Exception Handling
(A light introduction)

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today’s Plan

Announcements

Recap

Exceptions (light)

Announcements

Graduate school:
Wisdom, experience, and the facts
Brought to you by Computer Science faculty

Bring your questions!

WHEN: April 17 from 1 to 2:30
WHERE 1022 HN

It is never too early to plan…

Implement Stack ADT

```
#ifndef STACK_H_
#define STACK_H_

template<class ItemType>
class Stack
{

public:
    Stack();
    void push(const ItemType& newEntry); // adds an element to top of stack
    void pop(); // removes element from top of stack
    ItemType top() const; // returns a copy of element at top of stack
    int size() const; // returns the number of elements in the stack
    bool isEmpty() const; // returns true if no elements on stack false otherwise

private:
    Node<ItemType>* top_; // Pointer to top of stack
    int itemCount;        // number of items currently on the stack


};    //end Stack

#include "Stack.cpp"
#endif // STACK_H_
```

Problem!

What happens if we call top() on empty stack???

```
T Stack<T>::top() const
{
    if(isEmpty())
       //what do we return???
    else
        return top_->getItem();

}
```

Further Considerations

What happens when preconditions are not met or input data is malformed?

- Do nothing
- Return false - `bool add(const ItemType& newEntry);`
- Use sentinel value: return error codes (e.g. negative numbers)

In general

What happens when preconditions are not met or input data is malformed?

- Do nothing

???

- Return false - `bool add(const ItemType& newEntry);`
- Use sentinel value: return error codes (e.g. negative numbers)

In general

What happens when preconditions are not met or input data is malformed?

- Do nothing
- Return false - `bool add(const ItemType& newEntry);`
- Use sentinel value: return error codes (e.g. negative numbers)

Rely on user to handle problem

Rely on user to handle problem

Sometimes it is not possible to return an error code
E.g. , complex objects or templates
No universal “uninitialized” value

In general

What happens when preconditions are not met or input data is malformed?

- Do nothing
- Return false - `bool add(const ItemType& newEntry);`
- Use sentinel value: return error codes (e.g. negative numbers)

What happens if we call `top()` on an empty stack?

assert

```
#include <cassert>

// ...
assert(!isEmpty());
```

Make sure this is true

If assertion is false, program execution terminates

assert

```
#include <cassert>

// ...
assert(!isEmpty());
```

Make sure this is true

So drastic! Give me another chance!

If assertion is false, program execution terminates

Good for testing and debugging

Exceptions:
A Light Introduction

Exceptions

Client might be able to recover from a violation or unexpected condition

Communicate Exception (error) to client:

- Bypass normal execution
- Return control to client
- Communicate error

Exceptions

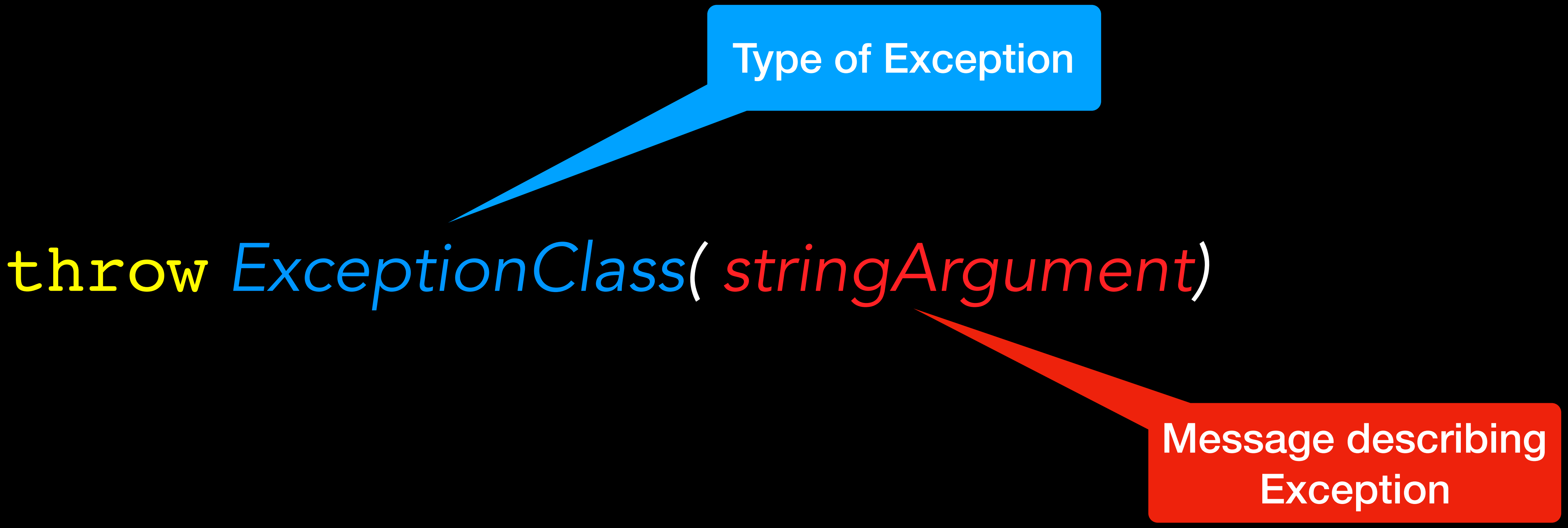
Client might be able to recover from a violation or unexpected condition

Communicate Exception (error) to client:

- Bypass normal execution
- Return control to client
- Communicate error

Throw and Exception

Throwing Exceptions



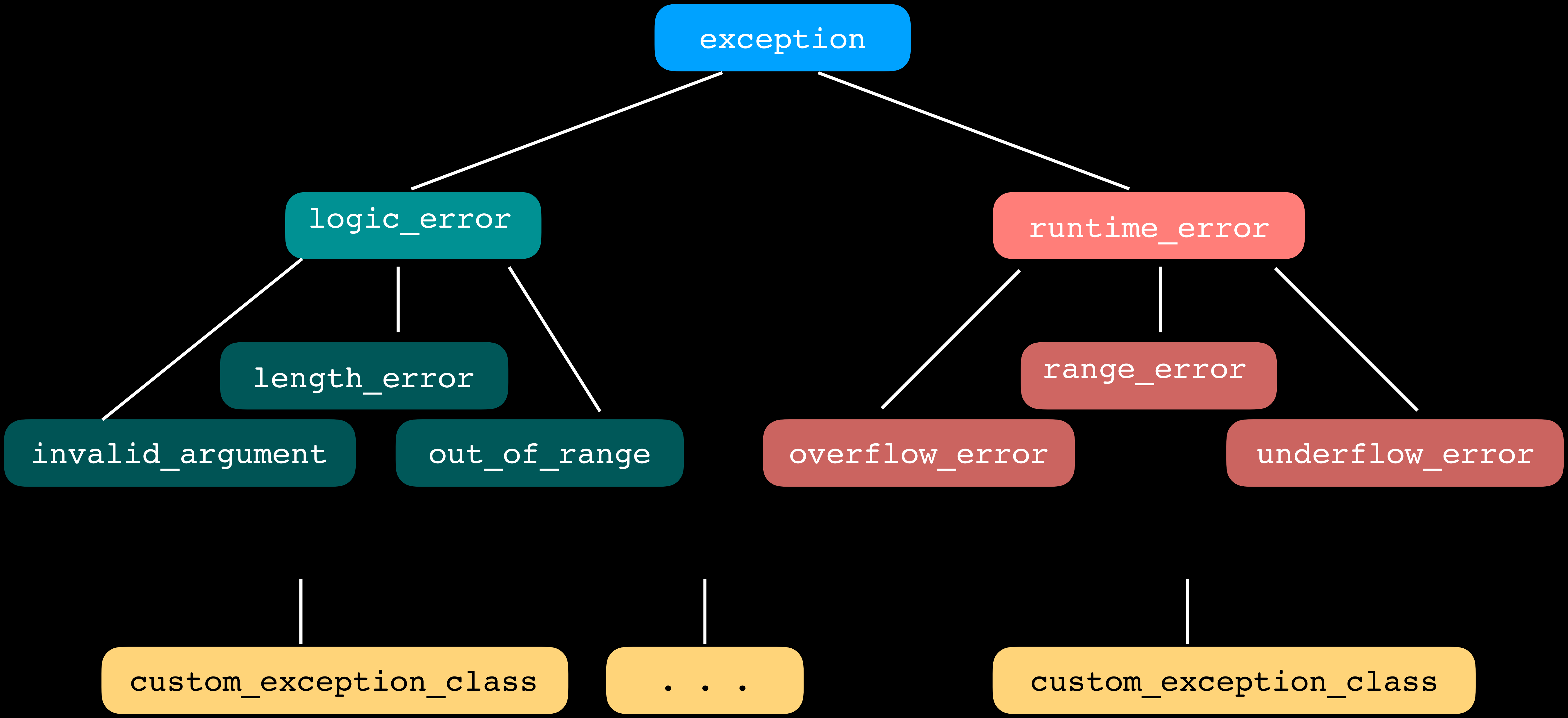
Type of Exception

throw *ExceptionClass*(*stringArgument*)

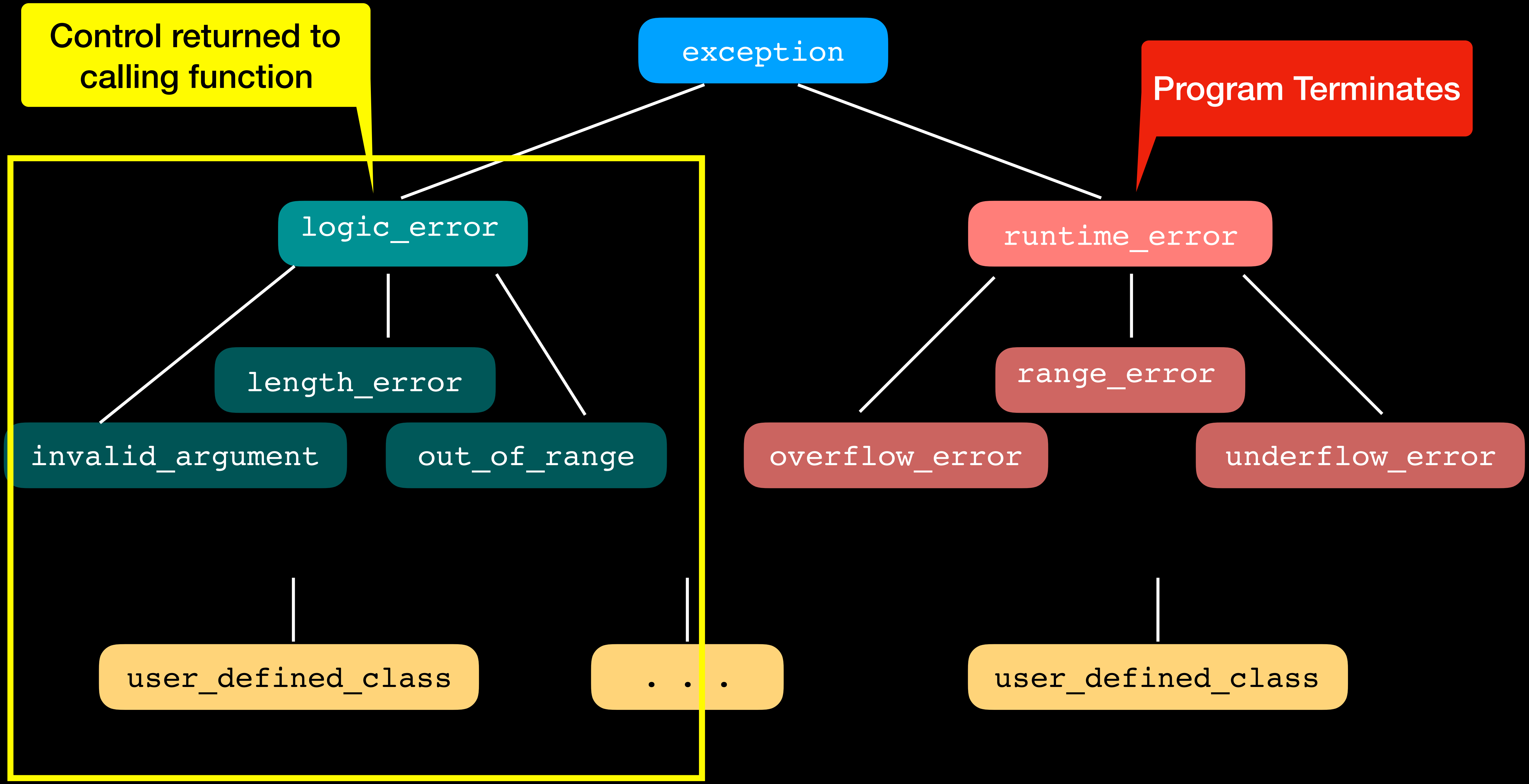
Message describing Exception

```
ItemType Stack<ItemType>::top() const
{

    if(isEmpty())
       throw std::out_of_range(“Attempt to access empty Stack");

    //code here

}
```

C++ Exception Classes



C++ Exception Classes



Control returned to calling function

exception

Program Terminates

logic_error

runtime_error

length_error

invalid_argument

out_of_range

range_error

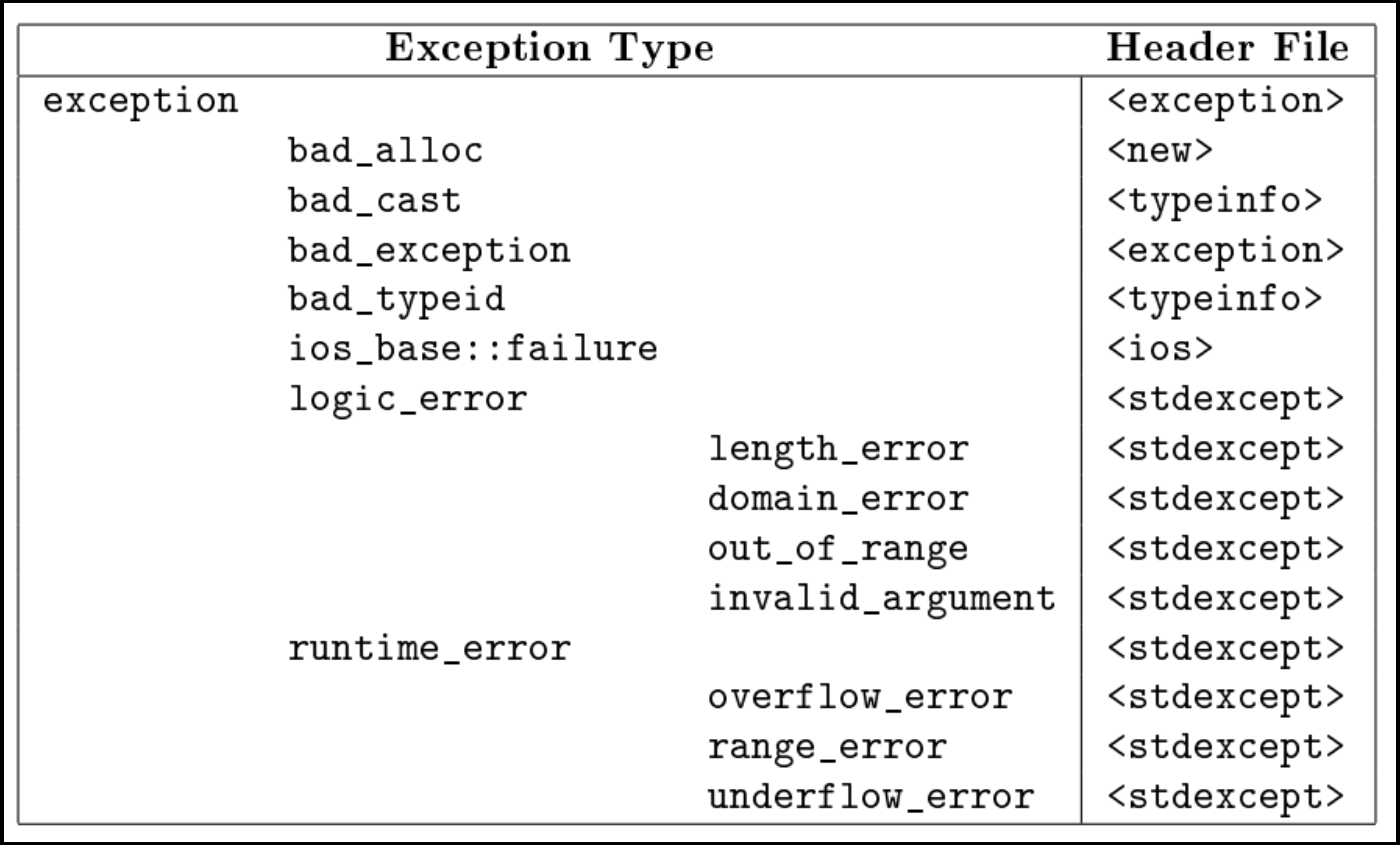
overflow_error

underflow_error

user_defined_class

. . .

user_defined_class



Exception Type			Header File
exception			<exception>
	bad_alloc		<new>
	bad_cast		<typeinfo>
	bad_exception		<exception>
	bad_typeid		<typeinfo>
	ios_base::failure		<ios>
	logic_error		<stdexcept>
		length_error	<stdexcept>
		domain_error	<stdexcept>
		out_of_range	<stdexcept>
		invalid_argument	<stdexcept>
	runtime_error		<stdexcept>
		overflow_error	<stdexcept>
		range_error	<stdexcept>
		underflow_error	<stdexcept>

Exception Handling

Can handle only exceptions of class `logic_error` and its derived classes

Exception Handling

```
try
{
    //statement(s) that might throw exception
}
catch(ExceptionClass1 identifier)
{
    //statement(s) that react to an exception
   // of type ExceptionClass1
}
catch(ExceptionClass2 identifier)
{
    //statement(s) that react to an exception
   // of type ExceptionClass2
}
. . .
```

Exception Handling

Arrange catch blocks in order of specificity,
catching most specific first
(i.e. lower in the Exception Class Hierarchy first)

```
try
{
    //statement(s) that might throw exception
}
catch(const ExceptionClass1& identifier)
{
    //statement(s) that react to an exception
   // of type ExceptionClass1
}
catch(const ExceptionClass2& identifier)
{
    //statement(s) that react to an exception
   // of type ExceptionClass2
}
. . .
```

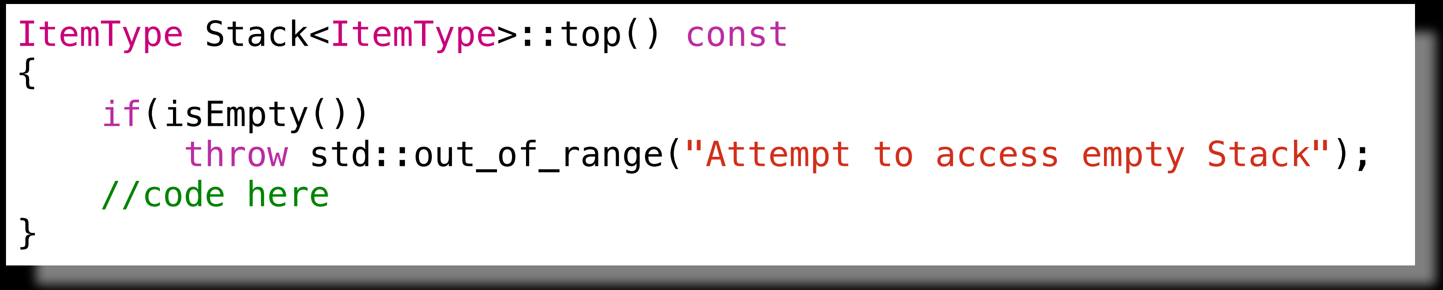
Good practice to catch exceptions by `const reference` whenever possible
(due to memory management, avoiding copying and slicing issues)

Exception Handling

You know `top()` may throw an exception so call it in a `try` block

```
try
{
    some_object = my_stack.top();
}
catch(const std::out_of_range& problem)
{
    some_object = valid_initial_value;
}
```

Exception Handling



```
ItemType Stack<ItemType>::top() const
{
    if(isEmpty())
        throw std::out_of_range("Attempt to access empty Stack");
    //code here
}
```

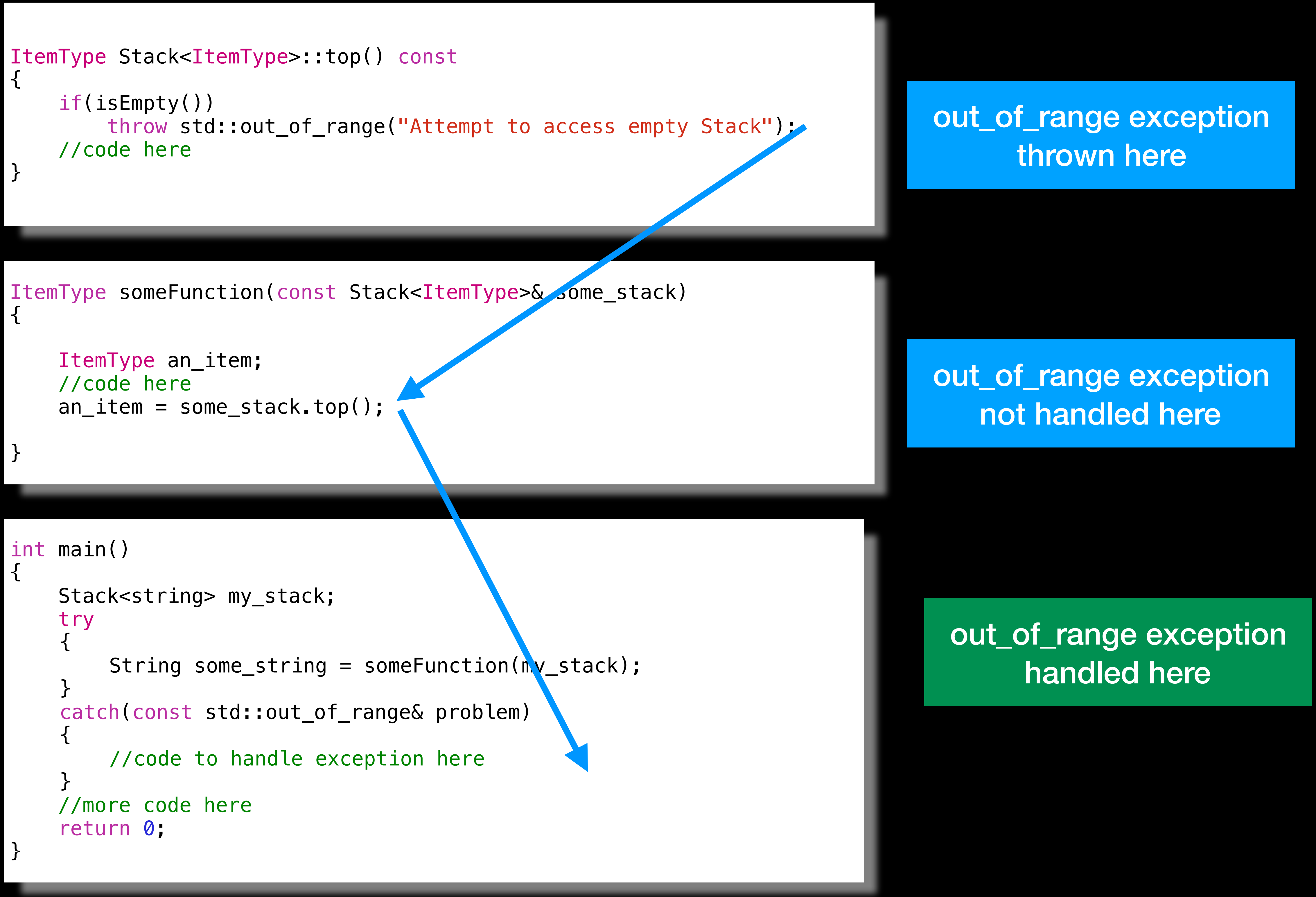
Returns string parameter to thrown exception

```
try
{
    some_object = my_stack.top();
}
catch(const std::out_of_range& problem)
{
    std::cerr << problem.what() << std::endl;
    some_object = valid_initial_value;
}
```

Error output stream:

Attempt to access empty Stack

Uncaught Exceptions



```
ItemType Stack<ItemType>::top() const
{
    if(isEmpty())
        throw std::out_of_range("Attempt to access empty Stack");
    //code here
}
```

out_of_range exception thrown here

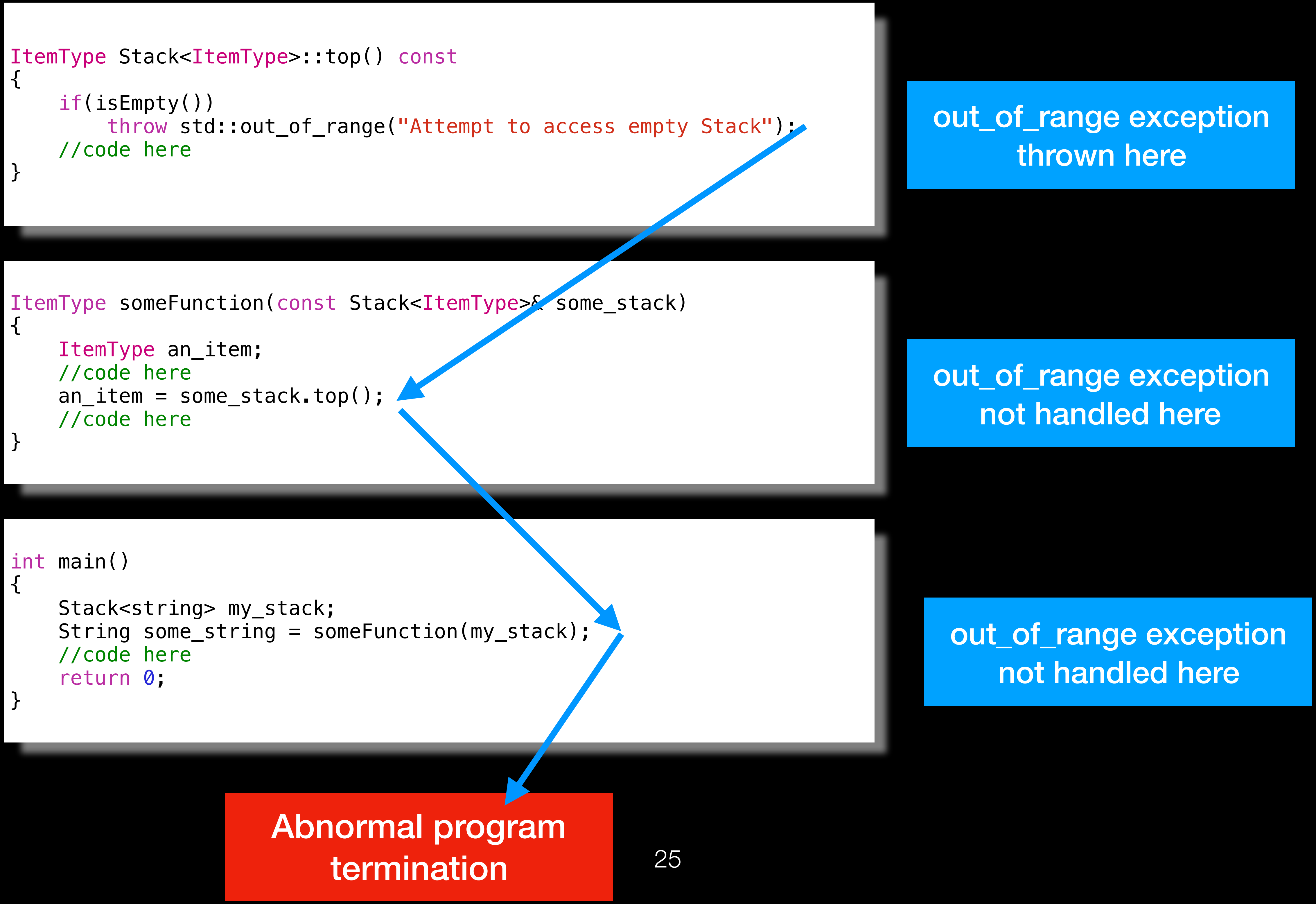
```
ItemType someFunction(const Stack<ItemType>& some_stack)
{

    ItemType an_item;
    //code here
    an_item = some_stack.top();

}
```

out_of_range exception not handled here

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

out_of_range exception handled here

Unhandled Exceptions



```
ItemType Stack<ItemType>::top() const
{
    if(isEmpty())
        throw std::out_of_range("Attempt to access empty Stack");
    //code here
}
```

out_of_range exception thrown here

```
ItemType someFunction(const Stack<ItemType>& some_stack)
{
    ItemType an_item;
    //code here
    an_item = some_stack.top();
    //code here
}
```

out_of_range exception not handled here

```
int main()
{
    Stack<string> my_stack;
    String some_string = someFunction(my_stack);
    //code here
    return 0;
}
```

out_of_range exception not handled here

Abnormal program termination

Implications

There could be several
. . . out of the scope of this course

We will discuss one:

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

Implications and Complications

There could be many
. . . out of the scope of this course

Memory leak!!!

We will discuss one:

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

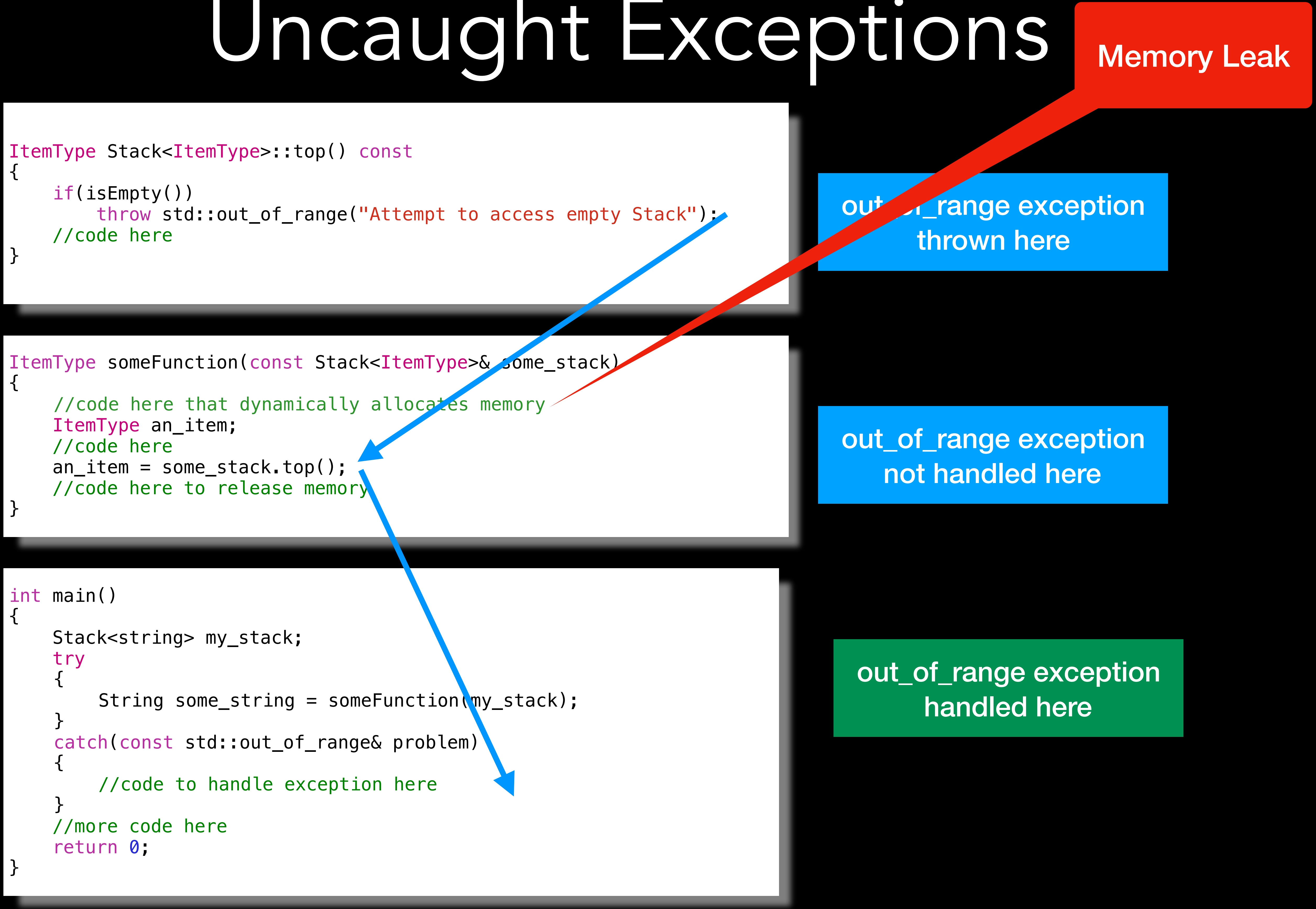
😱

Dynamically allocated memory never released!!!

Implications and Complications

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

Uncaught Exceptions



```
ItemType Stack<ItemType>::top() const
{
    if(isEmpty())
        throw std::out_of_range("Attempt to access empty Stack");
    //code here
}
```

```
ItemType someFunction(const Stack<ItemType>& some_stack)
{
    //code here that dynamically allocates memory
    ItemType an_item;
    //code here
    an_item = some_stack.top();
    //code here to release memory
}
```

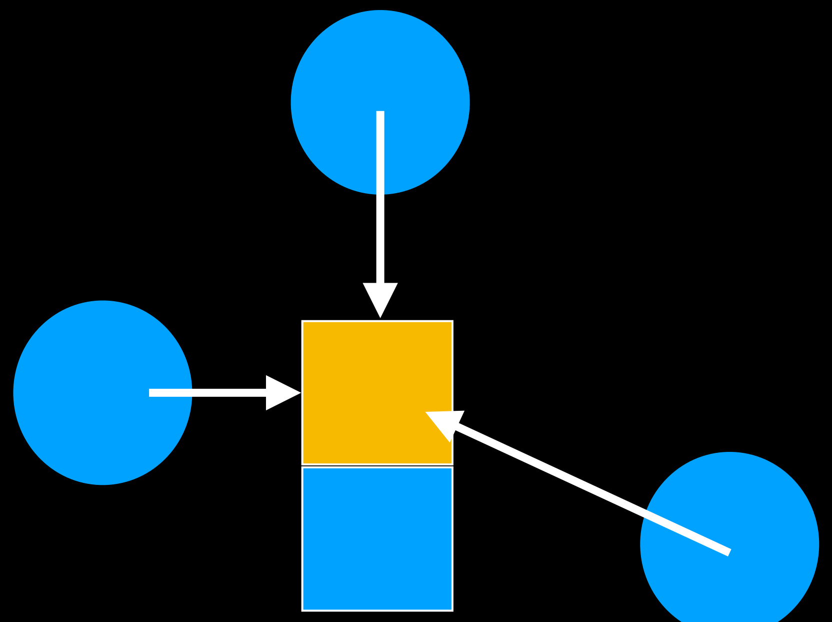
```
int main()
{
    Stack<string> my_stack;
    try
    {
        String some_string = someFunction(my_stack);
    }
    catch(const std::out_of_range& problem)
    {
        //code to handle exception here
    }
    //more code here
    return 0;
}
```

Memory Leak

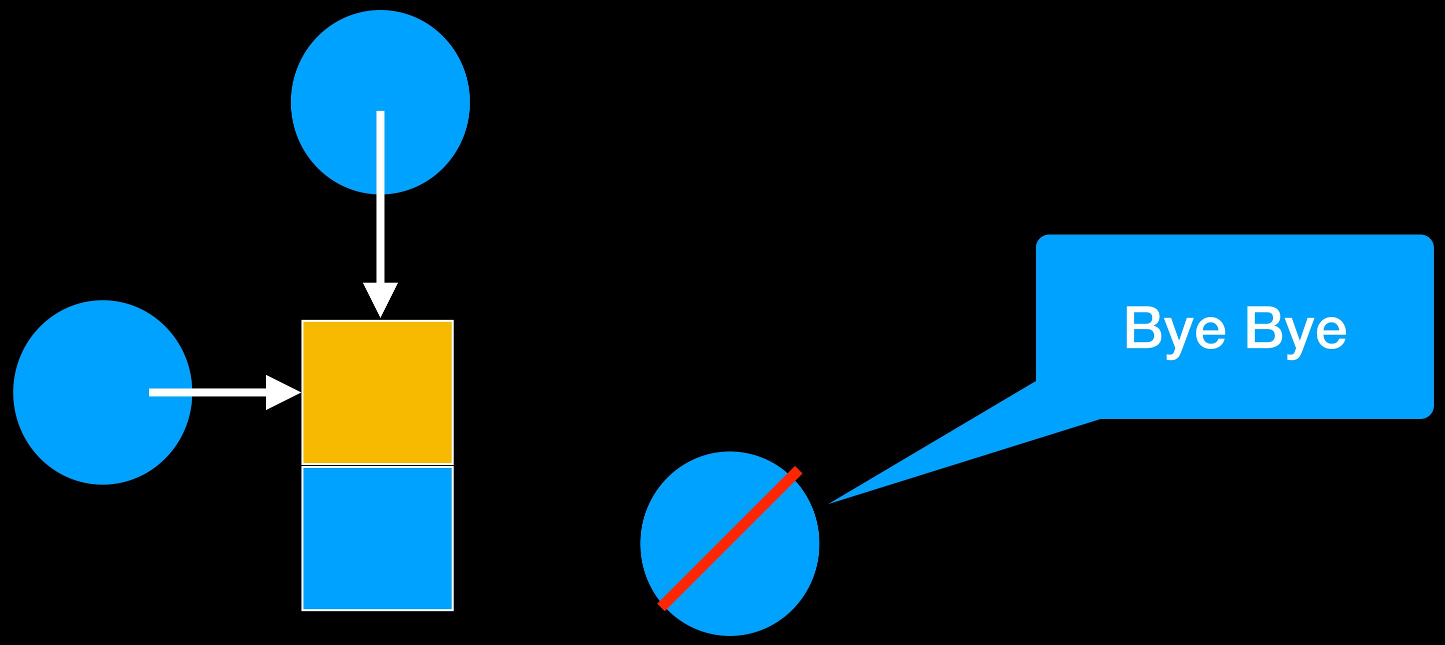
out_of_range exception thrown here

out_of_range exception not handled here

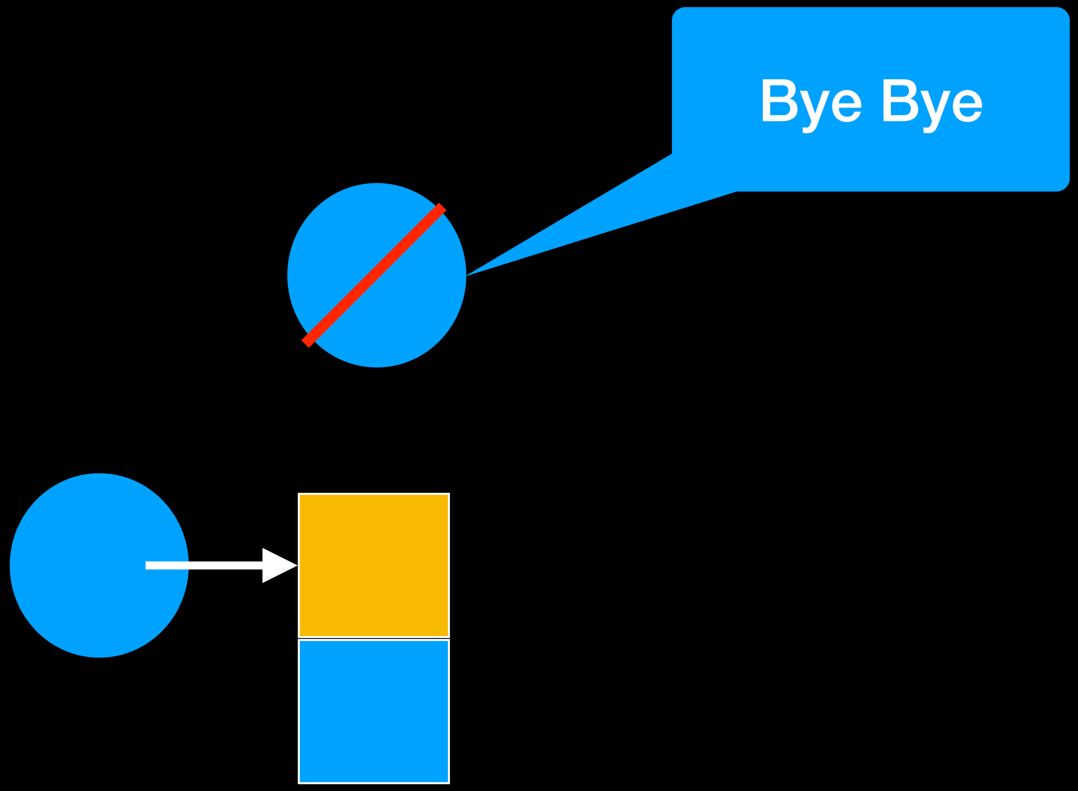
out_of_range exception handled here

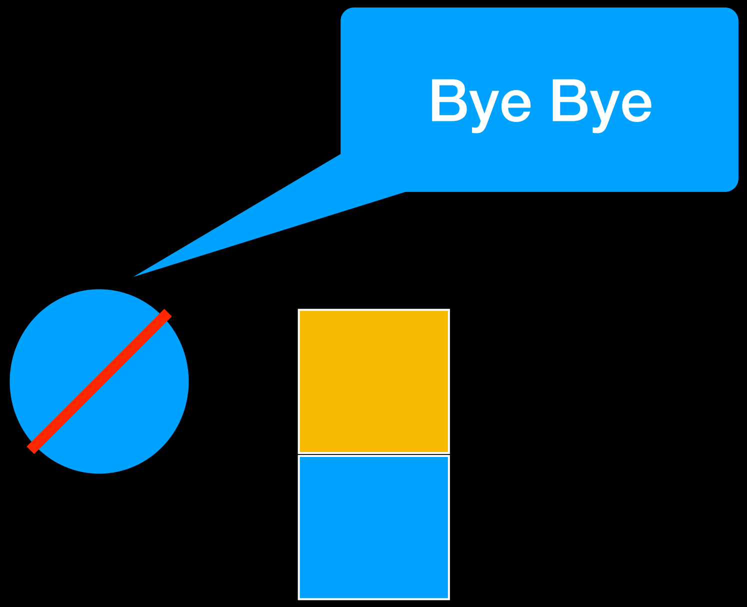


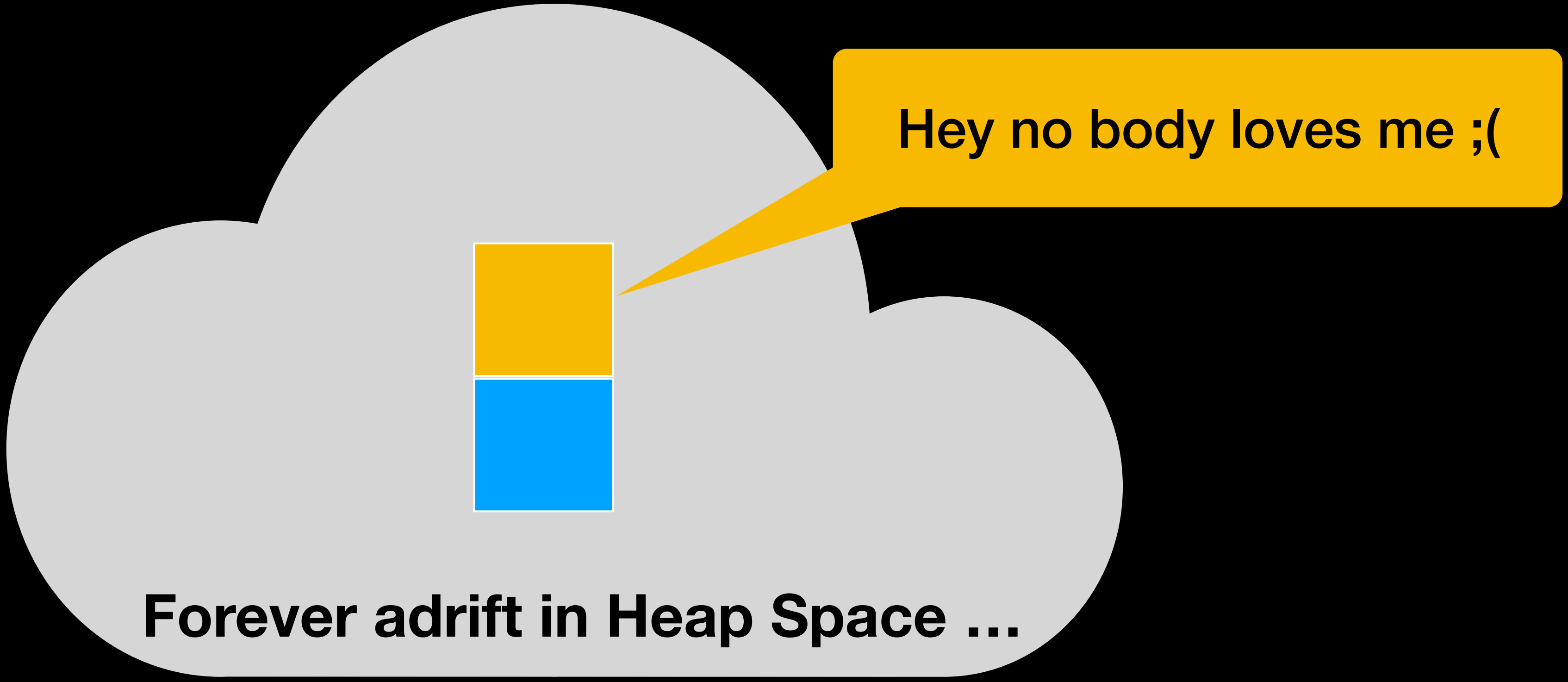
Pointers are not aware of each other



Bye Bye

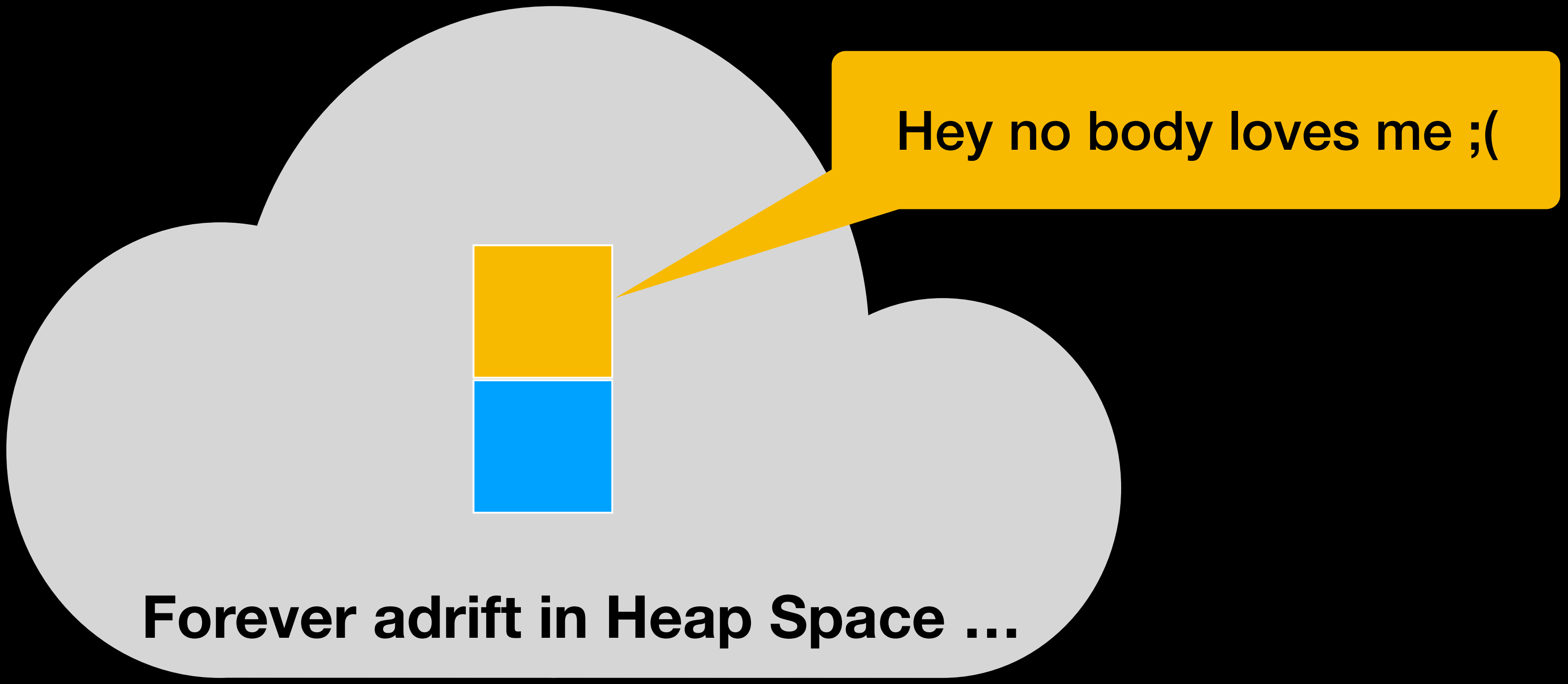

Bye Bye


Bye Bye



Hey no body loves me ;(

Forever adrift in Heap Space …



Hey no body loves me ;(

Forever adrift in Heap Space …

Programmer responsible for keeping track

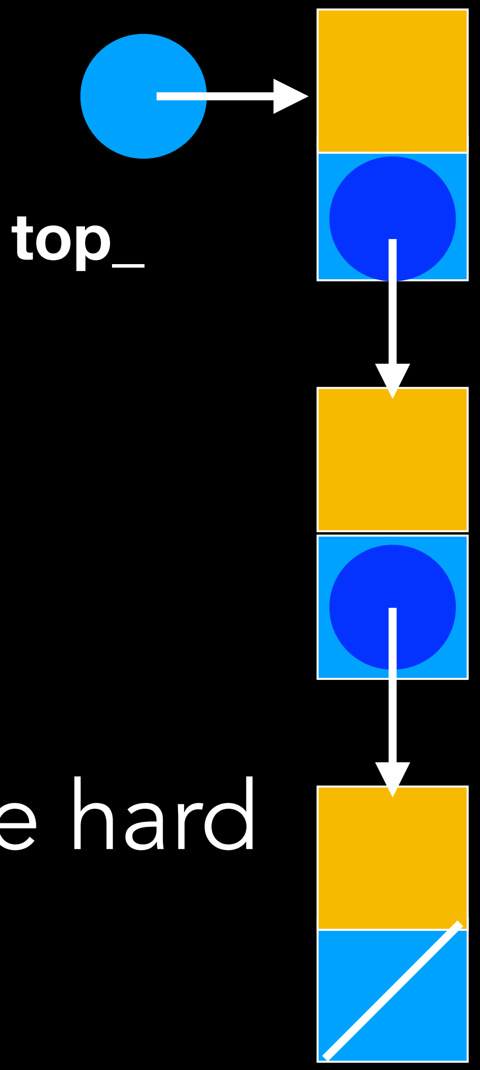
Ownership

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

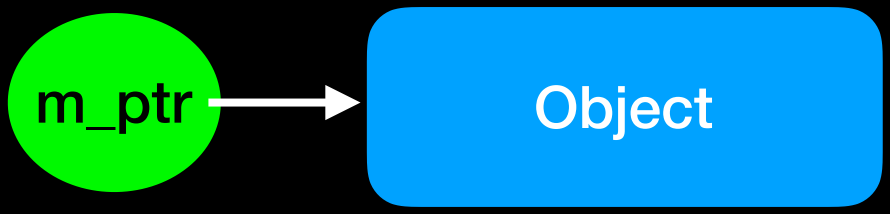
C++14

Distinguish from”smart”

Smart pointer:

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

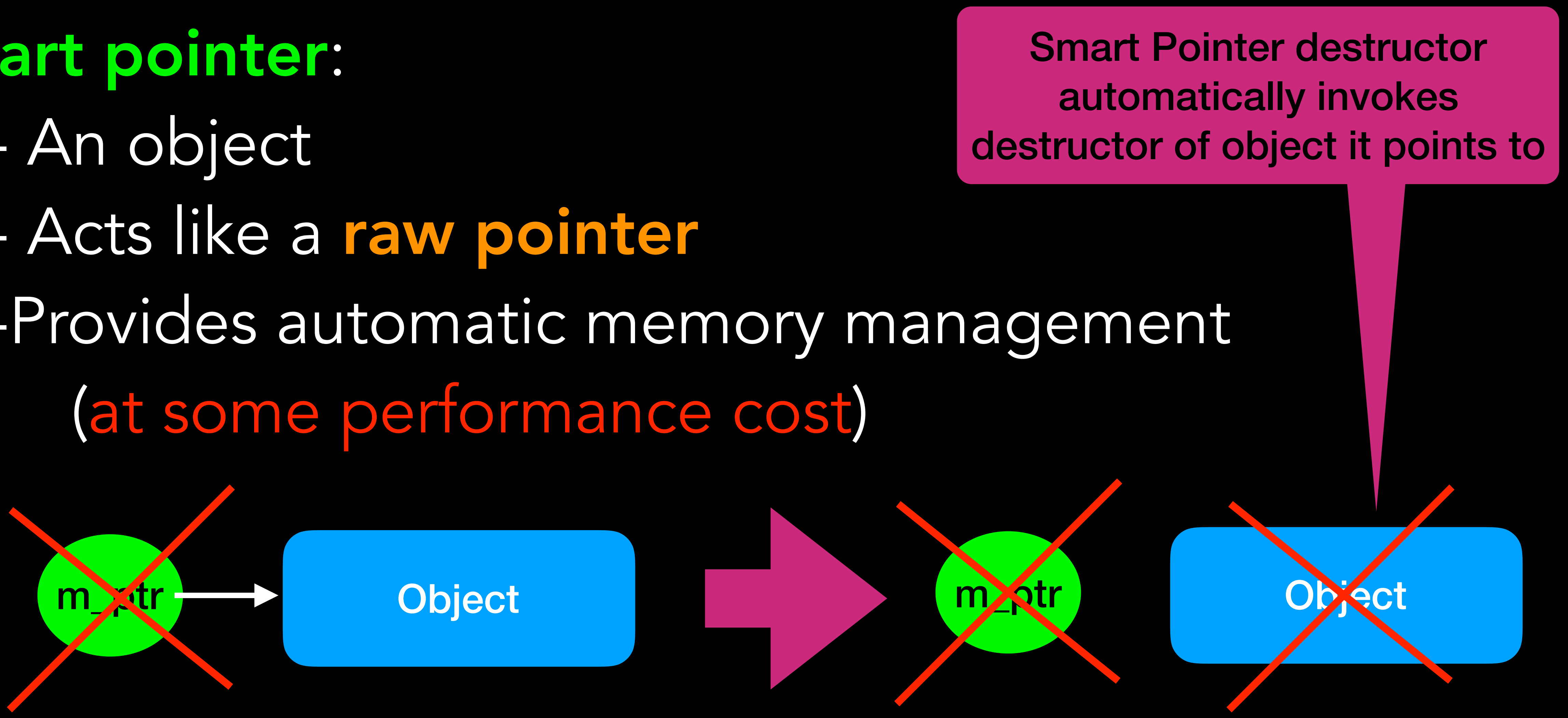
A non-trivial sentence but we will leave it at that



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
m_ptr
Object
m_ptr
Object

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

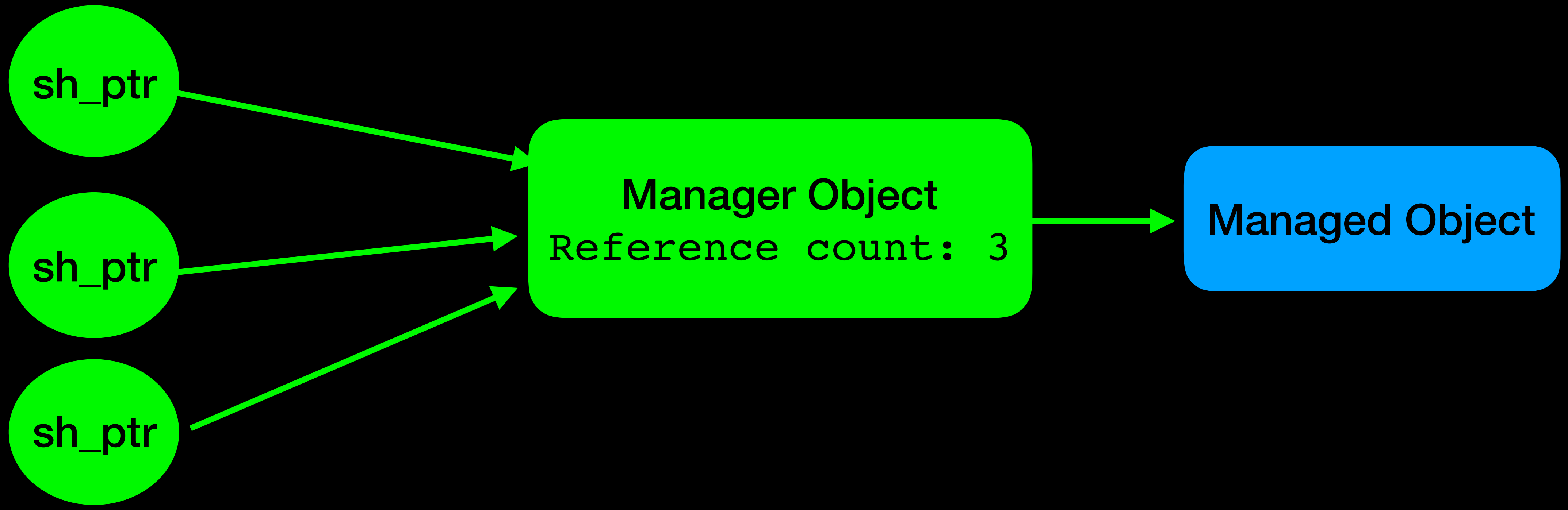
Unique ownership: only smart pointer allowed to point to the object

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

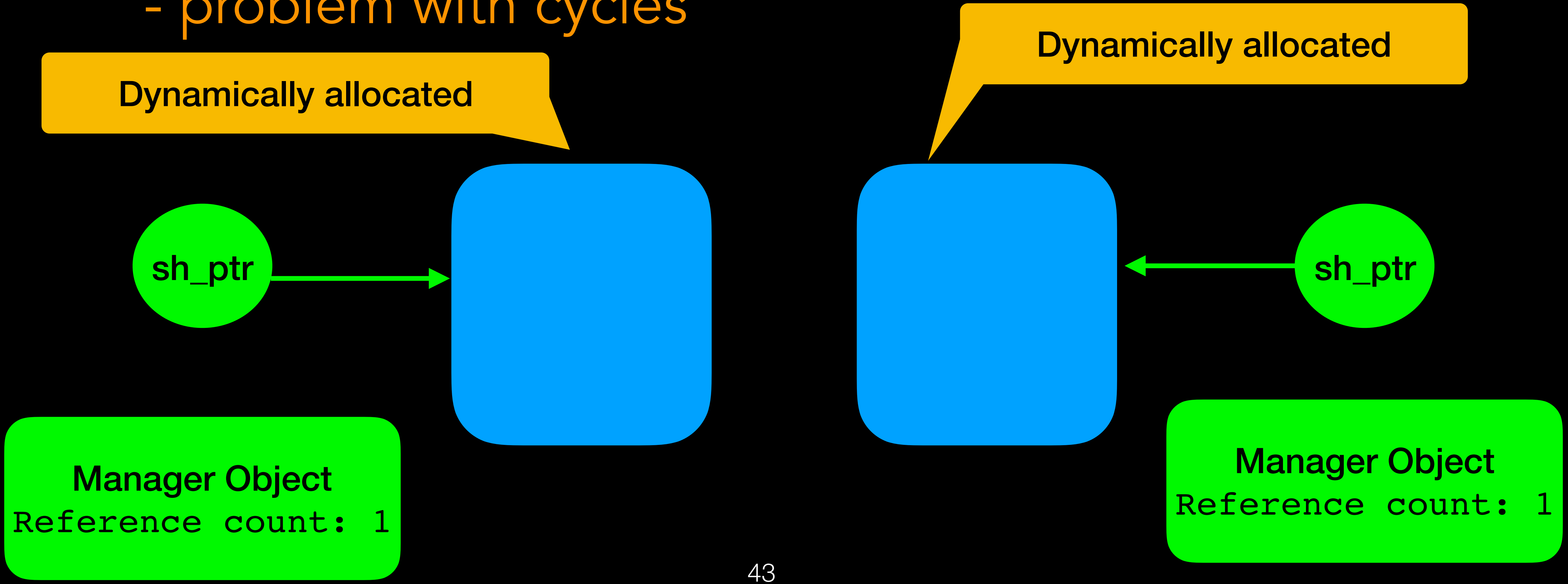
`shared_ptr`

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

Smart/Managed Pointers

`shared_ptr`

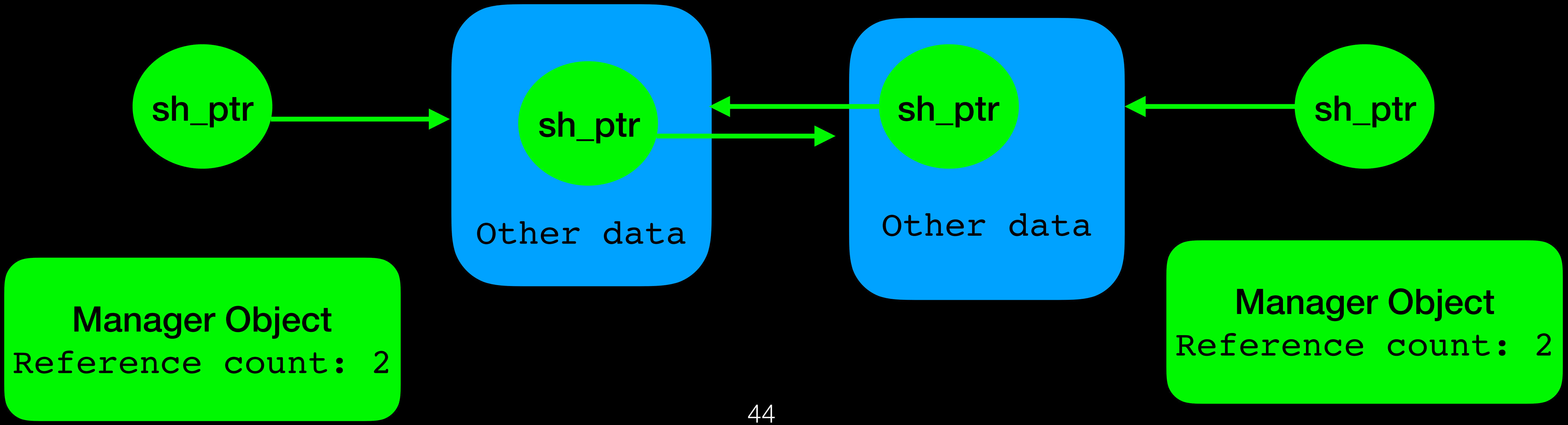
- keep count how many references to same object
- last pointer responsible for deleting object
- problem with cycles



Smart/Managed Pointers

`shared_ptr`

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

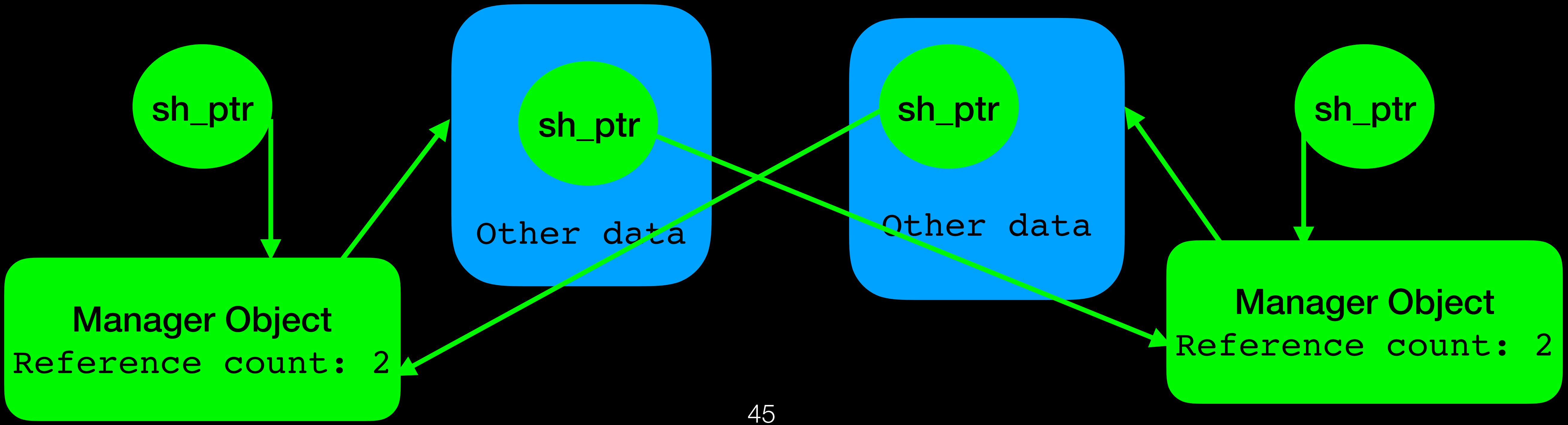


Smart/Managed Pointers

`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

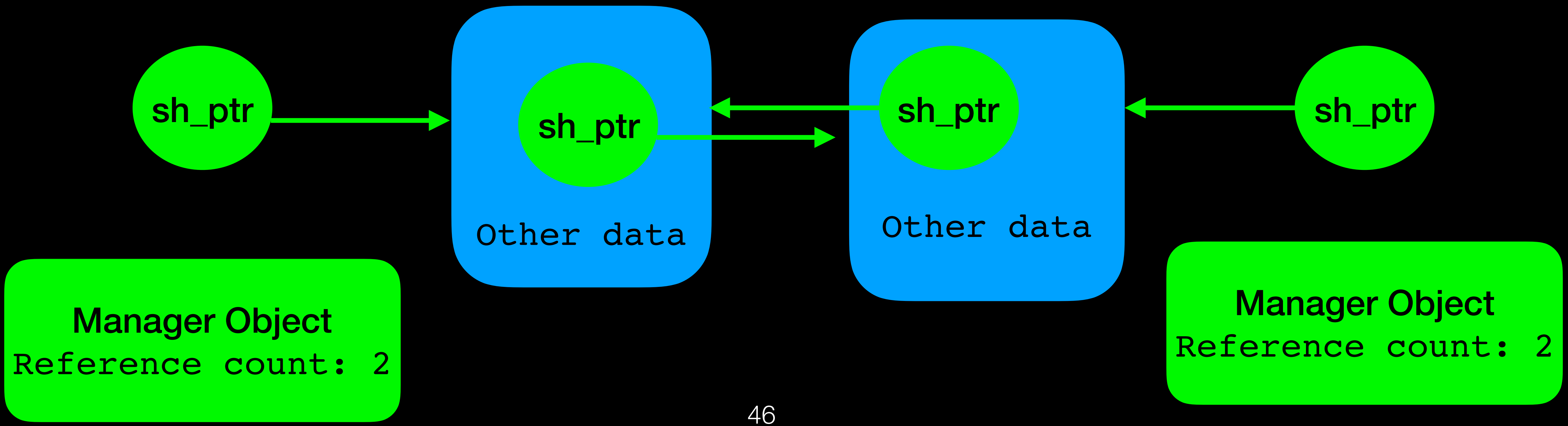


Smart/Managed Pointers

`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

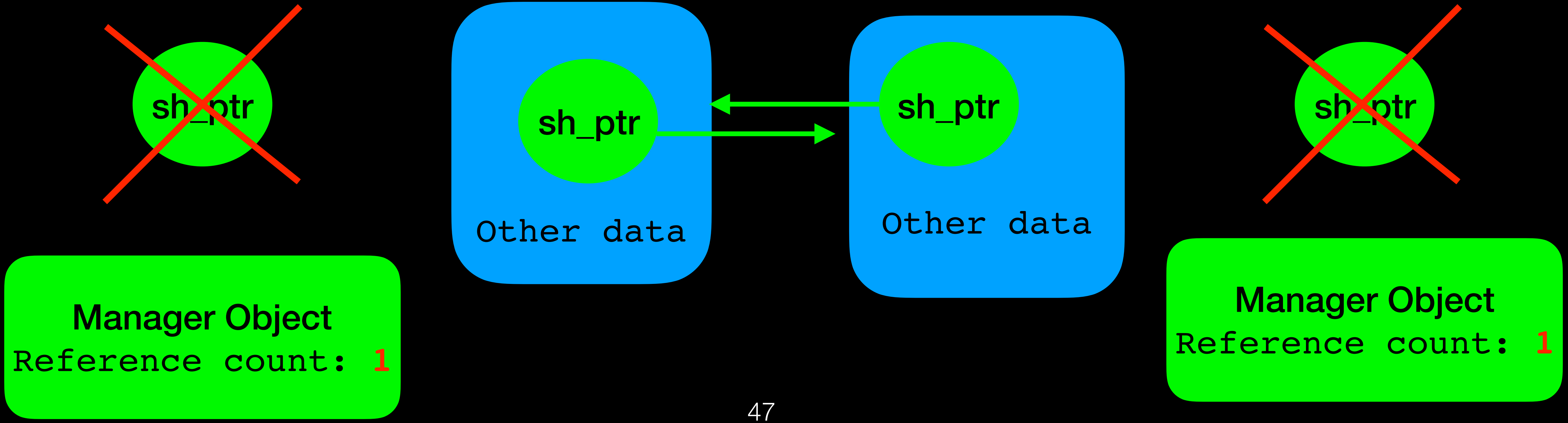


Smart/Managed Pointers

`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



sh_ptr

sh_ptr

Other data

sh_ptr

Other data

sh_ptr

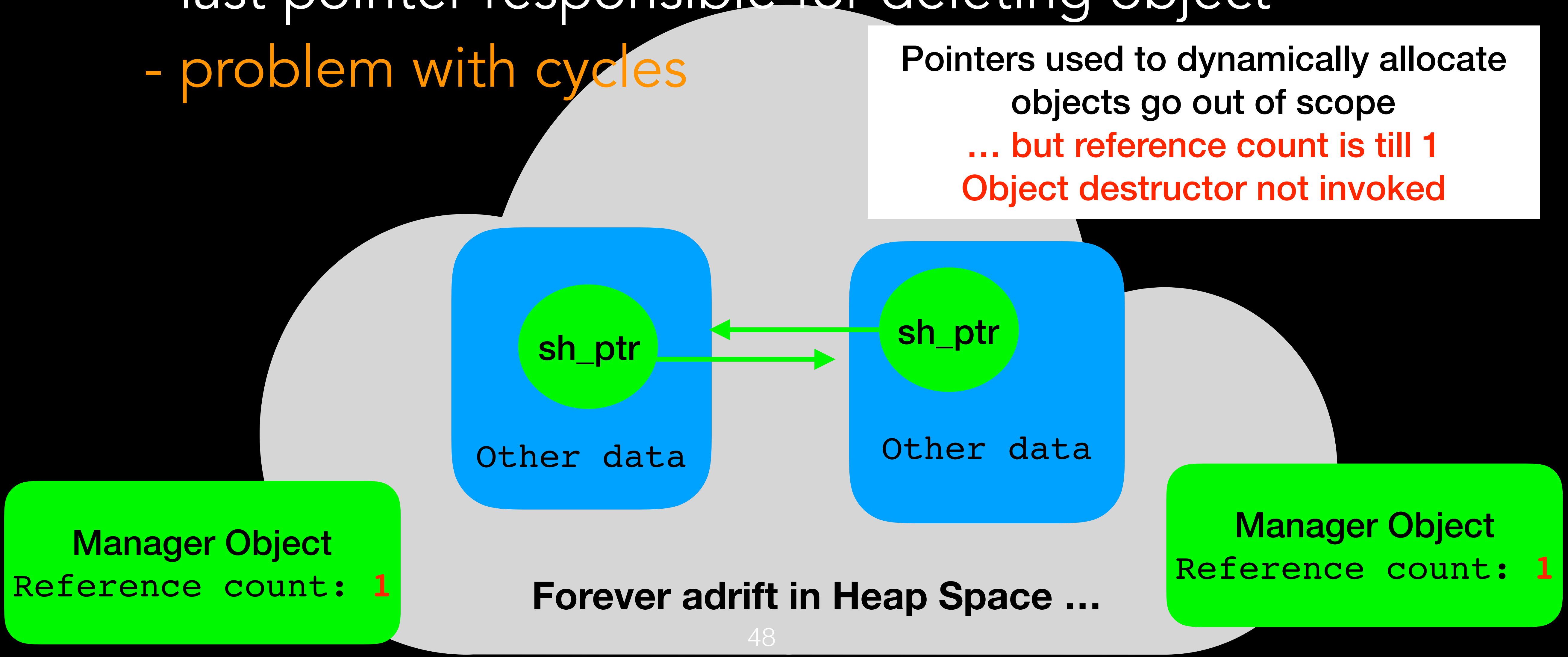
Manager Object
Reference count: 1

Manager Object
Reference count: 1

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

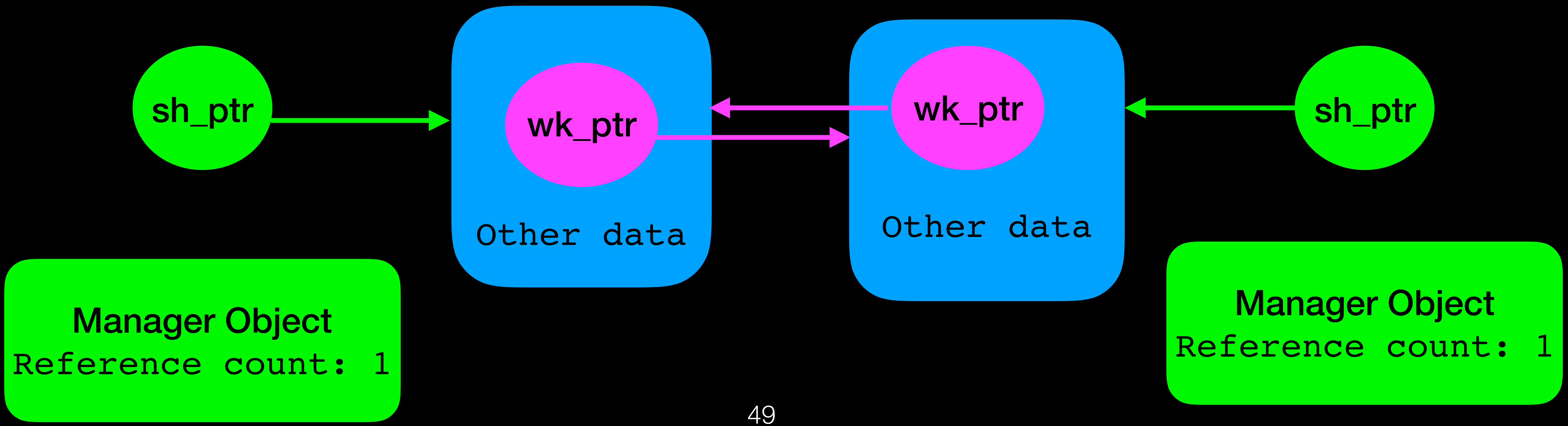
Forever adrift in Heap Space ...

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

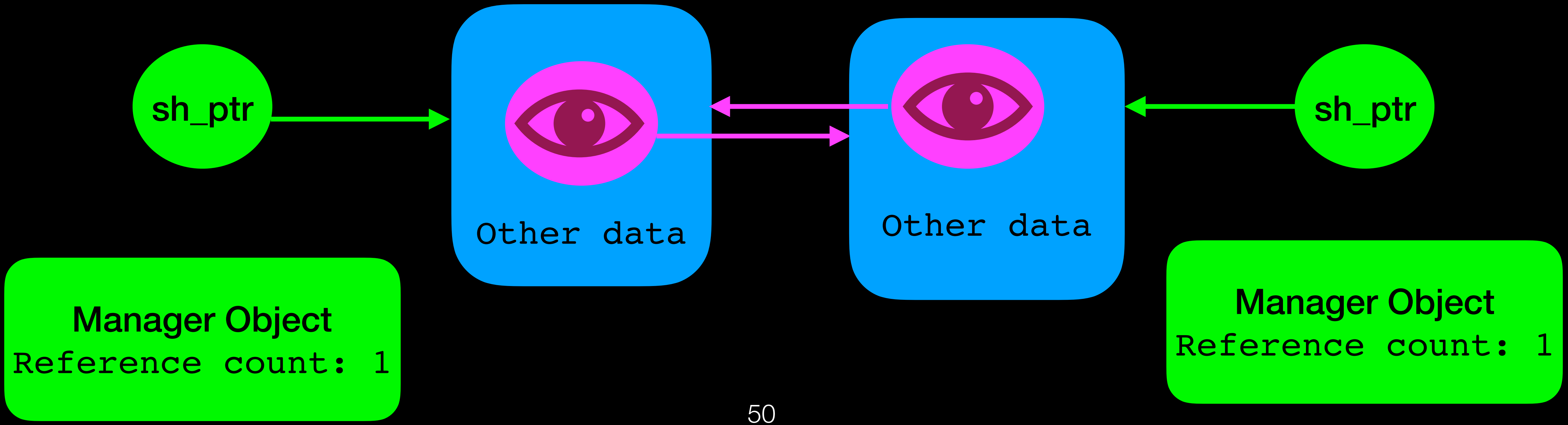


Smart/Managed Pointers

`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

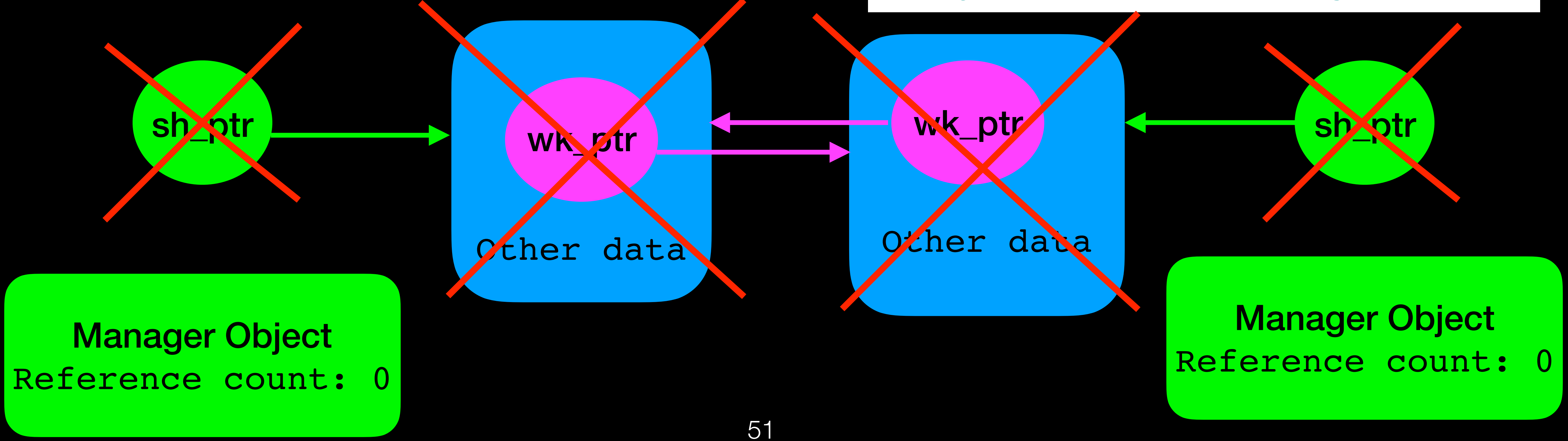


Smart/Managed Pointers

`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 pointer = new Object()
                    //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 =

More efficient
Do it this way

shared_ptr

```
std::shared_ptr<Song> song_ptr1; //declaration only

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

    // do stuff

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

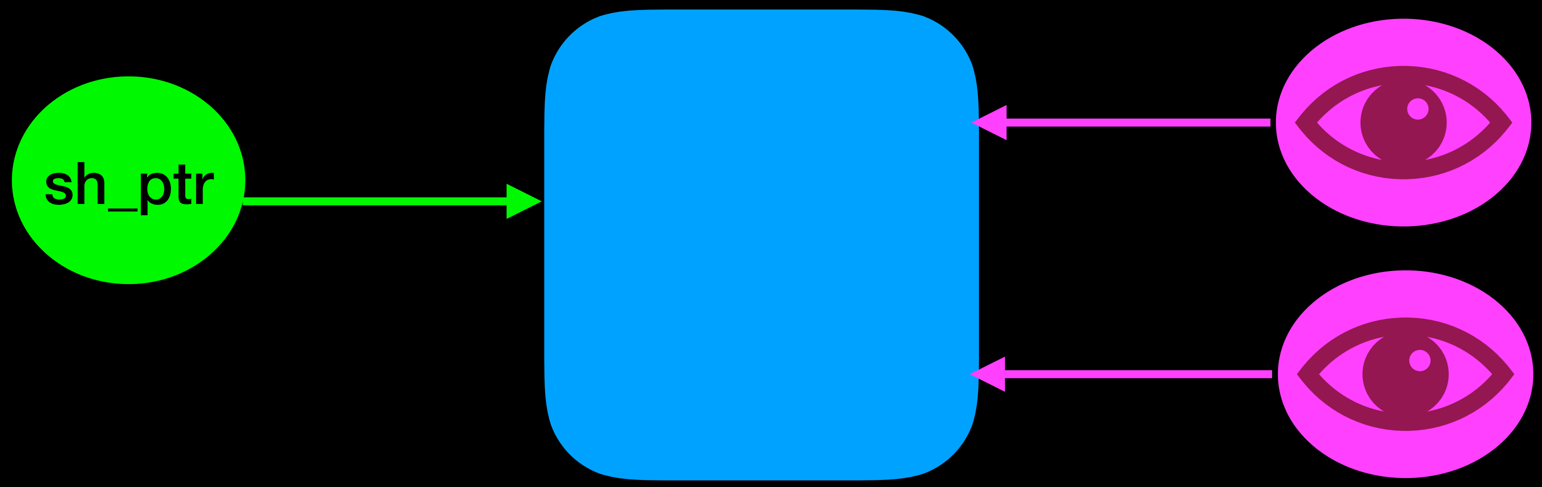
Use it just like you would a raw pointer

Syntax

weak_ptr cannot own object, so cannot be used to allocate a new object — must allocate new object through weak 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;
```



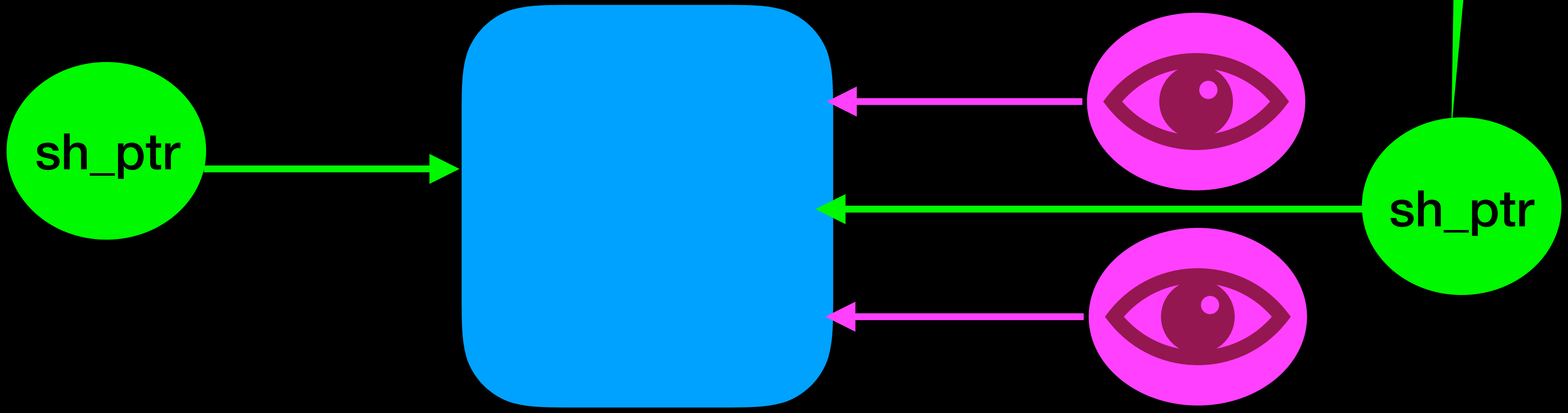
Syntax

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 still exists, false otherwise

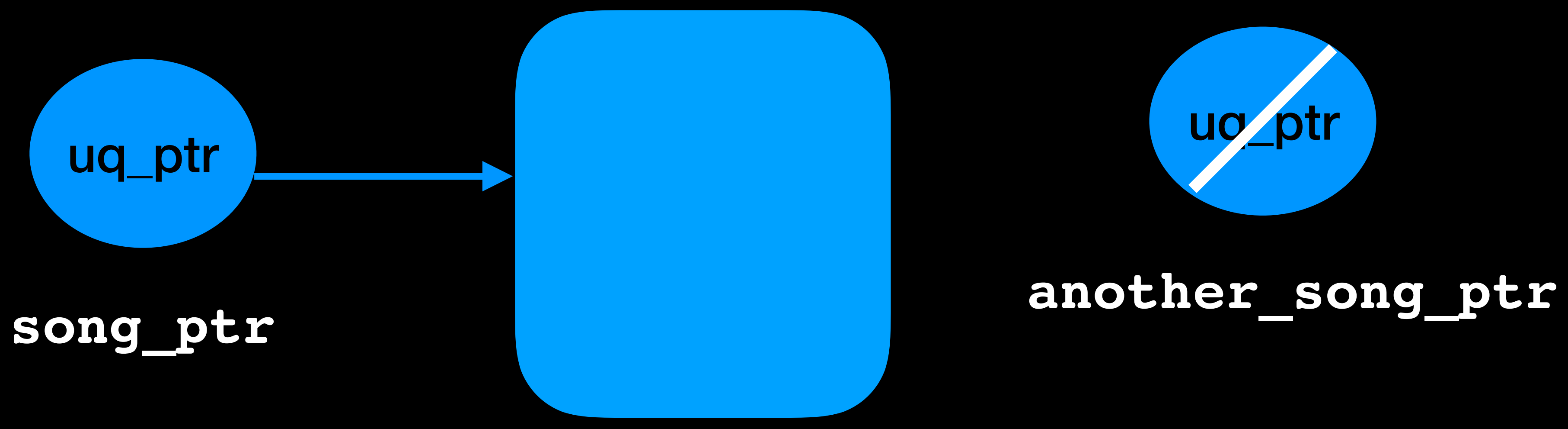


Smart/Managed Pointers

unique_ptr

Error

```
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
```

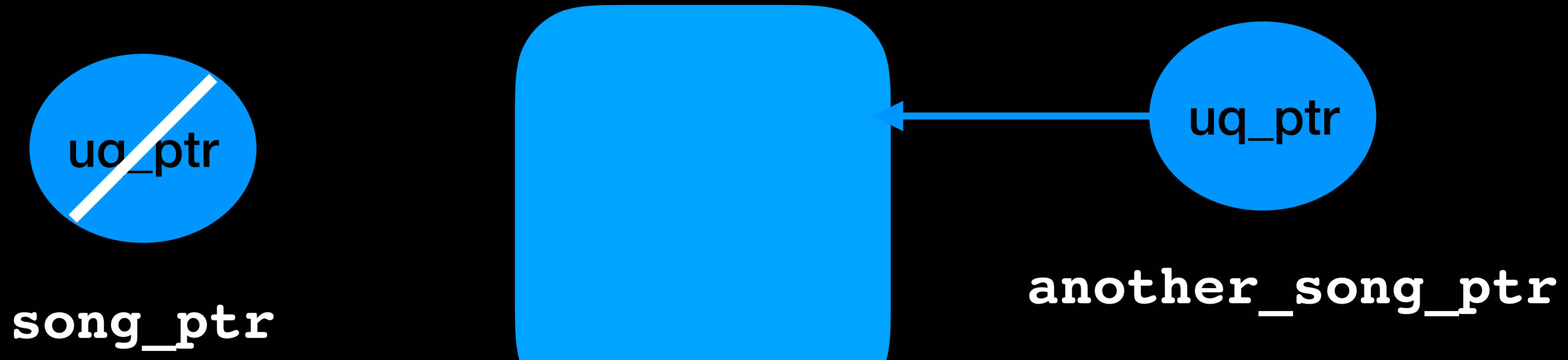


Smart/Managed Pointers

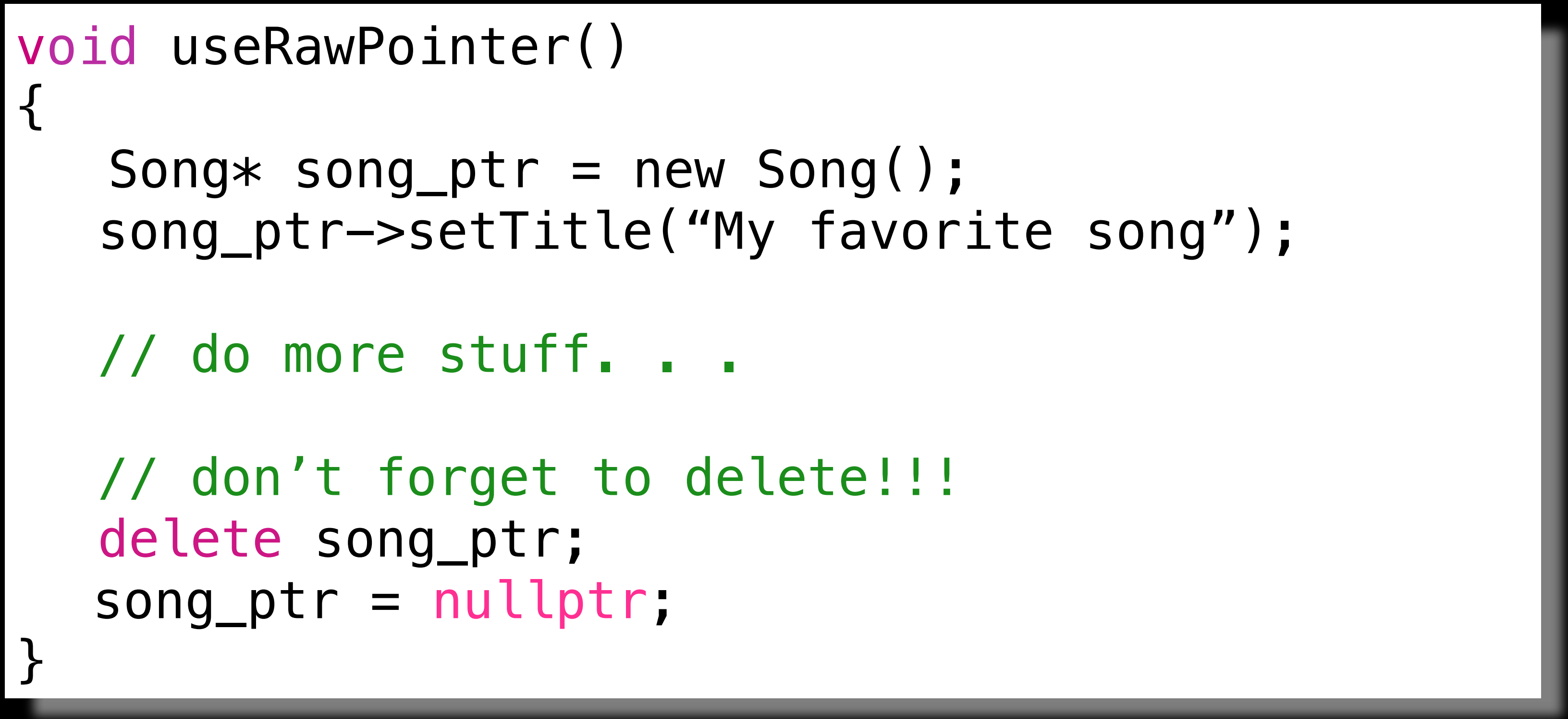
unique_ptr

Correct!

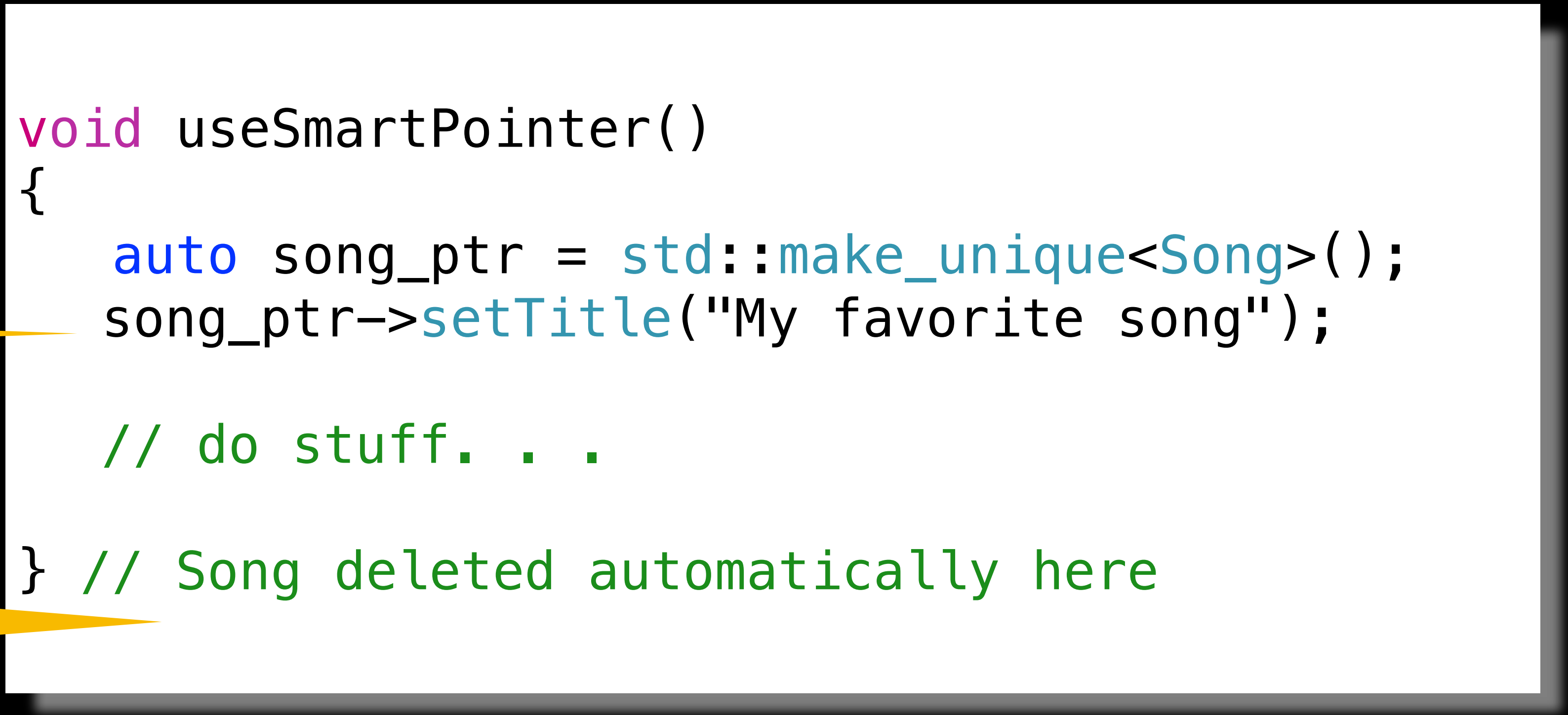
```
auto song_ptr = std::make_unique<Song>();
std::unique_ptr<Song> another_song_ptr;
                                //declaration only automatically set to nullptr

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 more stuff. . .

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



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

   // do stuff. . .

} // Song deleted automatically here
```

Use it just like a raw pointer

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

To summarize

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

Beware of cycles when using shared pointers