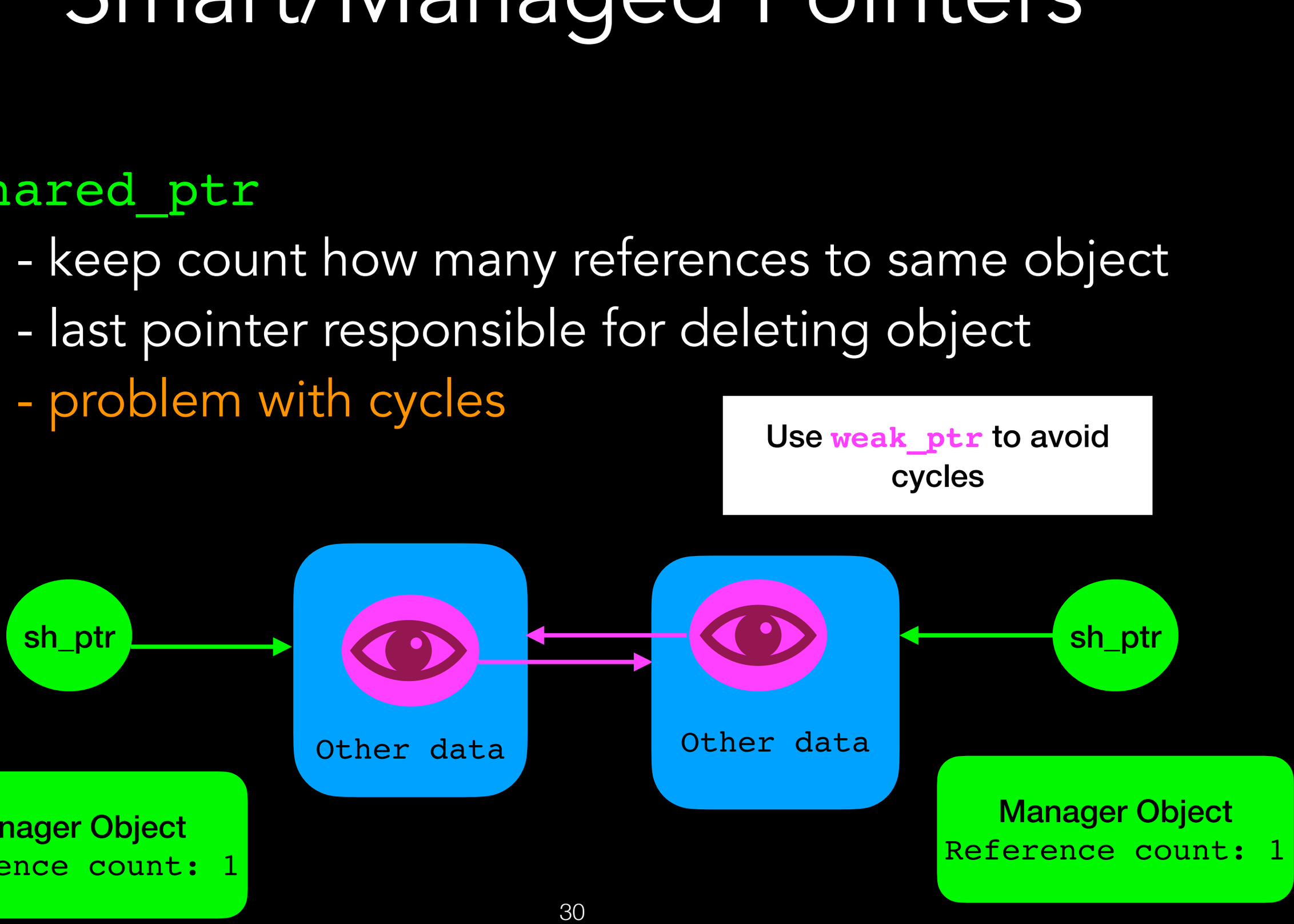


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

Use `weak_ptr` to avoid cycles

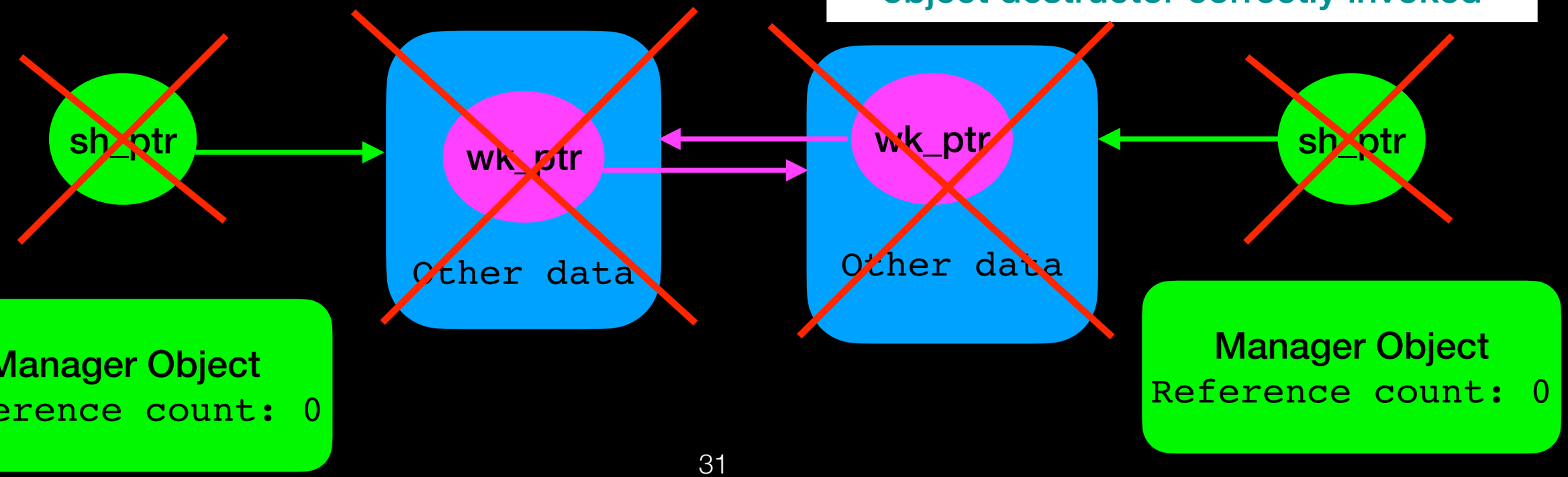


Smart/Managed Pointers

`shared_ptr`

- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles

Pointers used to dynamically allocate objects go out of scope
Reference count goes to 0 and object destructor correctly invoked



Syntax

shared_ptr

```
std::shared_ptr<Song> song_ptr1; //declaration only automatically set to nullptr

auto song_ptr2 = std::make_shared<Song>();// equivalent to Song* ptr = new Song
                                          //but creates manager and object in single memory allocation

    // do stuff

std::cout << song_ptr2->getTitle() << std::endl;
```

Syntax

`auto` says: "compiler you figure out the correct type based on what is returned by function on rhs of =

`shared_ptr`

More efficient
Do it this way

```
std::shared_ptr<Song> song_ptr1; //declaration only automatically set to nullptr

auto song_ptr2 = std::make_shared<Song>();// equivalent to Song* ptr = new Song()
                                          //but creates manager and object in single memory allocation

    // do stuff

std::cout << song_ptr2->getTitle() << std::endl;

song_ptr2.reset();
```

Use it just like you would a raw pointer

Set the `shared_ptr` to `nullptr` with `reset()` and it will take care of deleting the dynamic object.

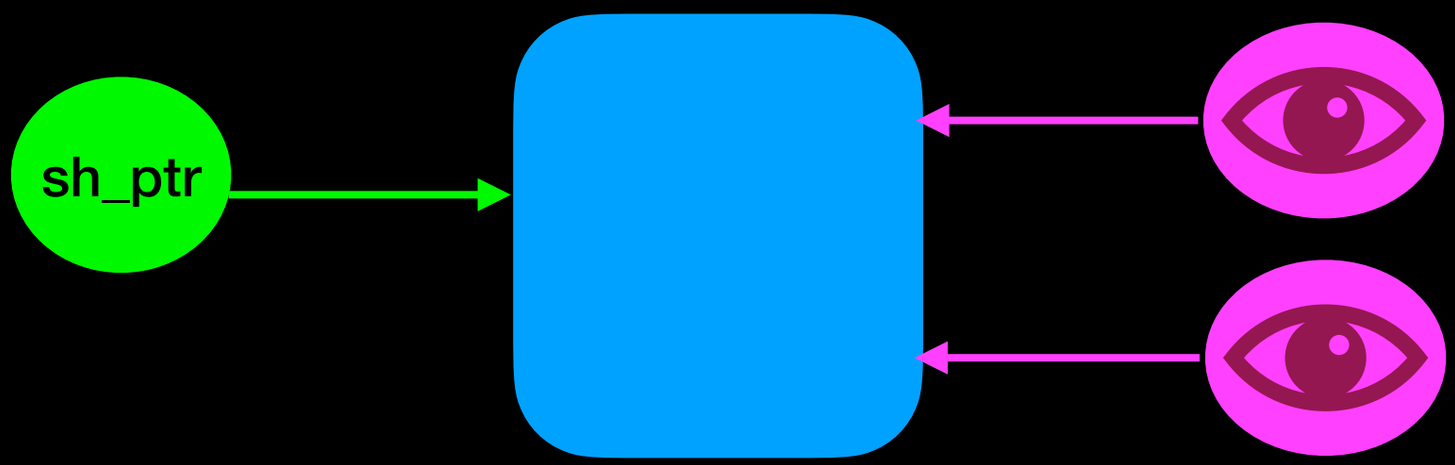
Syntax

weak_ptr cannot own object, so cannot be used to allocate a new object — must allocate new object through shared or unique

weak_ptr

```
auto shared_song_ptr = std::make_shared<Song>();

std::weak_ptr<Song> weak_song_ptr1 = shared_song_ptr;
auto weak_song_ptr2 = weak_song_ptr1;
```



Syntax

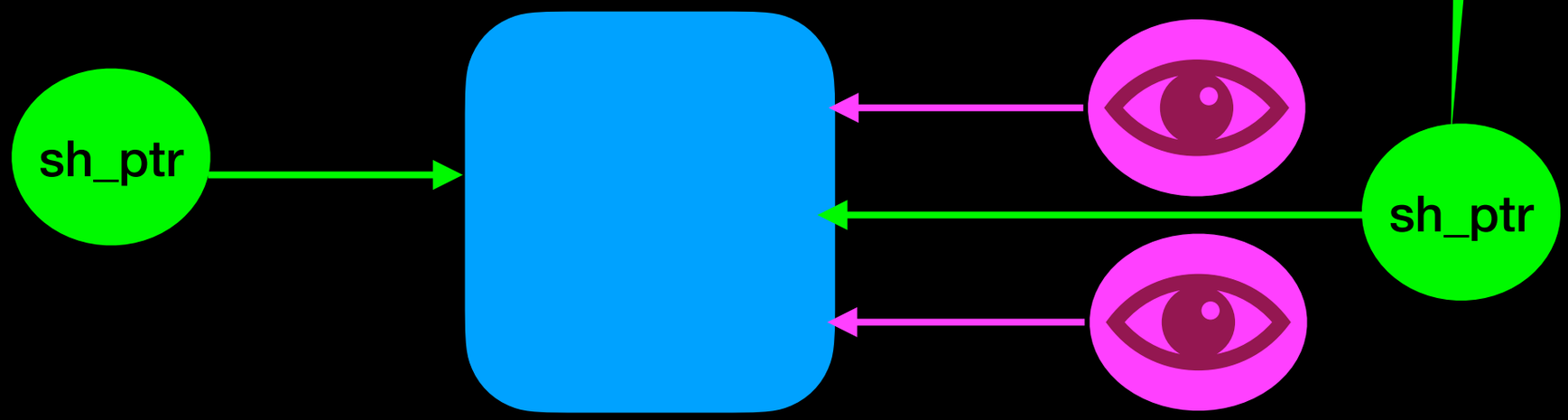
`weak_ptr`

```
//cannot directly access object from weak_ptr but can obtain a
//shared_ptr through a weak_ptr
std::shared_ptr<Song> another_shared_ptr =
weak_song_ptr1.lock();
another_shared_ptr->setTitle("my favorite song");

if(weak_song_ptr1.expired())
    //the object has been deleted
```

Obtained with `.lock()`

Returns true if object no longer exists, false otherwise



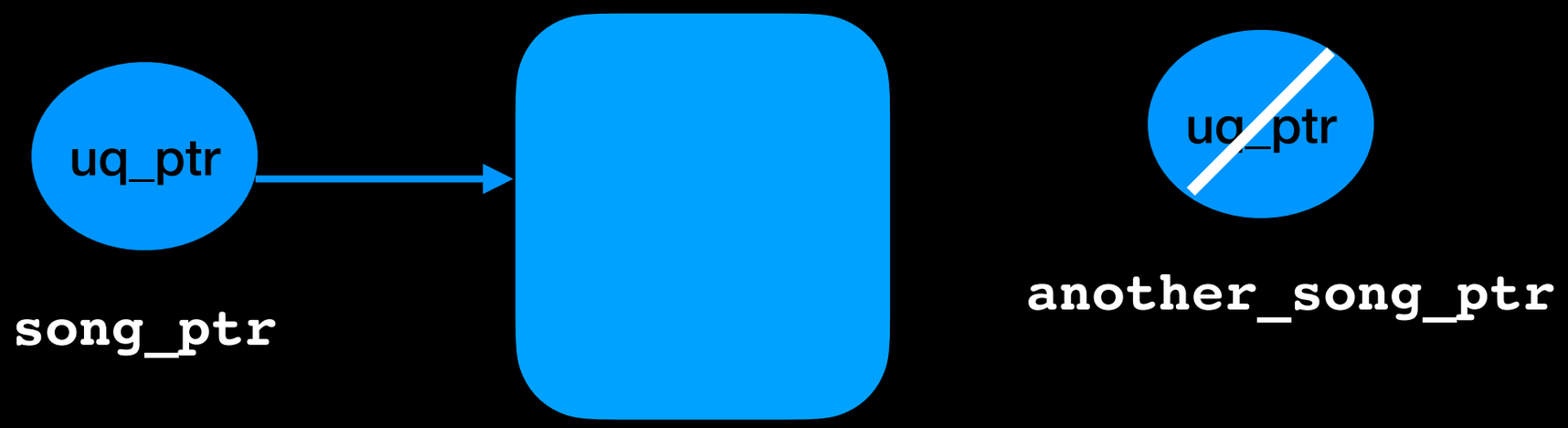
Smart/Managed Pointers

unique_ptr

```
auto song_ptr = std::make_unique<Song>();
std::unique_ptr<Song> another_song_ptr;
                                    //declaration only automatically set to nullptr

another_song_ptr = song_ptr;
                          //ERROR!!! copy assignment not permitted with unique_ptr
```

Error



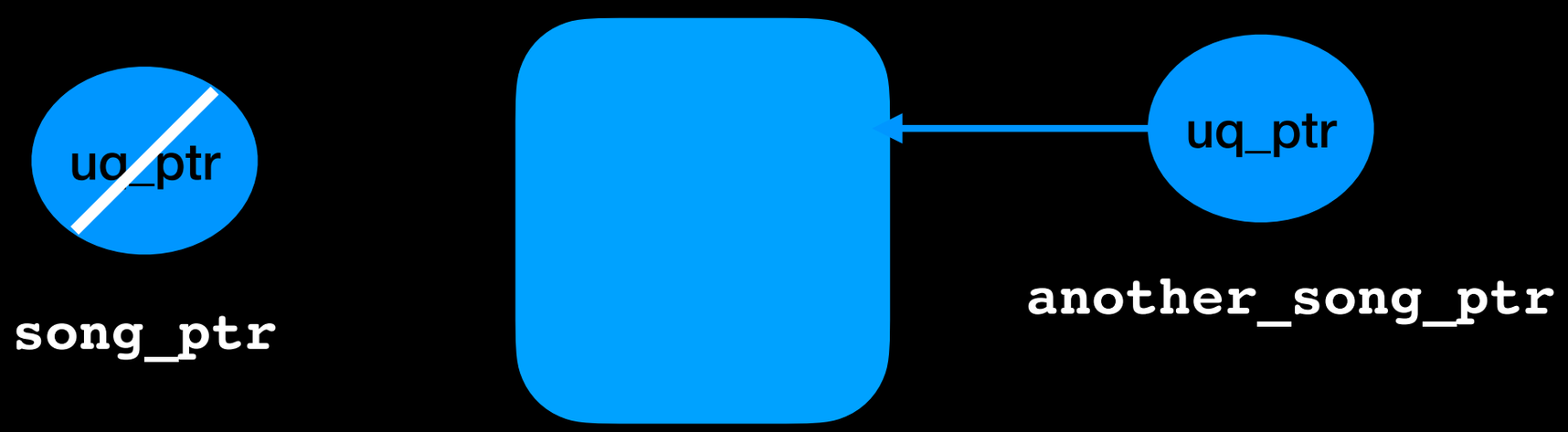
Smart/Managed Pointers

unique_ptr

Correct!

```
auto song_ptr = std::make_unique<Song>();
std::unique_ptr<Song> another_song_ptr;
                                    //declaration only automatically set to nullptr

another_song_ptr = std::move(song_ptr); //CORRECT! but song_ptr is now nullptr
```



In Essence

```
void useRawPointer()
{
   Song* song_ptr = new Song();
   song_ptr->setTitle(“My favorite song”);

   // do stuff

   // don’t forget to delete!!!
   delete song_ptr;
   song_ptr = nullptr;
}
```

Use it just like a raw pointer

It will take care of deleting the object automatically before its own destruction

```
void useSmartPointer()
{
    auto song_ptr = std::make_unique<Song>();
    song_ptr->setTitle("My favorite song");

    // do stuff

} // Song deleted automatically here
```

To summarize

Use smart pointers if you don’t have tight time/space constraints

Beware of cycles when using shared pointers