Pointers Review

Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan

A quick review of pointers and dynamic memory allocation

Next time: Linked-Based Implementation

Constructors Clarifications

- Multiple constructors, only one is invoked

main()

```
#include “Animal.hpp"

int main()
{

    Animal nameless; //calls default constructor

    Animal tiger(“tiger”); //calls parameterized const. w/ default args

    Animal shark(“shark”, false, true); //calls parameterized constructor
                                        //with all arguments
    //more code here . . .

}; //end main
```

```
class Animal
{
public:
    Animal(); //default constructor

    Animal(std::string name, bool domestic = false,
                             bool predator = false);//parameterized constructor

    // more code here


};// end Animal
```

Constructors Clarifications

- Multiple constructors, only one is invoked

- Initialize ALL data members in parameterized constructor, not only those with arguments

Constructors Clarifications

- Multiple constructors, only one is invoked

- Initialize ALL data members in parameterized constructor, not only those with arguments

- Explicitly call Base class constructor only if needs argument values or if there is no default to be called

```
class Fish: public Animal
{
public:
    Fish(); //default constructor

    Fish(std::string name, bool domestic = false,
                           bool predator = false);//parameterized constructor

    // more code here


};// end Fish
```

Fish.cpp

```
#include “Fish.hpp"


//default constructor
Fish::Fish(): venomous_{0}{}


//parameterized constructor
Fish::Fish(std::string name, bool domestic, bool predator):
           Animal(name, domestic, predator), venomous_{0}{}

//more code here . . .
```

Base class (Animal) constructor always called first. It will initialize derived data members.

Base class parameterized constructor needs access to argument values and must be called explicitly.

Recap

Bag ADT

Array implementation of Bag ADT

Algorithm Efficiency

Next: Linked implementation of Bag ADT

… but first

References ≠ Pointers

References Review

References

```
int x = 5;
int y = 8;


int& x_alias = x; //a reference or alias to x
```

References

```
int x = 5;
int y = 8;
```

We won’t do much of this

```
int& x_alias = x; //a reference or alias to x
```

Type	Name	Address	Data
int	x / x_alias	0x12345670	5
int	y	0x12345672	8

References

```
void increment(int x){    //pass by value: a copy of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter an integer for x: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “The value of x after calling increment is: “ << x;
}
```

References

```
void increment(int x){     //pass by value: a copy of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter an integer for x: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “The value of x after calling increment is: “ << x;
}
```

Function	Type	Name	Address	Data
main				
	int	x	0x12345670	5

References

```
void increment(int x){    //pass by value: a copy of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter an integer for x: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “The value of x after calling increment is: “ << x;
}
```

Function	Type	Name	Address	Data
increment	int	x	0x12345631	5
main				
	int	x	0x12345670	5

References

```
void increment(int x){     //pass by value: a copy of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter an integer for x: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “The value of x after calling increment is: “ << x;
}
```

Function	Type	Name	Address	Data
increment	int	x	0x12345631	6
main				
	int	x	0x12345670	5

References

```
void increment(int x){    //pass by value: a copy of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter an integer for x: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “The value of x after calling increment is: “ << x;
}
```

5

Function	Type	Name	Address	Data
main				
	int	x	0x12345670	5

References

The address of x

```
void increment(int& x){ //pass by reference: the address of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter a whole number: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “That number + 1 is: “ << x;
}
```

References

```
void increment(int& x){ //pass by reference: the address of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter a whole number: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “That number + 1 is: “ << x;
}
```

Function	Type	Name	Address	Data
increment	int&		0x12345670	
main				
	int	x	0x12345670	5

References

```
void increment(int& x){ //pass by reference: the address of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter a whole number: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “That number + 1 is: “ << x;
}
```

Function	Type	Name	Address	Data
increment	int&		0x12345670	
main				
	int	x	0x12345670	6

References

```
void increment(int& x){ //pass by reference: the address of x
    x+=1;
}

int main(){
    int x;
    std::cout << “Enter a whole number: ” << std::endl;
    std::cin >> x;
    increment(x);
    std::cout << “That number + 1 is: “ << x;
}
```

6

Function	Type	Name	Address	Data
main				
	int	x	0x12345670	6

Pointers and Dynamic Memory Allocation (Review)

Pointer Variables

A typed variable whose value is the address of another variable of same type

```
int x = 5;
int y = 8;
int *p, *q = nullptr;  //declares two int pointers

. . .
```

Make sure you do this if not assigning a value!

Program Stack

Type	Name	Address	Data
...	...	...	...
int	x	0x12345670	5
int	y	0x12345674	8
int pointer	p	0x12345678	`nullptr`
int pointer	q	0x1234567C	`nullptr`
...	...	...	...

```
int x = 5;
int y = 8;
int *p, *q = nullptr;  //declares two int pointers


. . .
p = &x;       // sets p to the address of x
q = &y;       // sets q address of y
```

Make sure you do this if not assigning a value!

Program Stack

Type	Name	Address	Data
...	...	...	...
int	x	0x12345670	5
int	y	0x12345674	8
int pointer	p	0x12345678	0x12345670
int pointer	q	0x1234567C	0x12345674
...	...	...	...

```
int x = 5;
int y = 8;
int *p, *q = nullptr;  //declares two int pointers

. . .
p = &x;       // sets p to the address of x
q = &y;       // sets q address of y
std::cout << p; // the value of p: address 0x12345670
std::cout << *p; // dereferencing: prints 5, the value of
                 // the variable it points to
```

Make sure you do this if not assigning a value!

We won't do much of this

Program Stack

Type	Name	Address	Data
...	...	...	...
int	x	0x12345670	5
int	y	0x12345674	8
int pointer	p	0x12345678	0x12345670
int pointer	q	0x1234567C	0x12345674
...	...	...	...

Dynamic Variables

Memory is allocated statically on the program stack at compile time

What if I cannot statically allocate data? (e.g. will be reading from input at runtime)

Allocate dynamically on the heap with **`new`**

Dynamic Variables

Created at runtime in the memory **heap**
using operator **new**

Nameless typed variables accessed through pointers

```
// create a nameless variable of type dataType on the
//application heap and stores its address in p
dataType *p = new dataType;
```

Program Stack

Type	Name	Address	Data
...	...	...	...
dataType ptr	p	0x12345678	0x100436f20
...	...	...	...

Heap

Type	Address	Data
...	...	...
dataType	0x100436f20	
...	...	...

Accessing members

```
dataType some_object;
dataType *p = new dataType;
// initialize and do stuff with instantiated objects

. . .

string my_string = some_object.getName();
string another_string = p->getName();
```

To access member functions in place of . operator

Deallocating Memory

```
delete p;
p = nullptr;
```

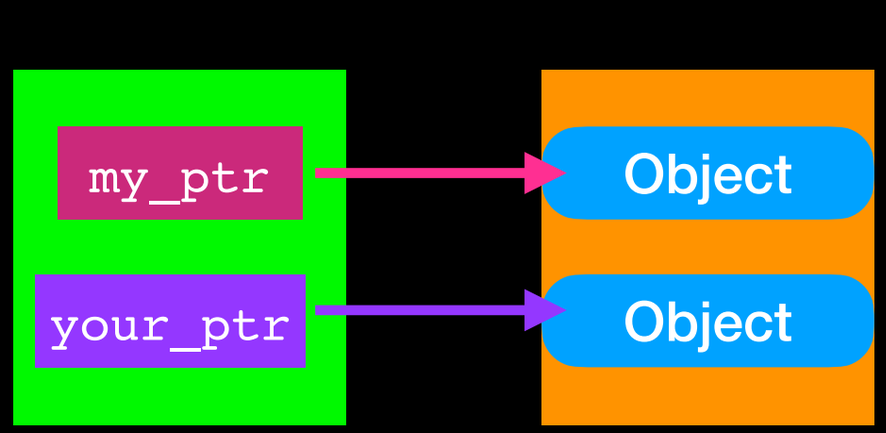
Deletes **the object** pointed to by p

Must do this!!!

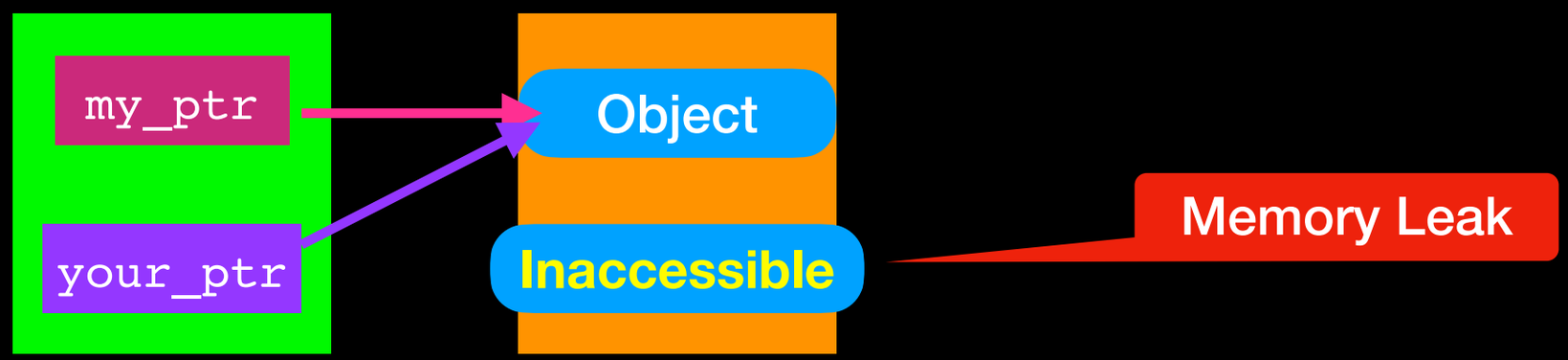
Avoid Memory Leaks (1)

Occurs when object is created in free store but program no longer has access to it

```
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
```



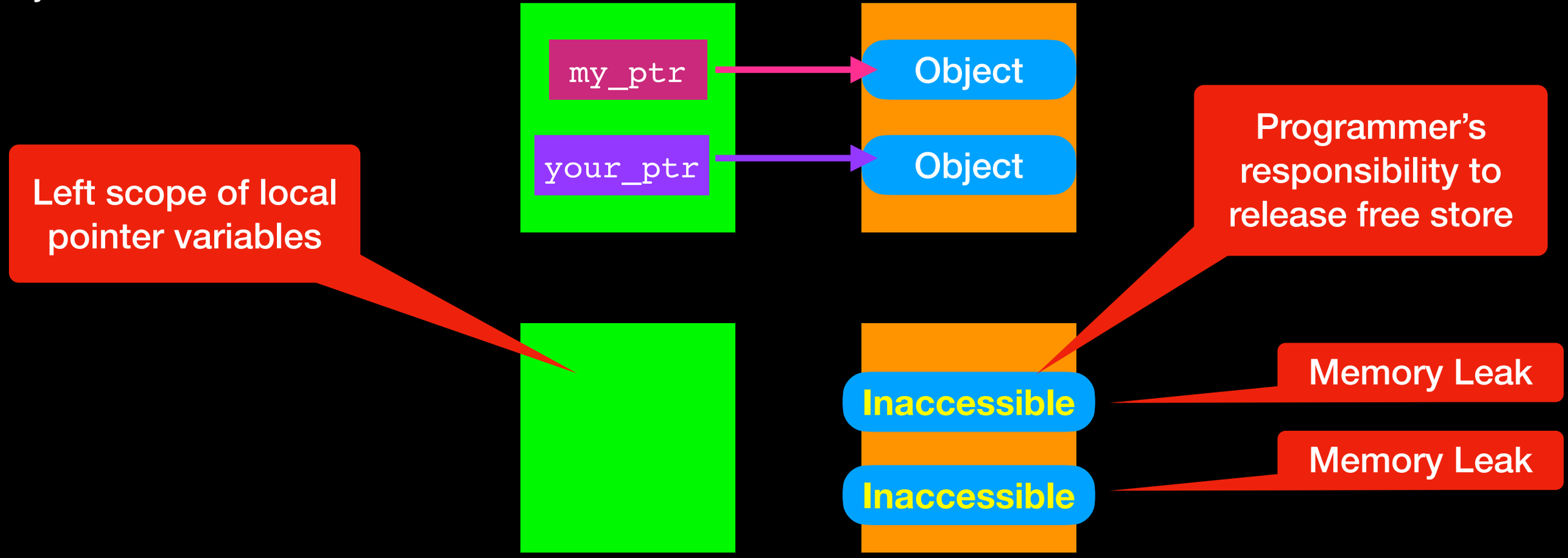

```
your_ptr = my_ptr;
```



Avoid Memory Leaks (2)

Occurs when object is created in free store but program no longer has access to it

```
void leakyFunction(){
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
}
```

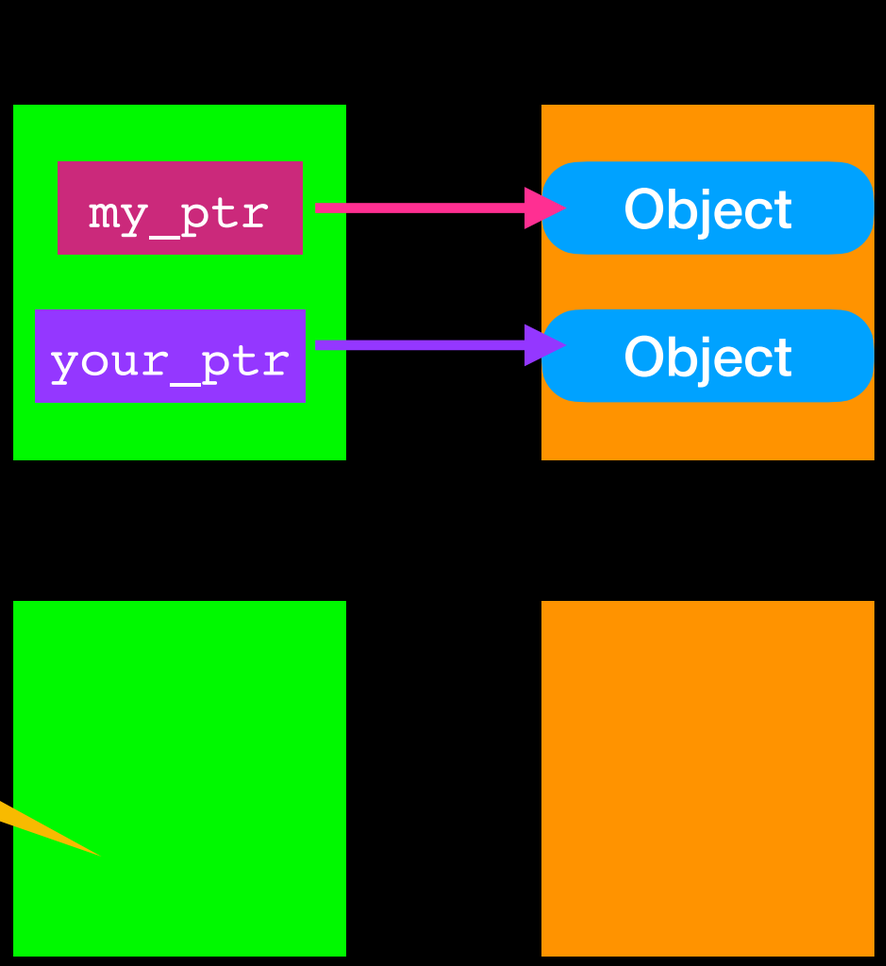


Avoid Memory Leaks (2)

Occurs when object is created in free store but program no longer has access to it

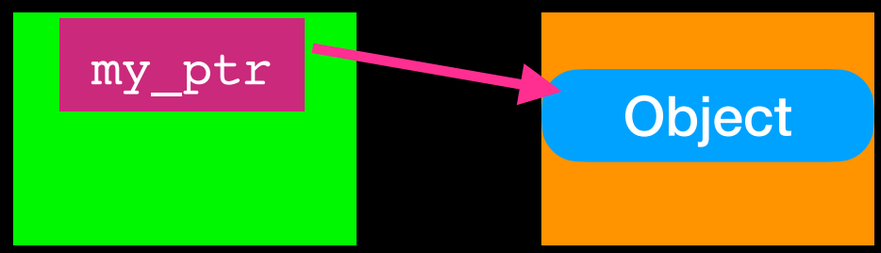
```
void leakyFunctionFixed(){
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
delete my_ptr;
my_ptr = nullptr;
delete your_ptr;
your_ptr = nullptr;
}
```

Left scope of local pointer variables but deleted dynamic objects first

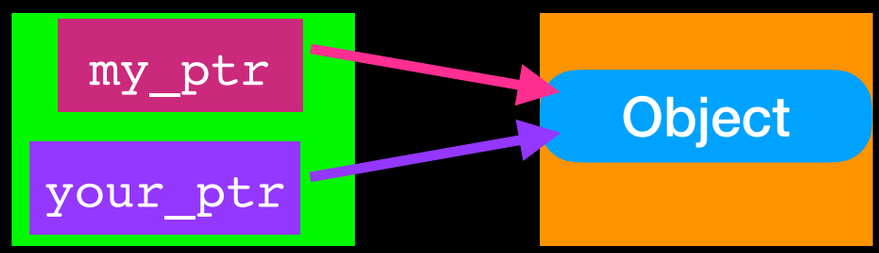


Moving Pointers

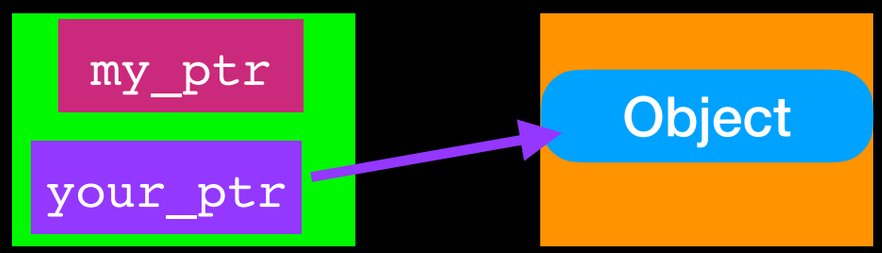
```
dataType *my_ptr = new dataType;
```




```
dataType *your_ptr = my_ptr;
```

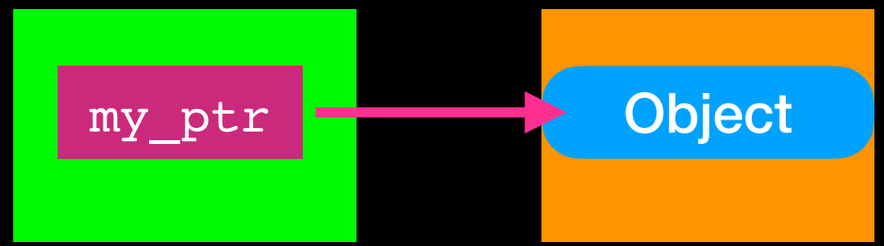



```
my_ptr = nullptr;
```

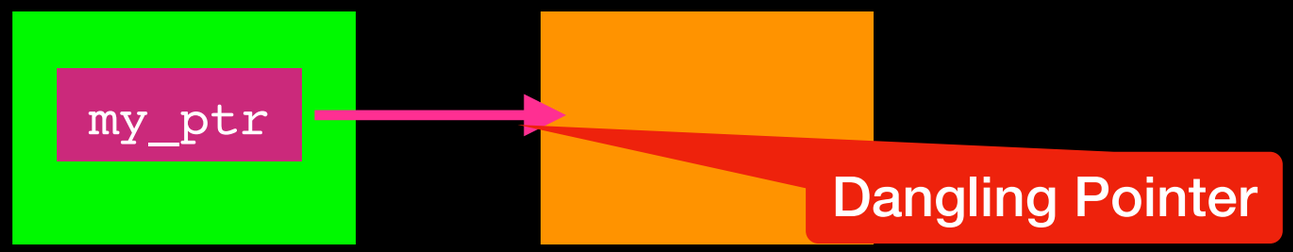


Avoid Dangling Pointers(1)

Pointer variable that no longer references a valid object



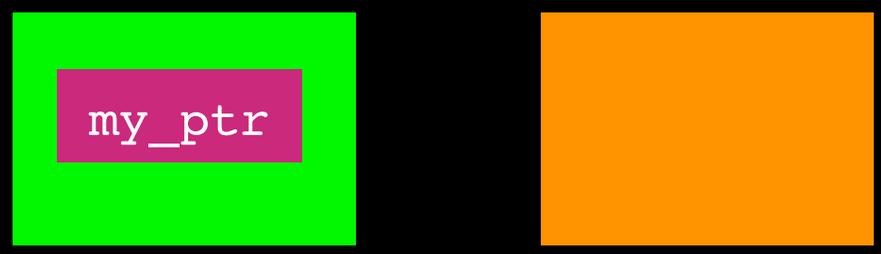

```
delete my_ptr;
```



Fix

```
delete my_ptr;
my_ptr = nullptr;
```

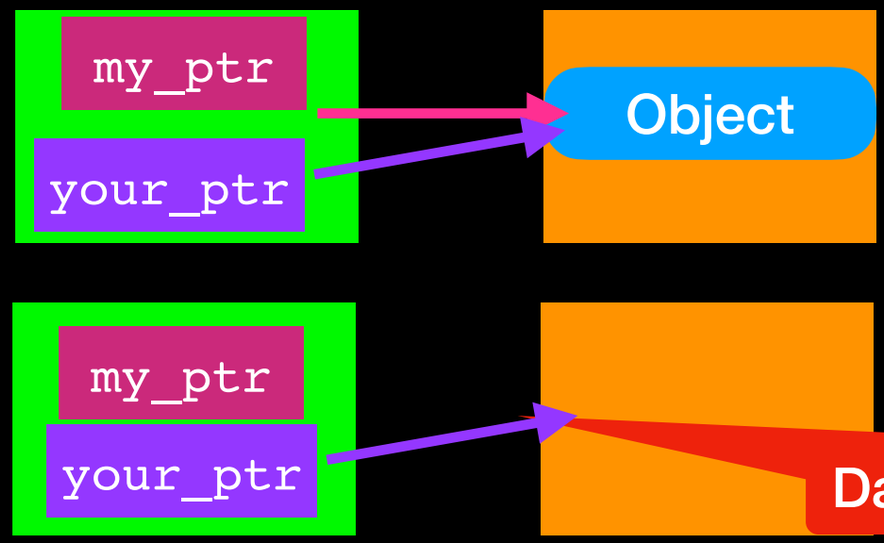
Must do this!!!



Avoid Dangling Pointers(2)

Pointer variable that no longer references a valid object

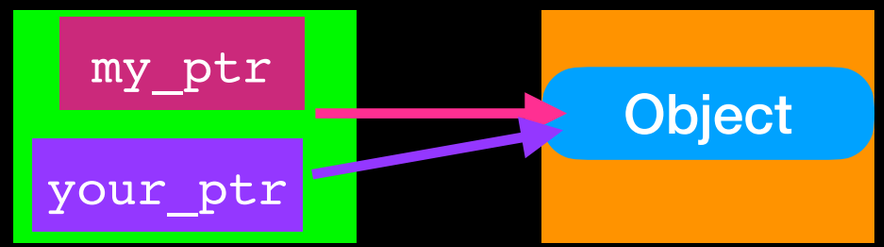
```
delete my_ptr;
my_ptr = nullptr;
```

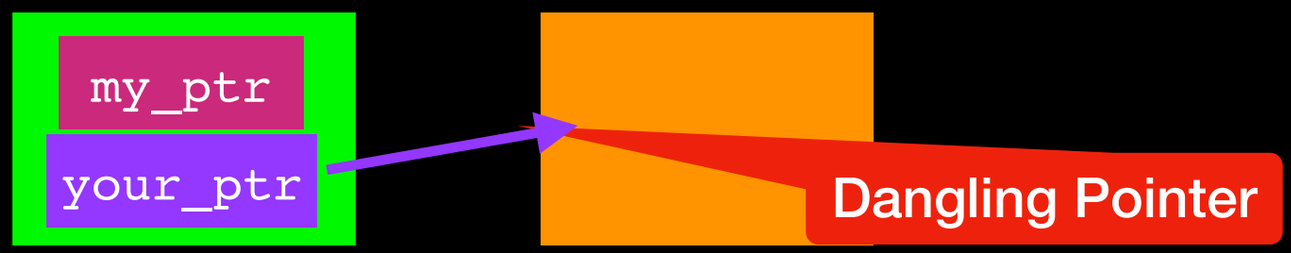


Avoid Dangling Pointers(2)

Pointer variable that no longer references a valid object

```
delete my_ptr;
my_ptr = nullptr;
```



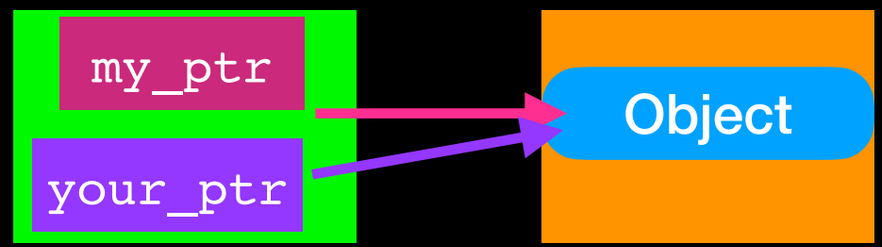


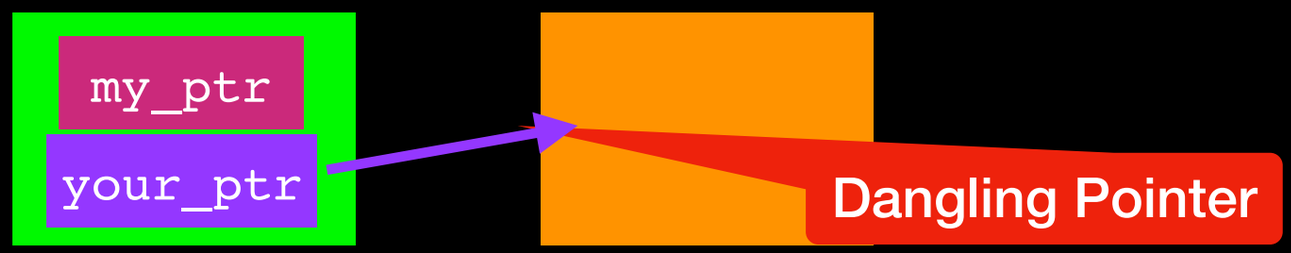
```
delete your_ptr;// ERROR!!!! No object to delete
```

Avoid Dangling Pointers(2)

Pointer variable that no longer references a valid object

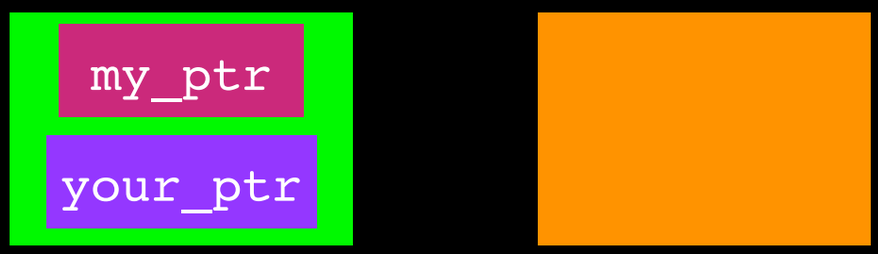
```
delete my_ptr;
my_ptr = nullptr;
```





Fix

```
delete my_ptr;
my_ptr = nullptr;
your_ptr = nullptr;
```



Must set all pointers to `nullptr`!!!

Lecture Activity

What is wrong with the following code?

```
void someFunction()
{
   int* p = new int[5];
   int* q = new int[10];

   p[2] = 9;
   q[2] = p[2]+5;
   p[0] = 8;
   q[7] = 15;

   std::cout<< p[2] << " " <<  q[2] << std::endl;
   q = p;
   std::cout<< p[0]  << " " << q[7] << std::endl;

}
```

What is wrong with the following code?

```
void someFunction()
{
   int* p = new int[5];
   int* q = new int[10];

   p[2] = 9;
   q[2] = p[2]+5;
   p[0] = 8;
   q[7] = 15;

   std::cout<< p[2] << " " <<  q[2] << std::endl;
   q = p;
   std::cout<< p[0]  << " " << q[7] << std::endl;

}
```

MEMORY LEAK:
`int[10]` **lost on heap**

SEGMENTATION FAULT
`int[5]` **index out of range**

MEMORY LEAK:
Did not delete `int[5]`
before exiting function

Next Time

Let’s try a different (linked) implementation of the Bag ADT