

CSE 333 Section 5

C++ Classes and Dynamic Memory



Logistics

- Homework 2:
 - Due **TODAY @ 11:59pm (5/01)**
- Next exercise out Friday after HW2 deadline, due on Monday
- Midterm: Friday, May 9, in class
 - See [Exams page](#) on the website

Review: Member vs. Non-Member Functions

- A **member function** is a part of the class and can be invoked on the objects of the class
- A **non-member function** is a normal function that happens to use the class
 - Often included in the module that defines the class
- Some functionality *must* be defined one way or the other, but a lot can be defined either way, so let's examine the differences...

Exercise 1



Exercise 1: Member vs Non-Member Comparison

	Member	Non-member
Access to Private Members:	Always	<ul style="list-style-type: none">• Through getters and setters• Through friend keyword (do not use unless needed)
Function call (Func):	obj1.Func(obj2)	Func(obj1, obj2)
Operator call (*):	obj1 * obj2	obj1 * obj2
When preferred:	<ul style="list-style-type: none">• Functions that <i>mutate</i> the object• “Core” class functionality	<ul style="list-style-type: none">• <i>Non-mutating</i> functions• Commutative functions• When the class must be on the right-hand side

The “Big 4” of Classes (Review)

```
class Bar {  
    public:  
        Bar();           // 0-arg ctor  
        Bar(int num);    // 1-arg ctor  
        Bar(const Bar& other); // cctor  
        Bar& operator=(const Bar& other); // op=  
        ~Bar();          // dtor  
        ...  
};
```

Constructors (ctor): Construct a new object (parameters must differ).

Copy Constructor (cctor): Constructs a new object based on another instance. Creates copies for pass-by-value (i.e., non-references) and value return as well as variable declarations.

Assignment Operator (op=): Updates existing object based on another instance.

Destructor (dtor): Cleans up the resources of an object when it falls out of scope or is deleted.

Construction and Destruction Details

Construction:

1. Construct/initialize data members in order of declaration within the class.
 - If data member appears in the **initialization list**, apply the specified initialization, otherwise, default initialize.
2. Execute the constructor body.

Construction and Destruction Details

Construction:

1. Construct/initialize data members in order of declaration within the class.
 - If data member appears in the **initialization list**, apply the specified initialization, otherwise, default initialize.
2. Execute the constructor body.

Destruction:

- When multiple objects fall out of scope simultaneously, they are destructed in the *reverse* order of construction.
 1. Execute the destructor body.
 2. Destruct data members in the *reverse* order of declaration within the class.

Exercise 2



Exercise 2: Foo Bar Ordering

```
class Bar {
public:
    Bar() : num_(0) { }                // 0-arg ctor
    Bar(int num) : num_(num) { }        // 1-arg ctor
    Bar(const Bar& other) : num_(other.num_) { } // cctor
    ~Bar() { }                          // dtor
    Bar& operator=(const Bar& other) = default; // op=
    int get_num() const { return num_; } // getter

private:
    int num_;
};

class Foo {
public:
    Foo() : bar_(5) { }                // 0-arg ctor
    Foo(const Bar& b) { bar_ = b; }      // 1-arg ctor
    ~Foo() { }                          // dtor

private:
    Bar bar_;
};
```

**Given these class declarations,
order the execution of the program
(on the next slide)**

Exercise 2: Foo Bar Ordering

```
int main() {  
    Bar b1(3);  
    Bar b2 = b1;  
    Foo f1;  
    Foo f2(b2);  
    return EXIT_SUCCESS;  
}
```

Method Invocation Order:

1. Bar 1-arg ctor (b1)
2. Bar ctor (b2)
3. Foo 0-arg ctor (f1)
4. ↳ Bar 1-arg ctor
5. Foo 1-arg ctor (f2)
6. ↳ Bar 0-arg ctor
7. ↳ Bar op=
8. Foo dtor (f2)
9. ↳ Bar dtor
10. Foo dtor (f1)
11. ↳ Bar dtor
12. Bar dtor (b2)
13. Bar dtor (b1)

b1

num_ = 3

b2

num_ = 3

f1

bar_(5)

num_ = 5

f2

bar_()

num_ = 3

Design Considerations

- What happens if you don't define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?
 - In C++, if you don't define any of these, one will be synthesized for you
 - The synthesized copy constructor does a shallow copy of all fields
 - The synthesized assignment operator does a shallow copy of all fields
 - The synthesized destructor calls the default destructors of any fields that have them
- How can you disable the copy constructor/assignment operator/destructor?

Set their prototypes equal to the keyword "delete":

SomeClass(const SomeClass&) = delete;

New and Delete Operators

new: Allocates the type on the heap, calling specified constructor if it is a class type

Syntax:

```
type* ptr = new type;
```

```
type* heap_arr = new type[num];
```

delete: Deallocates the type from the heap, calling the destructor if it is a class type.
For anything you called **new** on, you should at some point call **delete** to clean it up

Syntax:

```
delete ptr;
```

```
delete[] heap_arr;
```

Exercise 3



Exercise 3: Memory Leaks

Stack

Heap

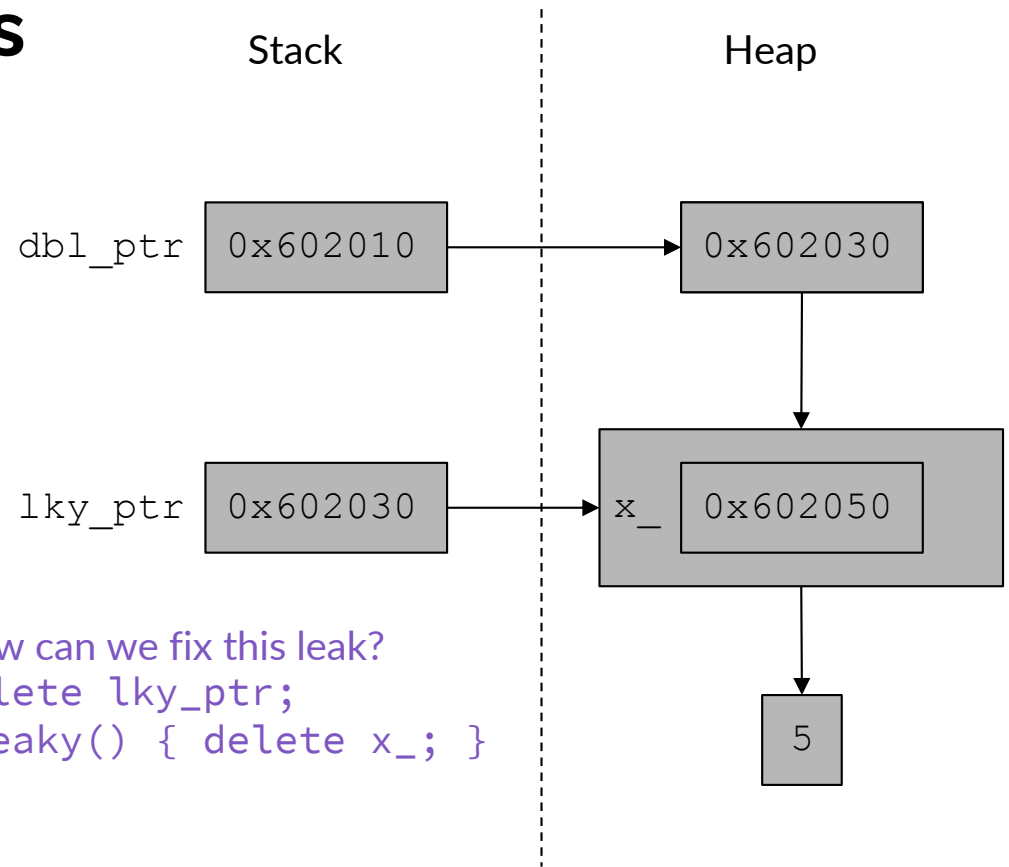
```
class Leaky {  
public:  
    Leaky() { x_ = new int(5); }  
private:  
    int* x_;  
};  
  
int main(int argc, char** argv) {  
    Leaky** dbl_ptr = new Leaky*;  
    Leaky* lky_ptr = new Leaky();  
    *dbl_ptr = lky_ptr;  
    delete dbl_ptr;  
    return EXIT_SUCCESS;  
}
```

Exercise 3: Memory Leaks

```
class Leaky {  
public:  
    Leaky() { x_ = new int(5); }  
private:  
    int* x_;  
};
```

```
int main(int argc, char** argv) {  
➡ Leaky** dbl_ptr = new Leaky*;  
➡ Leaky* lky_ptr = new Leaky();  
➡ *dbl_ptr = lky_ptr;  
➡ delete dbl_ptr;  
➡ return EXIT_SUCCESS;  
}
```

How can we fix this leak?
`delete lky_ptr;`
`~Leaky() { delete x_; }`



An Acronym to Know: RAII

- Stands for “Resource Acquisition Is Initialization”
- Any resources you acquire (locks, files, heap memory, etc.) should happen in a constructor (i.e., during initialization)
- Then freeing those resources should happen in the destructor (and handled properly in cctor, assignment operator, etc.)
- Prevents forgetting to call **free/delete**, the dtor is called automatically for you when the object managing the resource goes out of scope.
- For more: <https://en.cppreference.com/w/cpp/language/raii>

Exercise 4



Exercise 4: Bad Copy

Stack

Heap

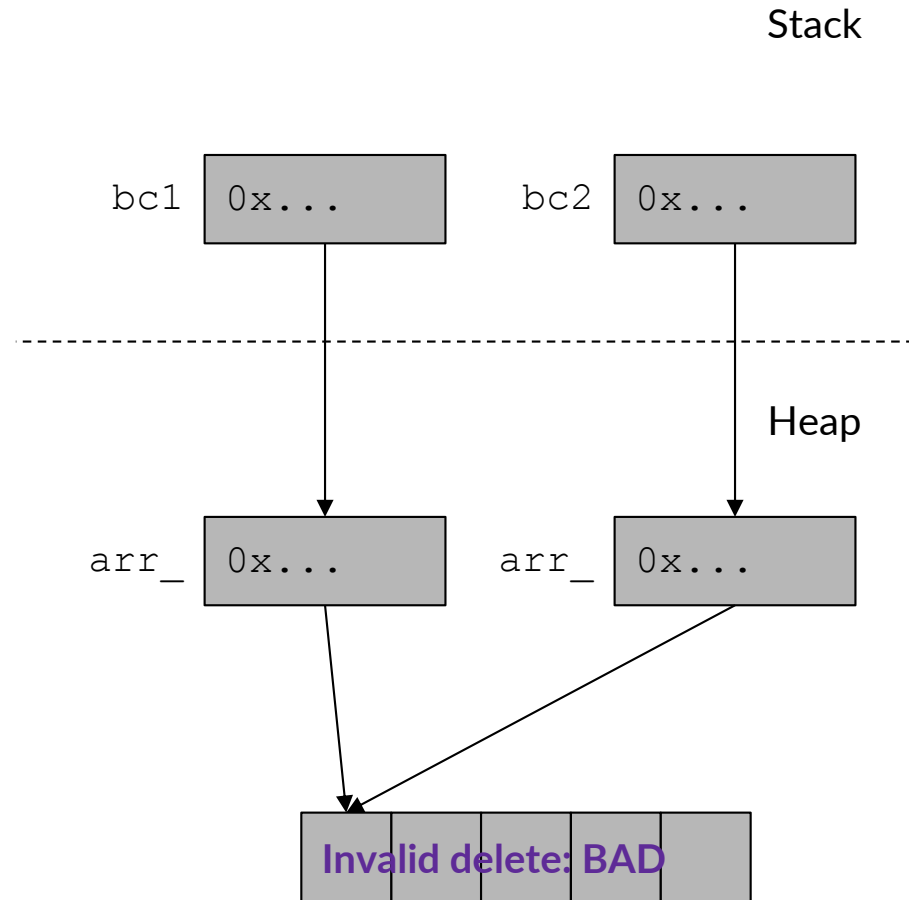
```
class BadCopy {
public:
    BadCopy() { arr_ = new int[5]; }
    ~BadCopy() { delete [] arr_; }
private:
    int* arr_;
};

int main(int argc, char** argv) {
    BadCopy* bc1 = new BadCopy;
    BadCopy* bc2 = new BadCopy(*bc1); // cctor
    delete bc1;
    delete bc2;
    return EXIT_SUCCESS;
}
```

Exercise 4: Bad Copy

```
class BadCopy {  
public:  
    BadCopy() { arr_ = new int[5]; }  
    ~BadCopy() { delete [] arr_; }  
private:  
    int* arr_;  
};
```

```
int main(int argc, char** argv) {  
➡ BadCopy* bc1 = new BadCopy;  
➡ BadCopy* bc2 = new BadCopy(*bc1);  
➡ delete bc1;  
➡ delete bc2;  
➡ return EXIT_SUCCESS; as if!  
}
```



The “Rule of Three”

- If your class needs its own destructor, assignment operator, or copy constructor, it almost certainly needs all three!
- **BadCopy** is a good example why, we need a destructor to **delete arr**, and so we needed a copy constructor too because otherwise we end up with a double **delete**
- **BadCopy** also needs its own assignment operator for the same reason, even with a fixed copy constructor, **b1 = b2;** would still break!
- For more info/examples, see https://en.cppreference.com/w/cpp/language/rule_of_three