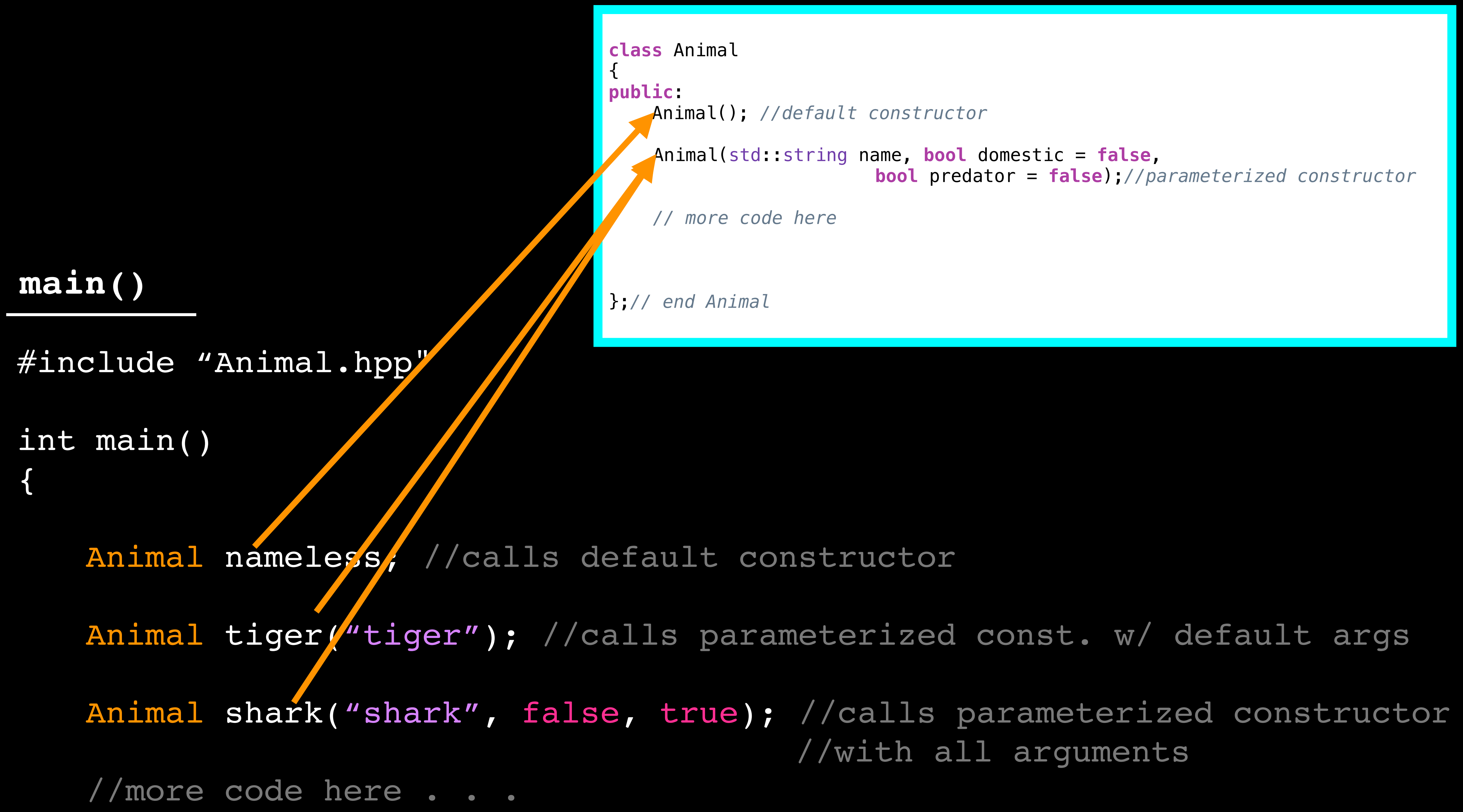
Pointers and Dynamic Memory Allocation

Tiziana Ligorio

Constructors Clarifications

- Multiple constructors, only one is invoked

main()



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

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

    // more code here


};// end Animal
```

```
#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
```

Constructors Clarifications

- Multiple constructors, only one is invoked

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

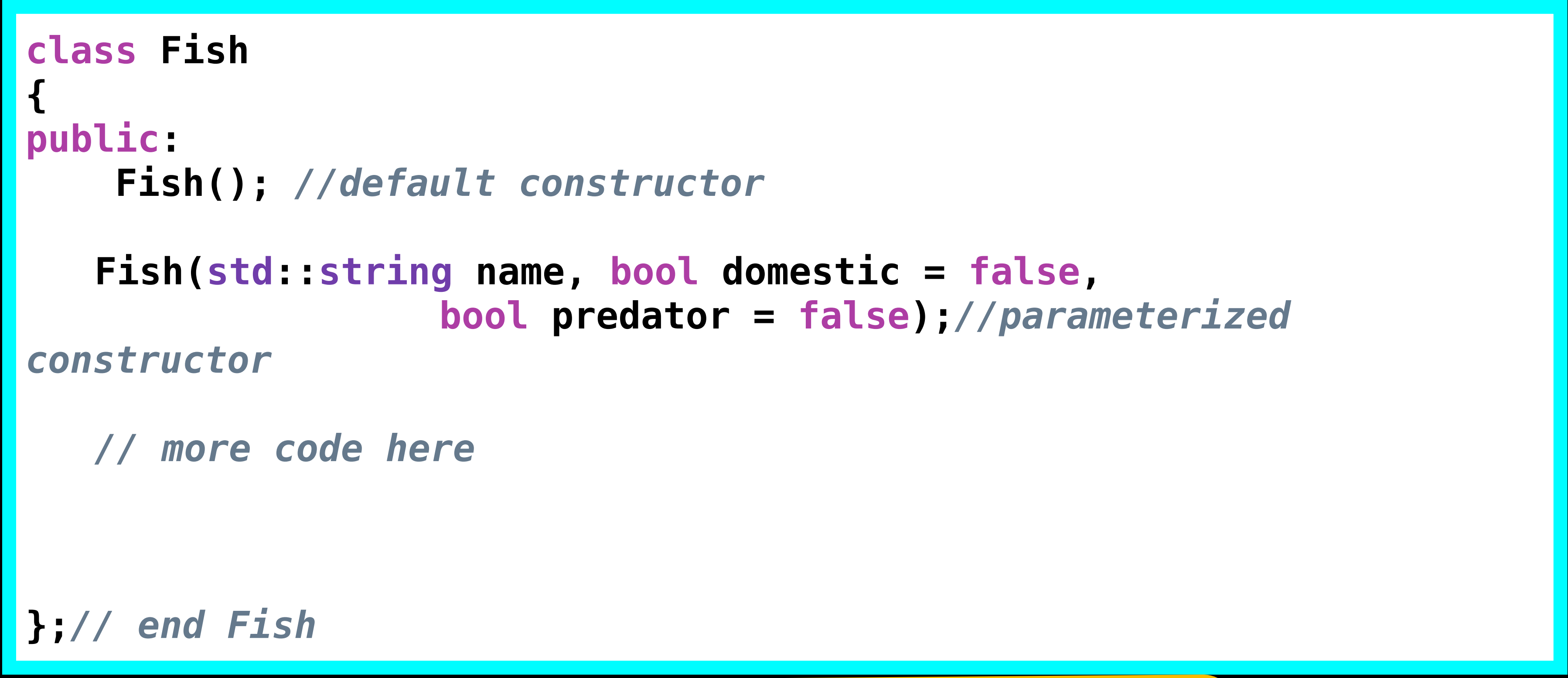
- Explicitly call Base class constructor only if needs argument values or if there is no default to be called

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:
    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){}
```

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

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

//more code here . . .
```

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

Pointer Variables

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

Make sure you do this if not assigning a value!

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


. . .
```

Program Stack

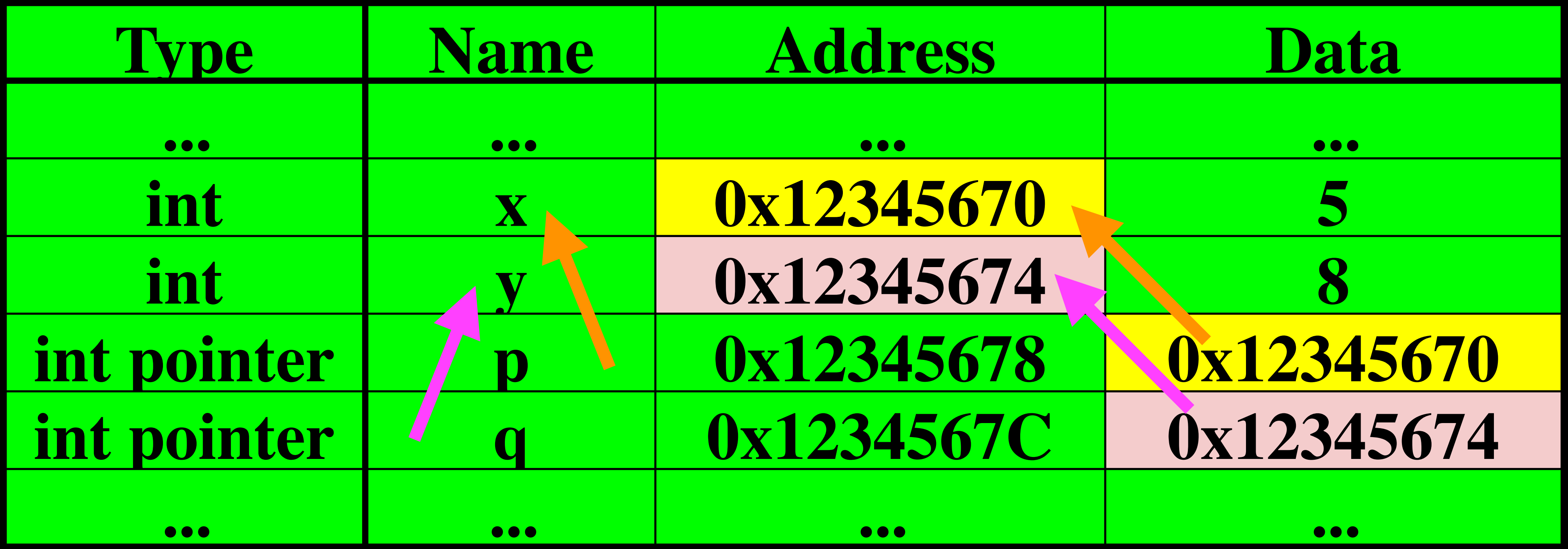
Type	Name	Address	Data
...	...	...	...
int	x	0x12345670	5
int	y	0x12345674	8
int pointer	p	0x12345678	nullptr
int pointer	q	0x1234567C	nullptr
...	...	...	...

Make sure you do this if not assigning a value!

```
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
```

Program Stack



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

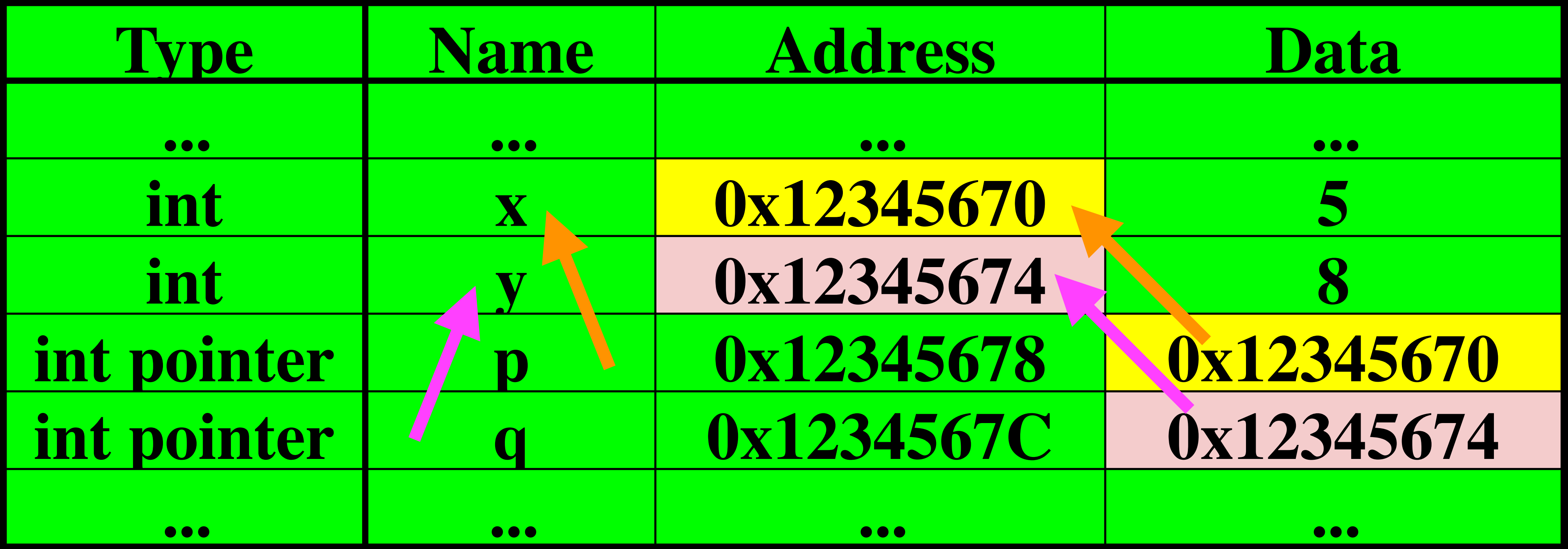
Make sure you do this if not assigning a value!

```
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
```

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
…	…	…	…

Recall Dynamic Variables

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

Recall Dynamic Variables

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

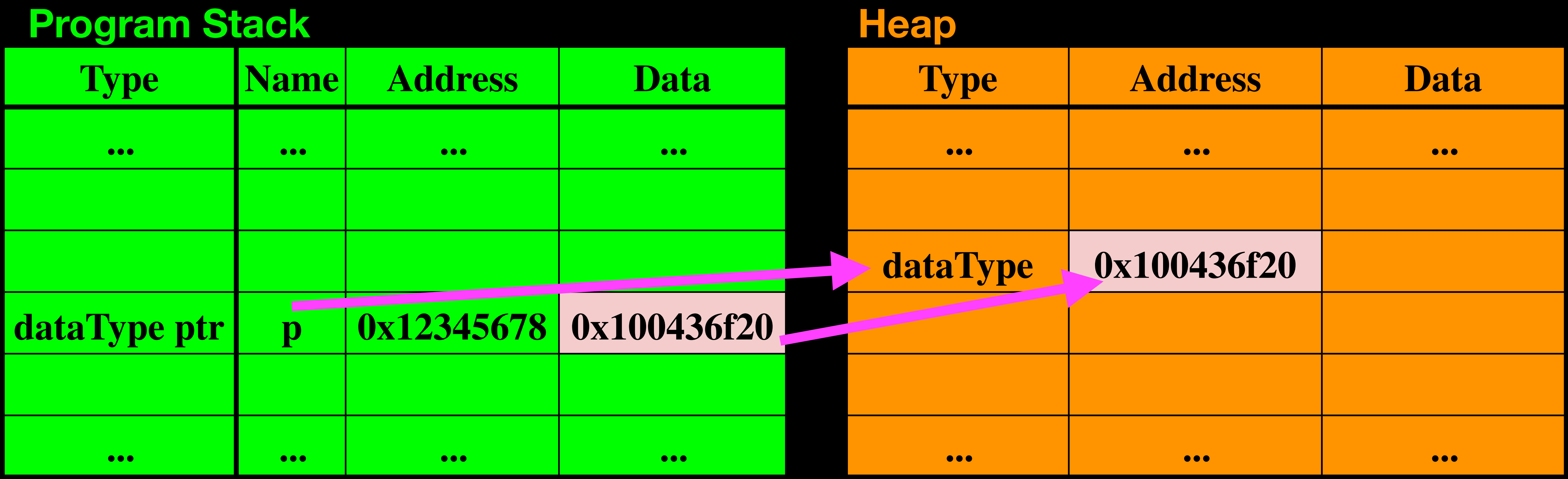
Allocate dynamically with **`new`**

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;
```

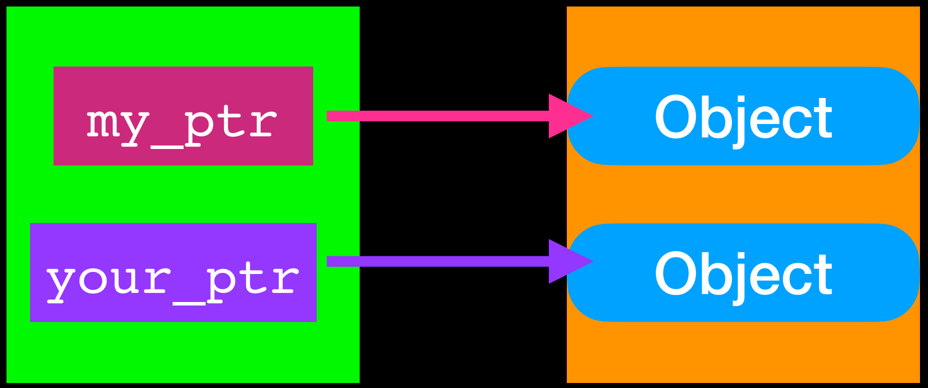
Deletes the object pointed to by p

Must do this!!!

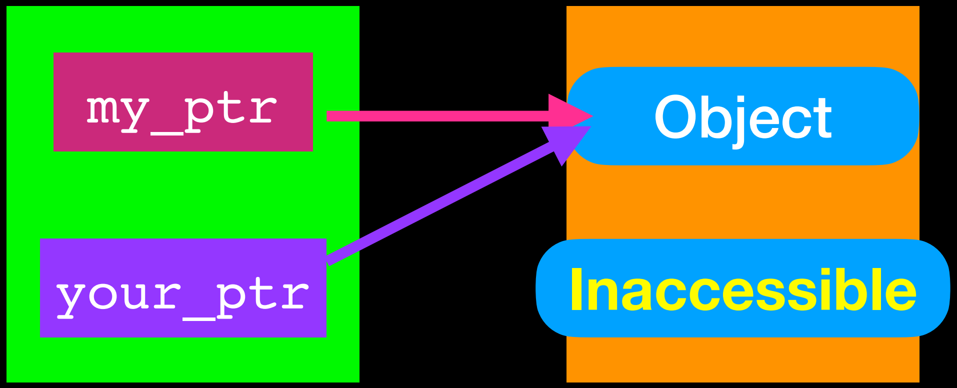
Avoid Memory Leaks (1)

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;
```

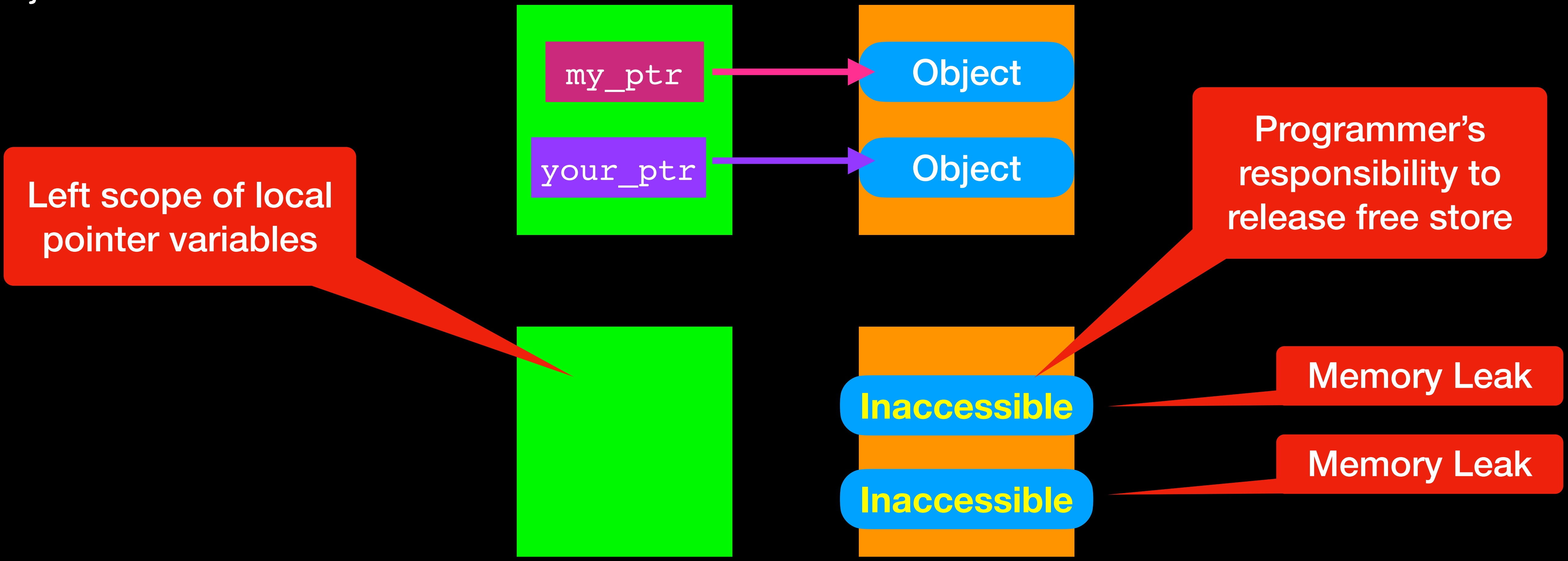


Memory Leak

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
}
```



my_ptr

your_ptr

Object

Object

Left scope of local pointer variables

Programmer’s responsibility to release free store

Inaccessible

Inaccessible

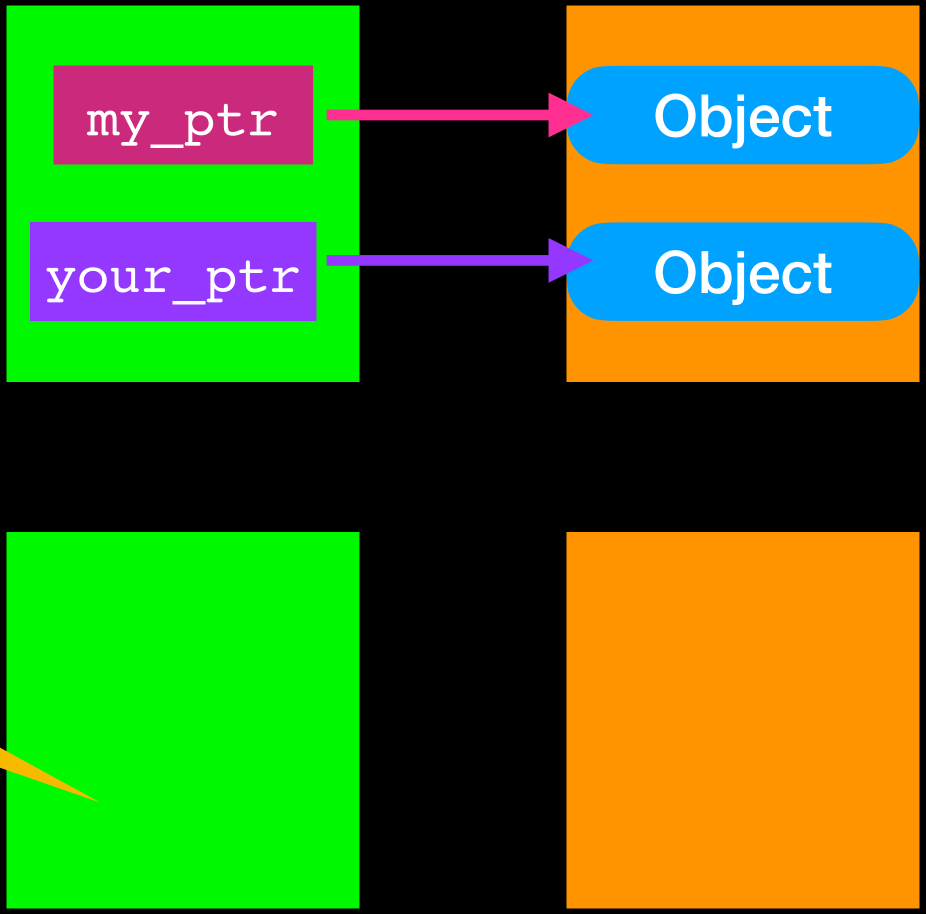
Memory Leak

Memory Leak

Avoid Memory Leaks (2)

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

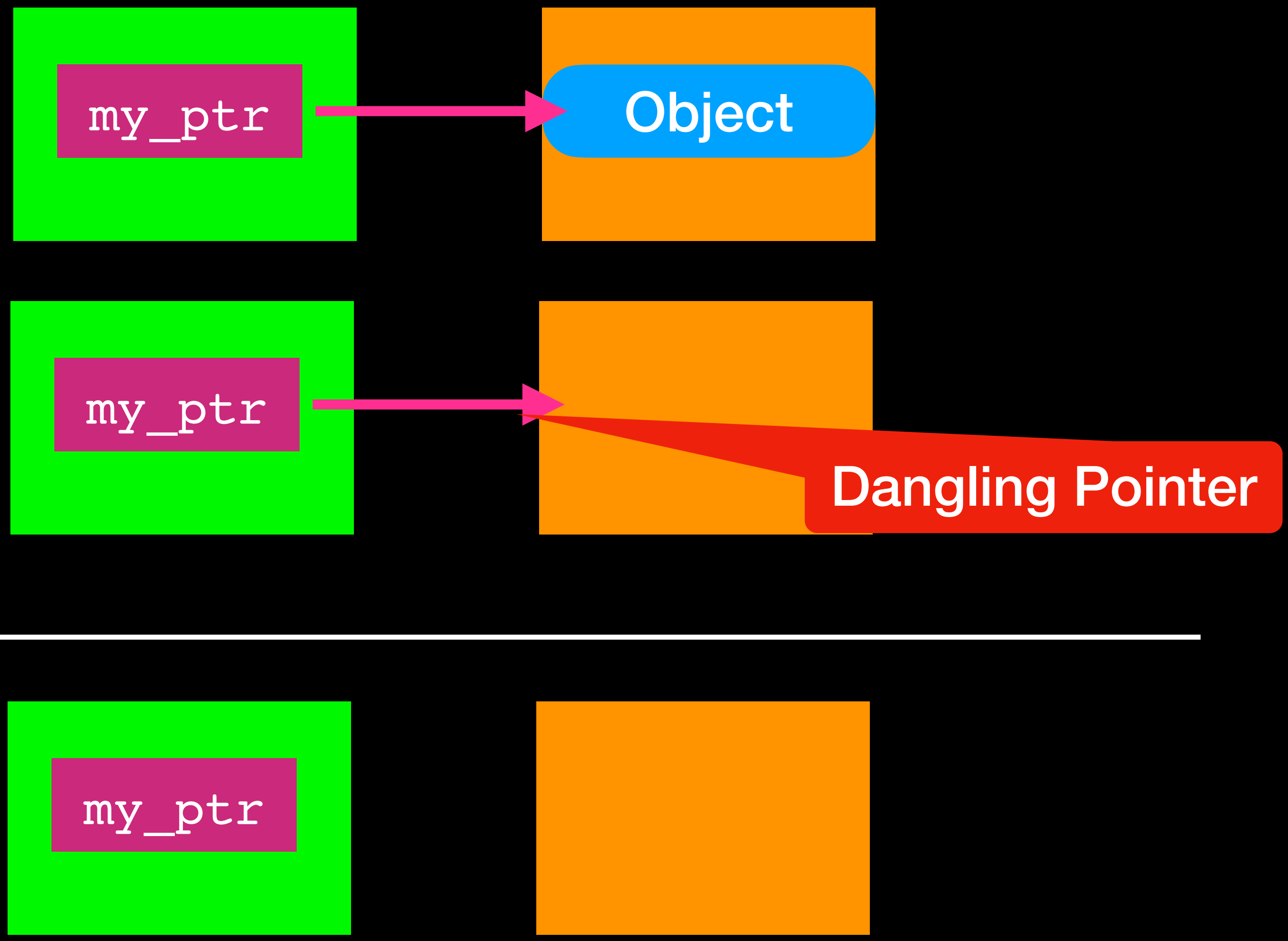
```
void leakyFunctionFixed(){
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
delete my_ptr;
my_ptr = nullptr;
delete your_ptr;
your_ptr = nullptr;
}
```



Left scope of local pointer variables but deleted dynamic objects first

Avoid Dangling Pointers

Pointer variable that no longer references a valid object



```
delete my_ptr;
```

Fix

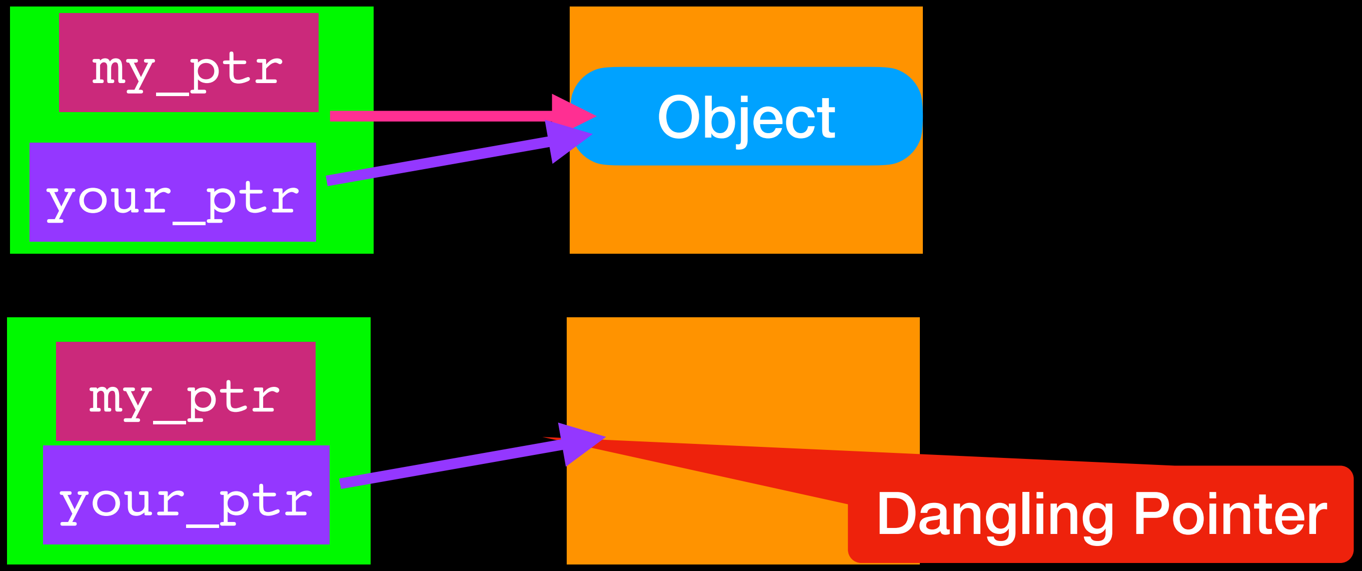
```
delete my_ptr;
my_ptr = nullptr;
```

Must do this!!!

Avoid Dangling Pointers

Pointer variable that no longer references a valid object

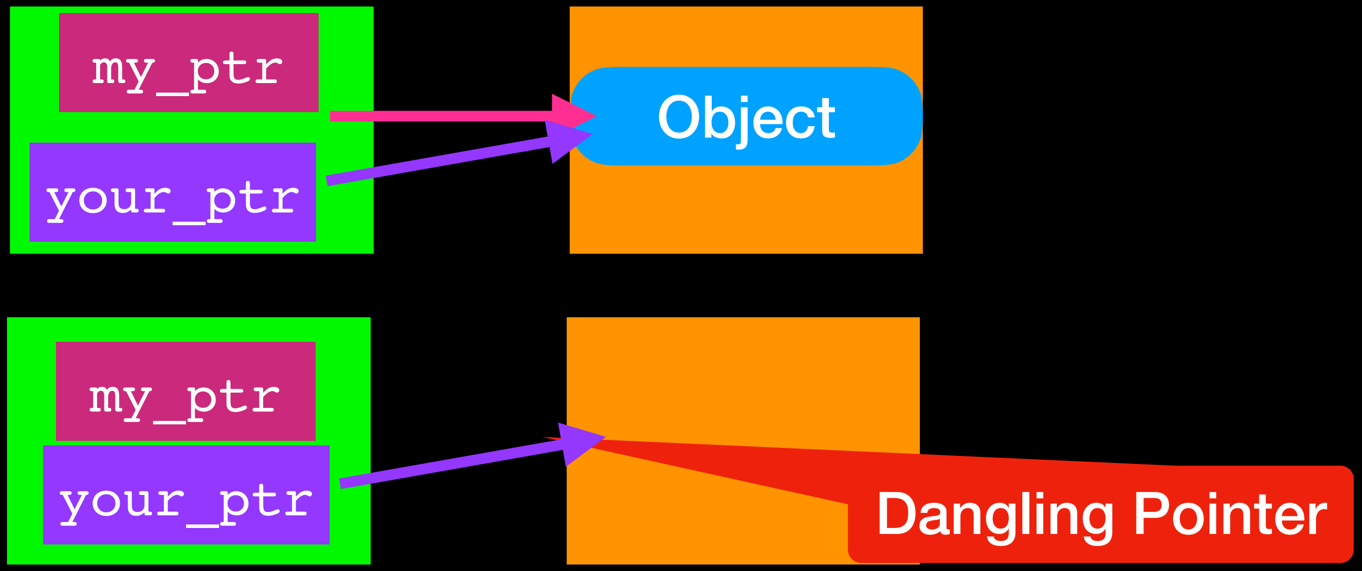
```
delete my_ptr;
my_ptr = nullptr;
```



Avoid Dangling Pointers

Pointer variable that no longer references a valid object

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

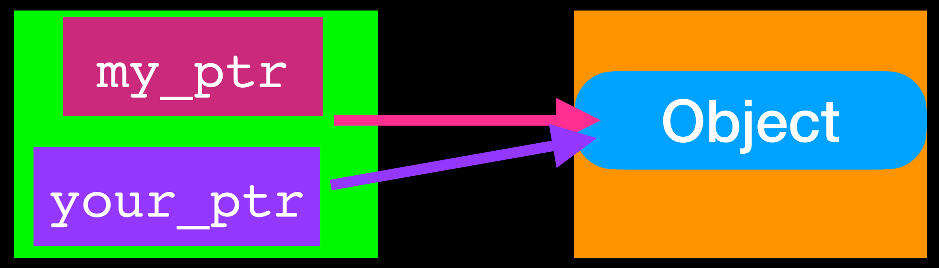


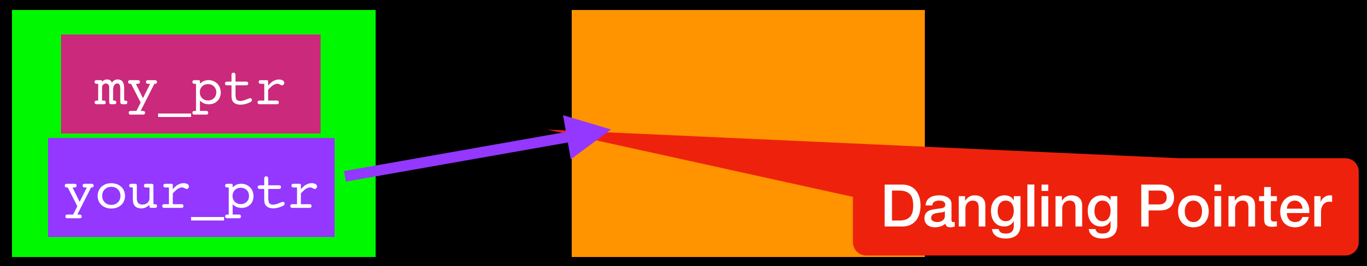
```
delete your_ptr;// ERROR!!!! No object to delete
```

Avoid Dangling Pointers

Pointer variable that no longer references a valid object

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

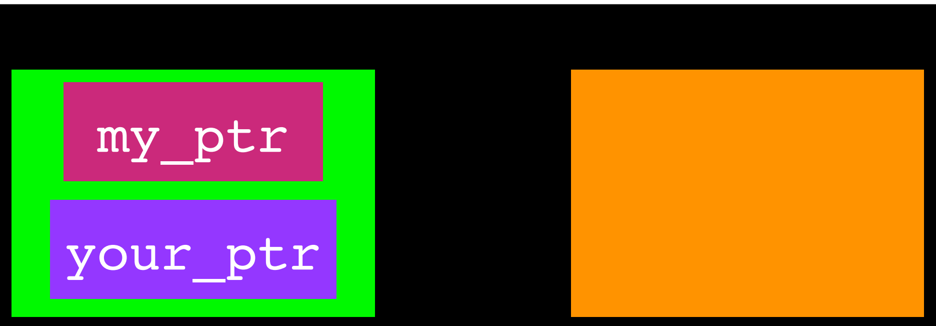




Dangling Pointer

Fix

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



Must set all pointers to `nullptr`!!!

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 let's try a different implementation for Bag