Object-oriented Programming

Lecture 8

Destructor

- A class’ destructor is automatically called when an object of that class is “**destroyed”**

- **Destruction** of an object means when program execution leaves the scope in which object was instantiated

Destructor

- Has the same name as that of class
  Example: ~MyClass() { . . . }

- The destructor itself does not release object's memory, it just performs *termination tasks* right before the memory is reclaimed

Destructor

- A destructor cannot return a value and cannot take any arguments

- A destructor cannot be overloaded

- A class can thus have only one destructor

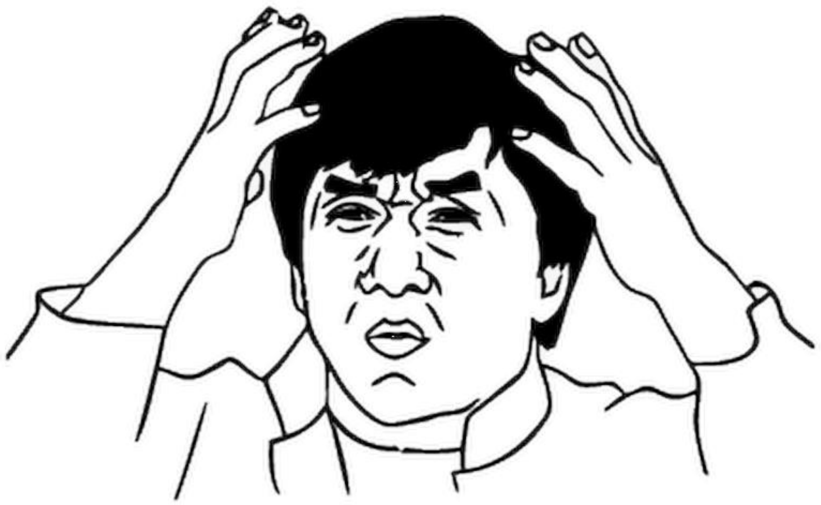
- If you do not explicitly define a destructor, the compiler provides a default “empty” destructor

Order of calling Destructors

```
class MyClass
{
    int objectID;

    MyClass(int objectID)
    {
        this->objectID = objectID;
    }

   ~MyClass()
    {
        cout << objectID << " deleted";
    }
}
```

Order of calling Destructors?

```
MyClass ob1 (1);

void func()
{
    MyClass ob3 (3);
    MyClass ob4 (4);
}

int main()
{
    MyClass ob2 (2);
    func();
    MyClass ob5 (5);
}
```



Order of calling Destructors

```
MyClass ob1 (1);        \\ destroyed fifth

void func()
{
    MyClass ob3 (3);    \\ destroyed second
    MyClass ob4 (4);    \\ destroyed first
}

int main()
{
    MyClass ob2 (2);    \\ destroyed fourth
    func();
    MyClass ob5 (5);    \\ destroyed third
}
```

Why are Destructors useful?

- Useful for garbage collection

- Garbage-collected languages like JAVA do not have a destructor, because:
  - There is no guarantee of when an object will be destroyed