[368] Object Oriented Programming

Tyler Caraza-Harter

Outline

TopHat and Worksheet

Methods

Encapsulation

Constructors/Destructors

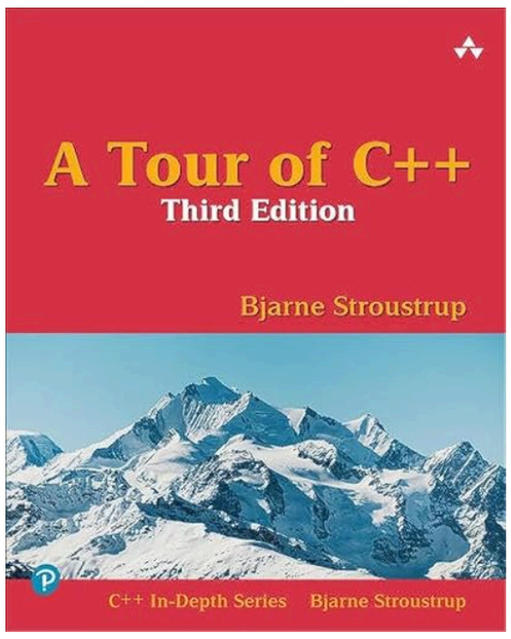
Copy Constructors

Demos

C++ History

Fun Fact: C++ was originally called "C with Classes"

- 1979: Work on "C with Classes" started
- 1984: "C with Classes" was renamed to C++



Ch 19.1

What will you learn today?

Learning objectives

- organize data and related functions together using structs/classes
- control visibility with public/private/friend in order to encapsulate internal implementation details, separate from the public interface
- identify where in code constructors and destructors will execute
- write copy constructors together with with destructors to avoid memory bugs (specifically leaks and double frees)

Outline

TopHat and Worksheet

Methods

Encapsulation

Constructors/Destructors

Copy Constructors

Demos

Motivation for Methods

```
struct Cat {
    ...
}

struct Dog {
    ...
}
```

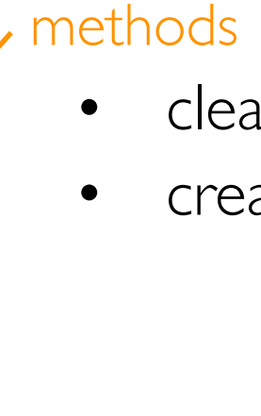
in C, it is very common (and cumbersome) to have a collection of functions for each struct that:

- have a name prefixed with struct name
- take a pointer to struct as first param

```
void dog_speak(struct *Dog this) {...}
void cat_speak(struct *Cat this) {...}

struct Dog d;
struct Cat c;

dog_speak(&d);
cat_speak(&c);
```

Motivation for Methods

```
struct Dog d;
struct Cat c;

d.speak();
c.speak();
```



- cleaner syntax
- create possibility to override inherited methods

Outline

TopHat and Worksheet

Methods

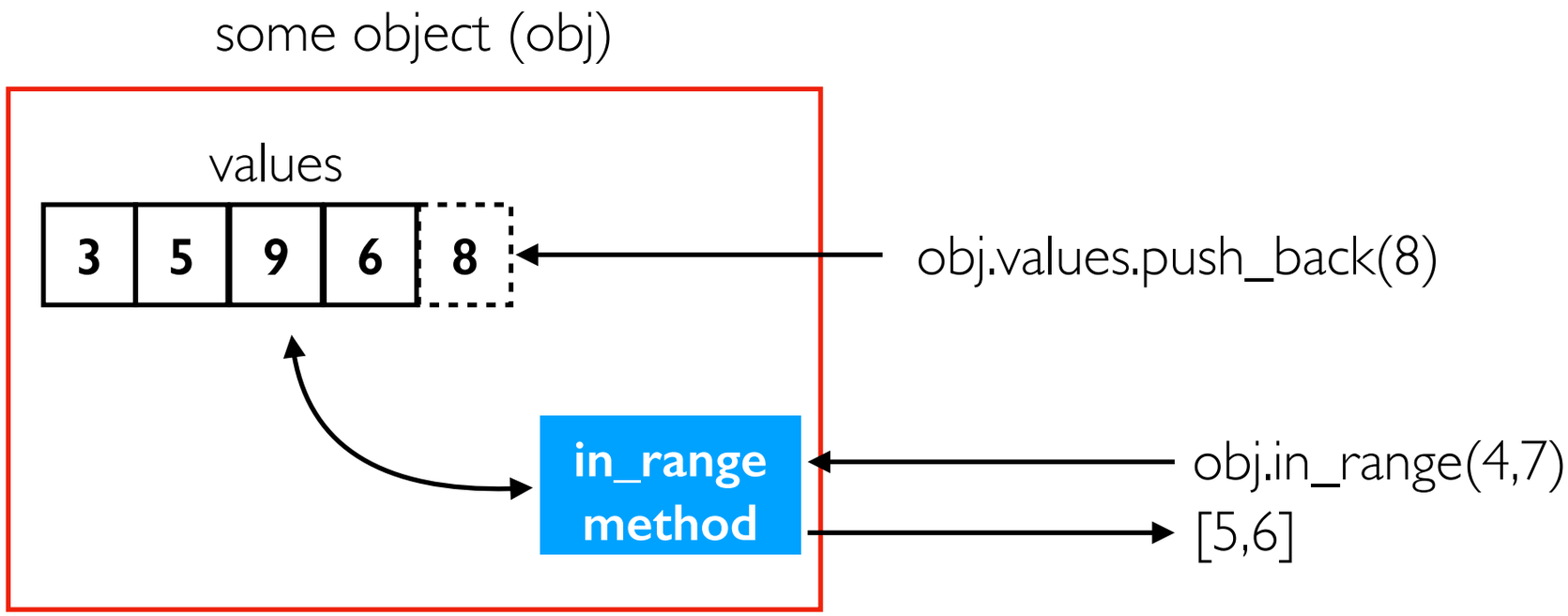
Encapsulation

Constructors/Destructors

Copy Constructors

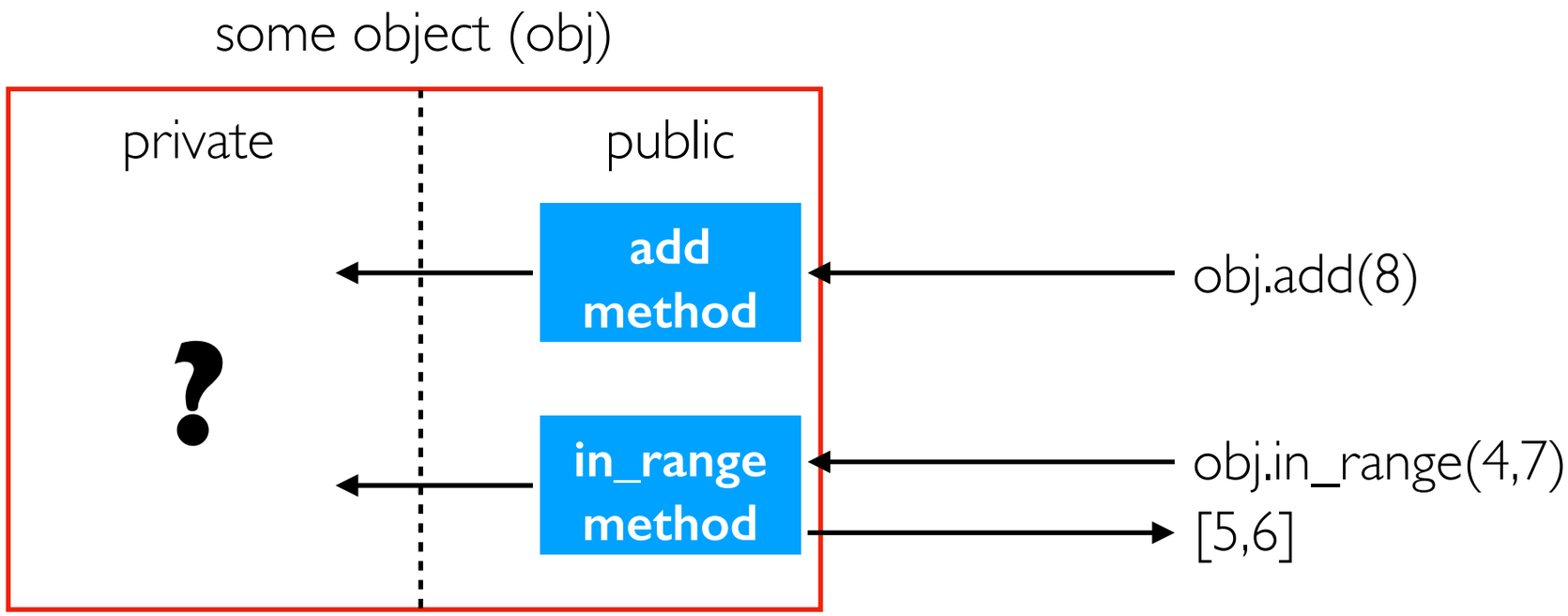
Demos

Motivation for Encapsulation



- if we add frequently and call in_range rarely, this implementation is good
- what if we call in_range frequently? Can we improve the library without breaking all the programs that use the library?

Motivation for Encapsulation



Keeping implementation private lets us change the implementation without breaking users!

- perhaps a binary search tree is a better data structure than a vector
- can also easily change the code (but not declaration!) of the public methods. For example, perhaps **add** should sort a vector of values after each number is added.

Outline

TopHat and Worksheet

Methods

Encapsulation

Constructors/Destructors

Copy Constructors

Demos

Creating an Object

Allocation: getting some memory for an object

- malloc *allocates* memory on the **heap**
- calling a function (or creating local variables) *allocates* memory on the **stack**

Initialization: setting some values in that object

- a constructor *initializes* that memory. You write the constructor!

When it happens

- Heap: new (1) calls malloc then (2) calls the constructor
- Stack: local variable creation (1) allocates stack memory then (2) calls the constructor

Constructors

```python
# Python
class Coord:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```java
// Java
public class Coord {
    private int x;
    private int y;

    public Coord(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```cpp
// C++
class Coord {
private:
    int x;
    int y;

public:
    Coord(int x, int y) : x(x), y(y) {}
};
```

Releasing an Object

Deallocation: giving back some memory for an object object

- Java/Python: deallocation is performed by the garbage collector. Hard to say when it will happen!
- C++: deallocation happens when a local variable goes out of scope (stack) or we call delete (heap). Can determine exactly when it will happen.

Destruction: performing additional cleanup

- a C++ destructor performs additional cleanup work, just before deallocation. You write the constructor!
- Java/Python: don't have serious destructors (Python has __del__, but you can't know if/when it will be called)

Examples of extra cleanup

- closing a file
- closing a socket
- calling delete on a pointer from the object being destroyed to another object (when we know we have the only pointer to the second object)
- logging something (e.g., how long the object lived)

Outline

TopHat and Worksheet

Methods

Encapsulation

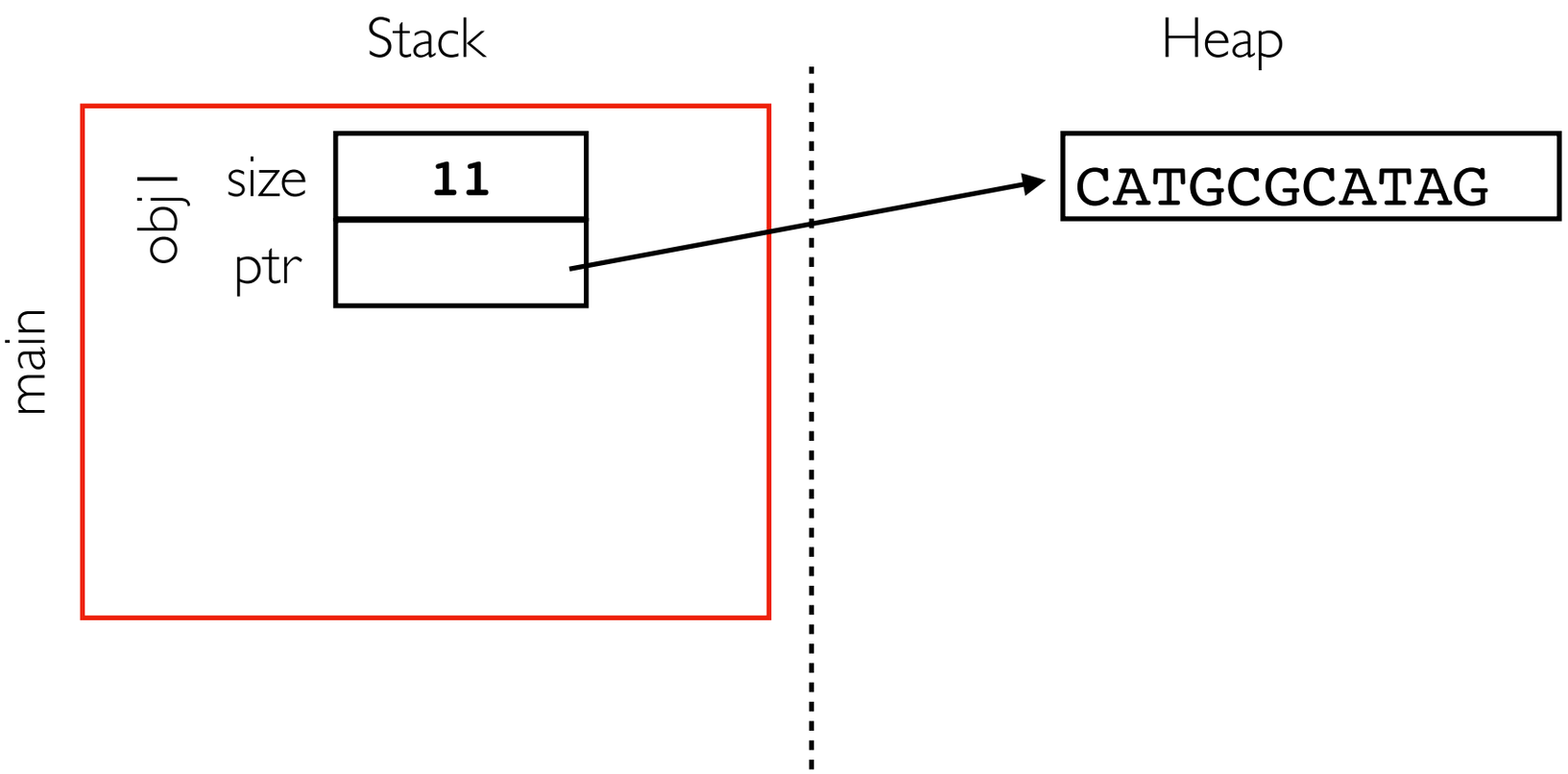
Constructors/Destructors

Copy Constructors

Demos

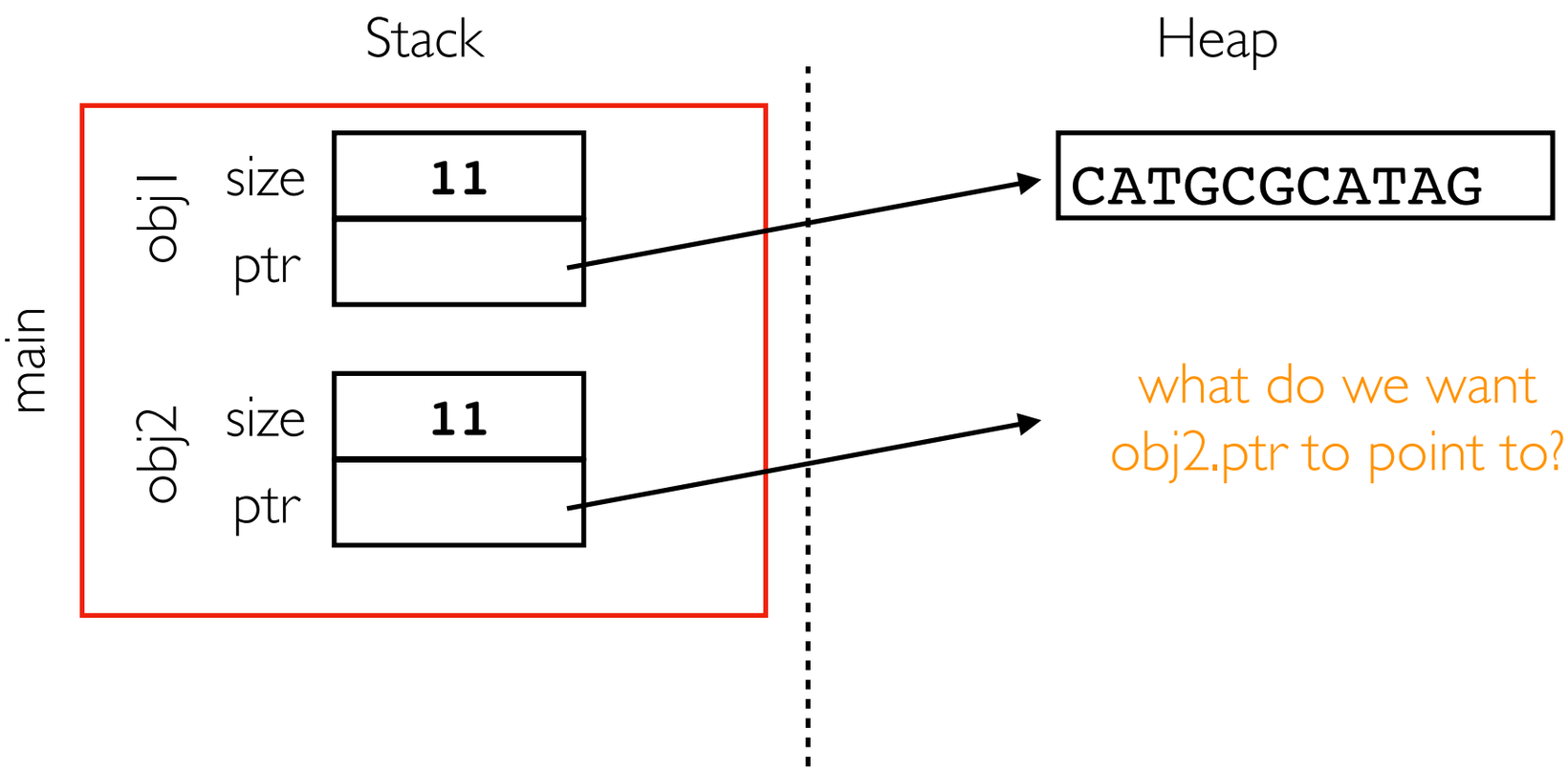
Copy Motivation

```
// CODE:
DNA obj1{"CATGCGCATAG};
```



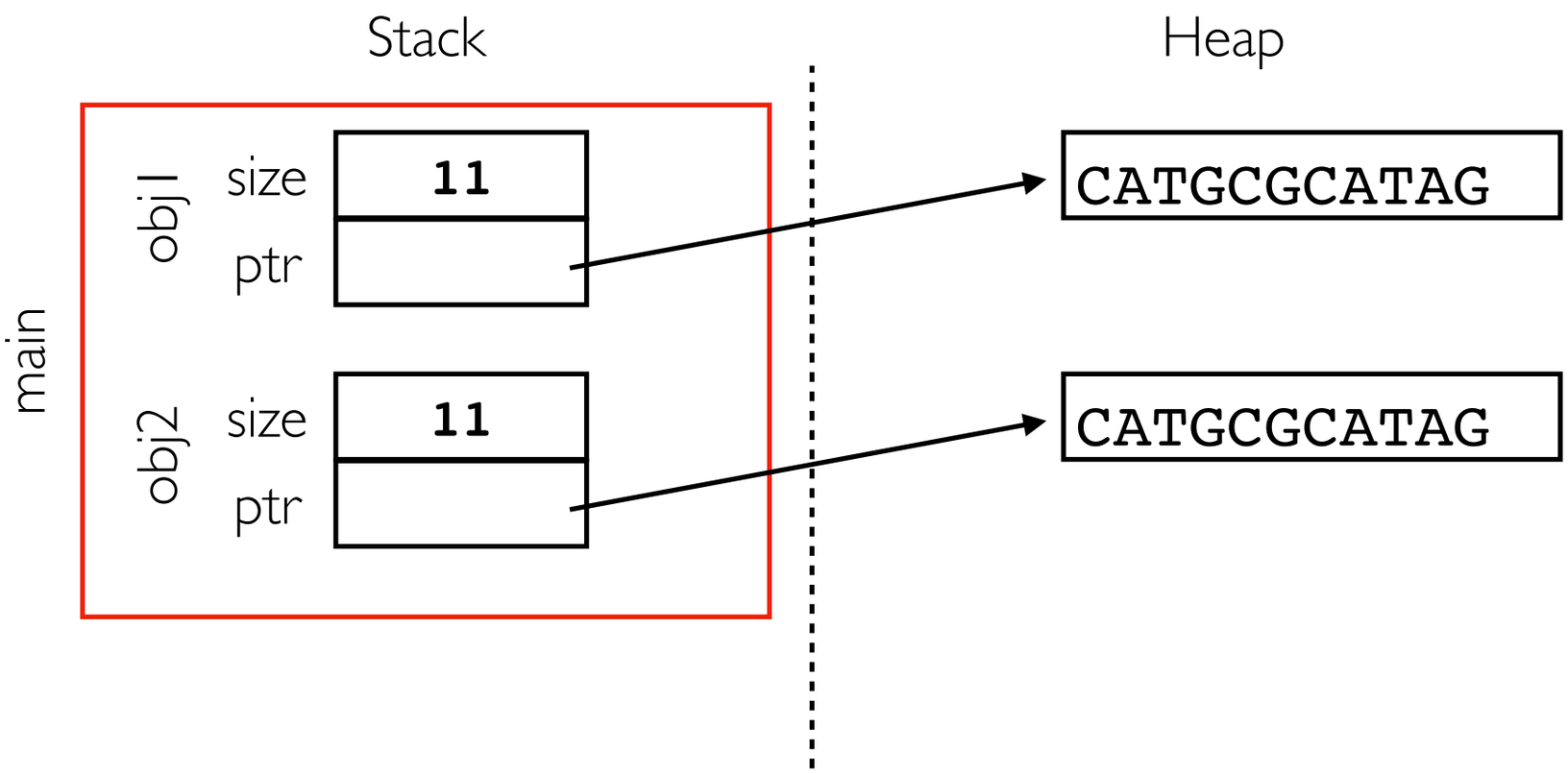
Copy Motivation

```
// CODE:
DNA obj1{"CATGCGCATAG};
DNA obj2 = obj1;
```



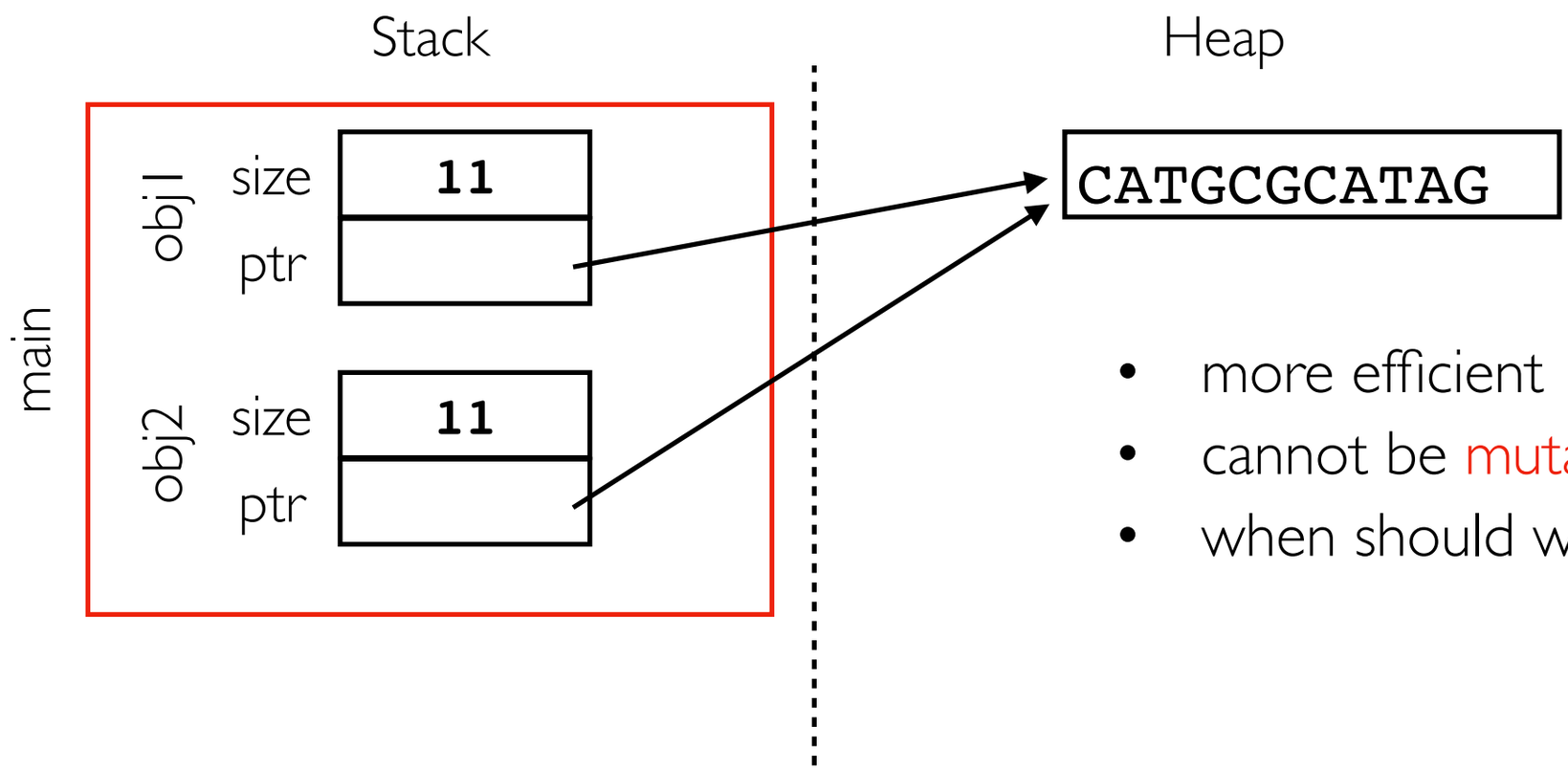
Option 1: Copy Children Too

```
// CODE:
DNA obj1{"CATGCGCATAG};
DNA obj2 = obj1;
```



Option 2: Share Heap Memory

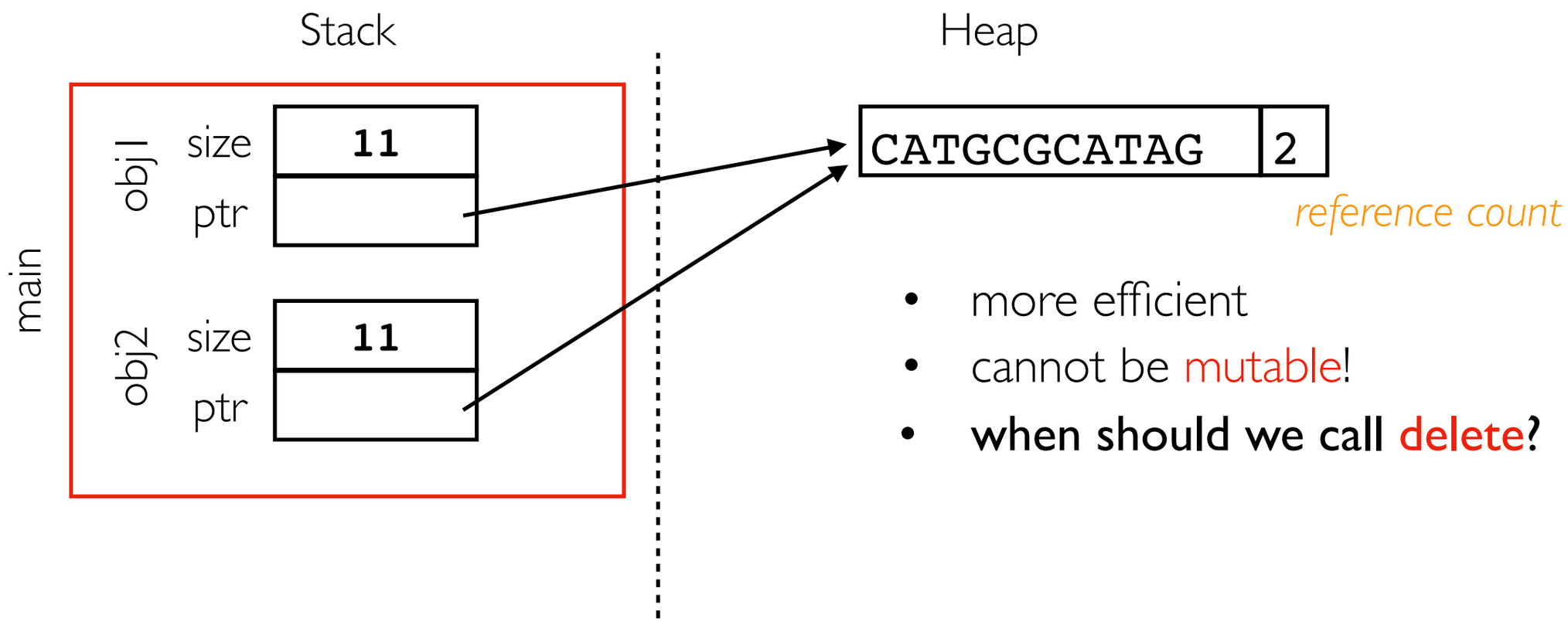
```
// CODE:
DNA obj1{"CATGCGCATAG};
DNA obj2 = obj1;
```



- more efficient
- cannot be mutable!
- when should we call delete?

Option 2: Share Heap Memory

```
// CODE:
DNA obj1{"CATGCGCATAG};
DNA obj2 = obj1;
```



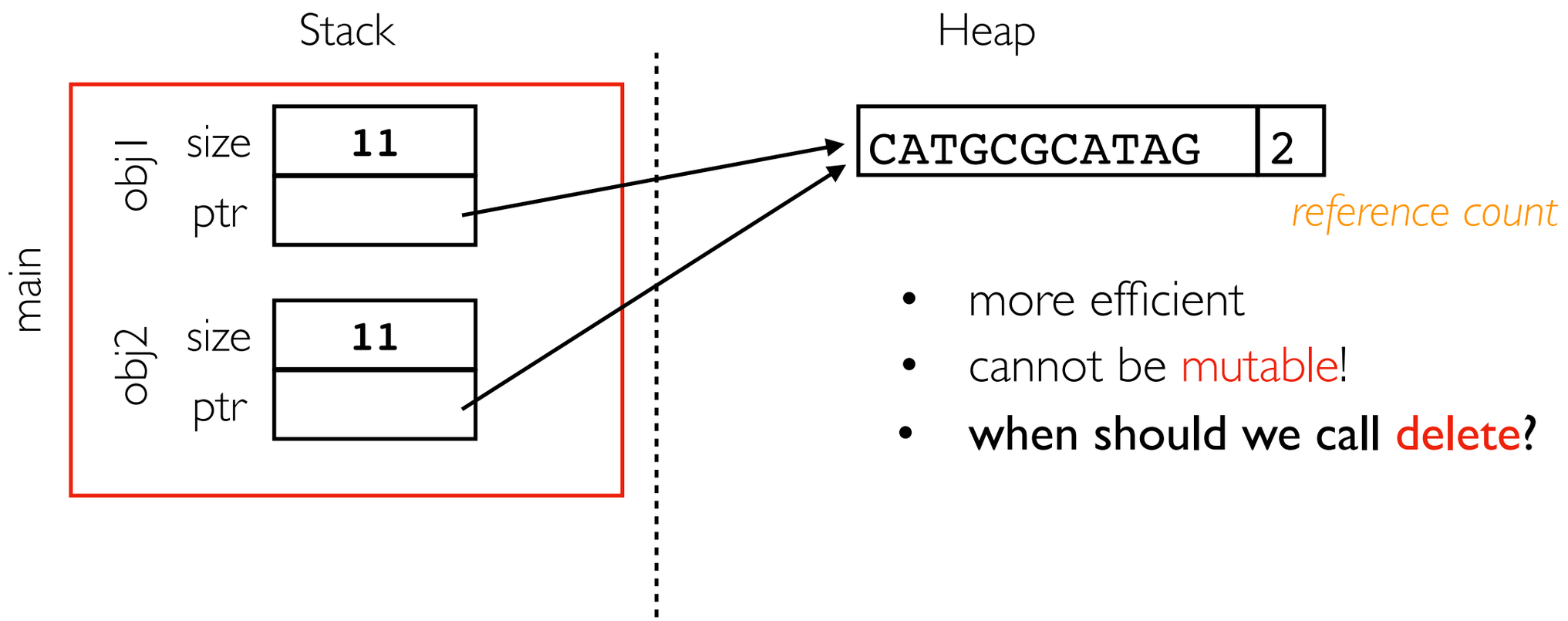
- more efficient
- cannot be mutable!
- **when should we call delete?**

Copy Constructor

```
// CODE:
DNA obj1{"CATGCGCATAG};
DNA obj2 = obj1;
```

implicit call

```
// copy constructor
DNA(const &other) {
    // choose copy behavior
    // here
}
```



- more efficient
- cannot be mutable!
- **when should we call delete?**

Special OOP Functions in C++

```
// CODE:
DNA obj1{"CATGCGCATAG};
DNA obj2 = obj1;
```


```
// copy constructor
DNA(const &other) {
    // choose copy behavior
    // here
}
```

Notes

- most classes have code for some kind of constructor
- in many cases, C++ provides reasonable copy constructors and destructors
- you can choose to "delete" some of the defaults C++ gives you (e.g., creating on "uncopyable" object) or define custom implementations
- it is *usually* incorrect to have a destructor without a copy constructor (and vice versa)
- other special functions (for another day): move constructor, copy assignment operator, move assignment operator

Outline

TopHat and Worksheet

Methods

Encapsulation

Constructors/Destructors

Copy Constructors

Demos