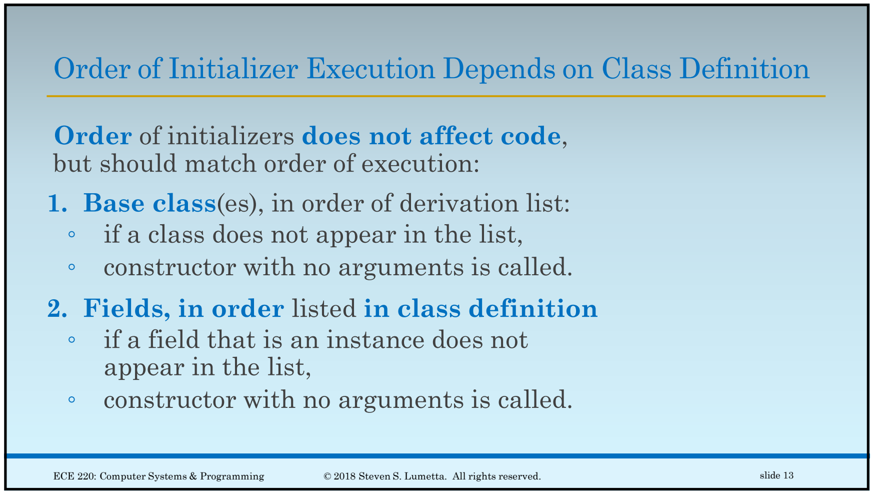
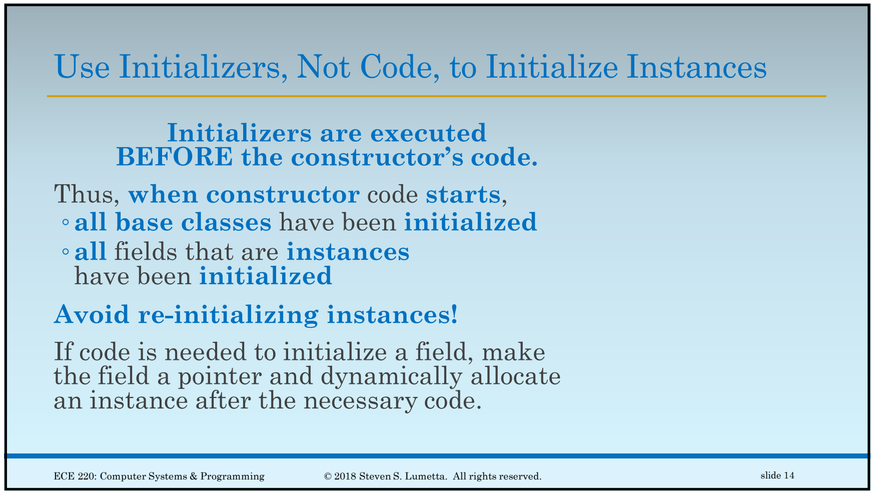


Order of Initializer Execution Depends on Class Definition

Order of initializers **does not affect code**, but should match order of execution:

1. **Base class**(es), in order of derivation list:
   - if a class does not appear in the list,
   - constructor with no arguments is called.
2. **Fields, in order** listed **in class definition**
   - if a field that is an instance does not appear in the list,
   - constructor with no arguments is called.

13



Use Initializers, Not Code, to Initialize Instances

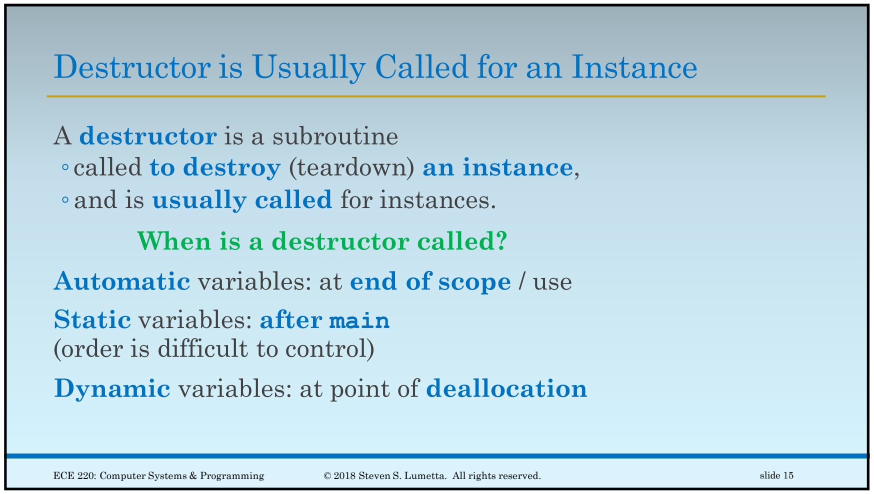
Initializers are executed BEFORE the constructor's code.

Thus, **when constructor** code **starts**,
- **all base classes** have been **initialized**
- **all** fields that are **instances** have been **initialized**

Avoid re-initializing instances!

If code is needed to initialize a field, make the field a pointer and dynamically allocate an instance after the necessary code.

14



Destructor is Usually Called for an Instance

A **destructor** is a subroutine
- called **to destroy** (teardown) **an instance**,
- and is **usually called** for instances.

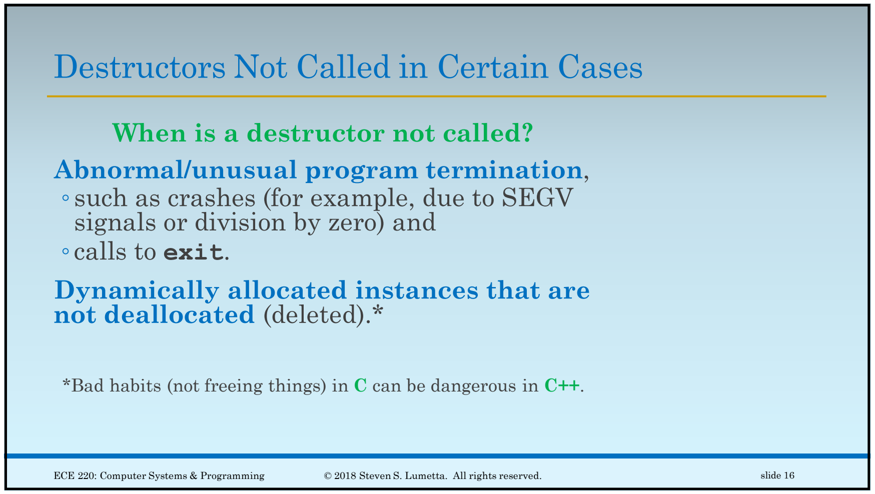
When is a destructor called?

Automatic variables: at **end of scope** / use

Static variables: **after `main`** (order is difficult to control)

Dynamic variables: at point of **deallocation**

15



Destructors Not Called in Certain Cases

When is a destructor not called?

Abnormal/unusual program termination,
- such as crashes (for example, due to SEGV signals or division by zero) and
- calls to **`exit`**.

Dynamically allocated instances that are not deallocated (deleted).*

*Bad habits (not freeing things) in **C** can be dangerous in **C++**.

16