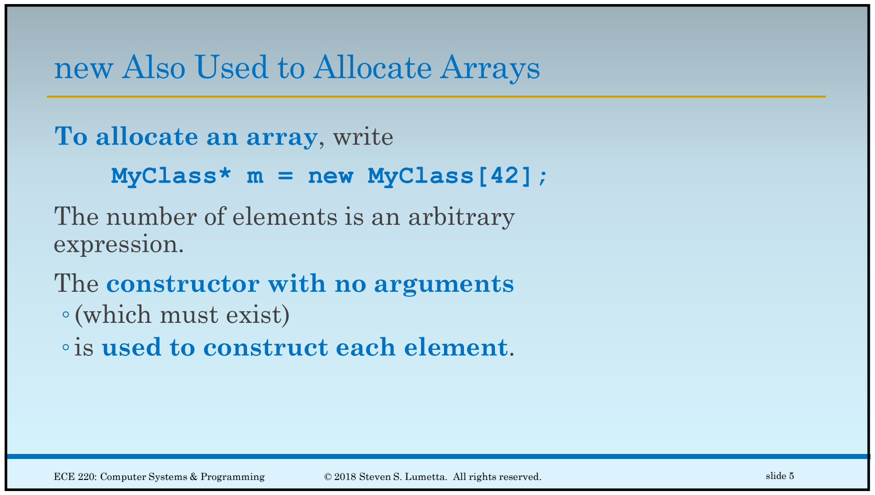


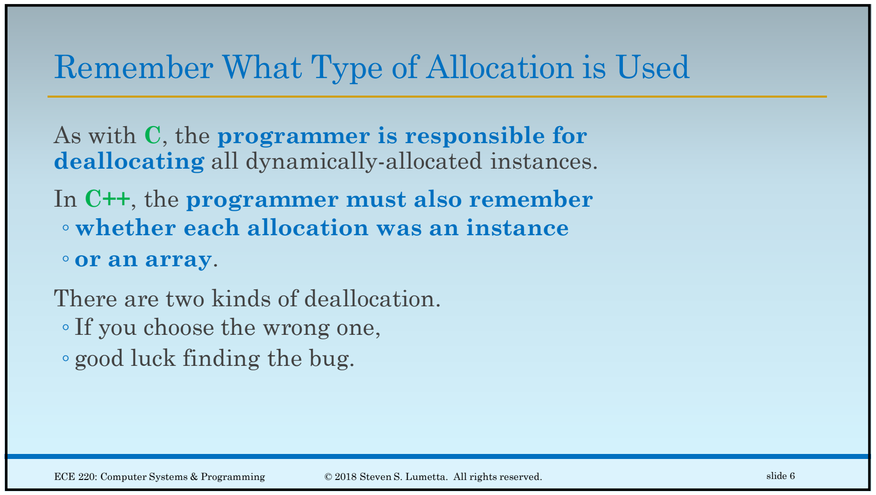
new Also Used to Allocate Arrays

To allocate an array, write

```
MyClass* m = new MyClass[42];
```

The number of elements is an arbitrary expression.

The **constructor with no arguments**
- (which must exist)
- is **used to construct each element**.

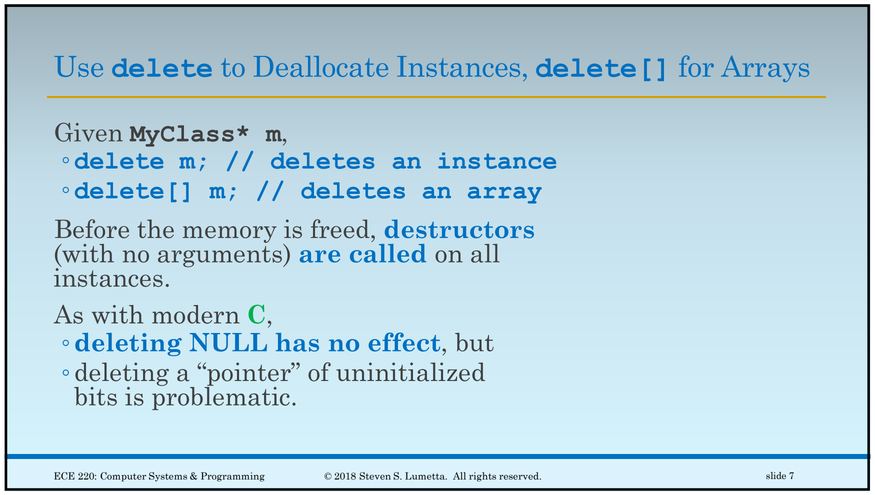


Remember What Type of Allocation is Used

As with **C**, the **programmer is responsible for deallocating** all dynamically-allocated instances.

In **C++**, the **programmer must also remember**
- **whether each allocation was an instance**
- **or an array**.

There are two kinds of deallocation.
- If you choose the wrong one,
- good luck finding the bug.

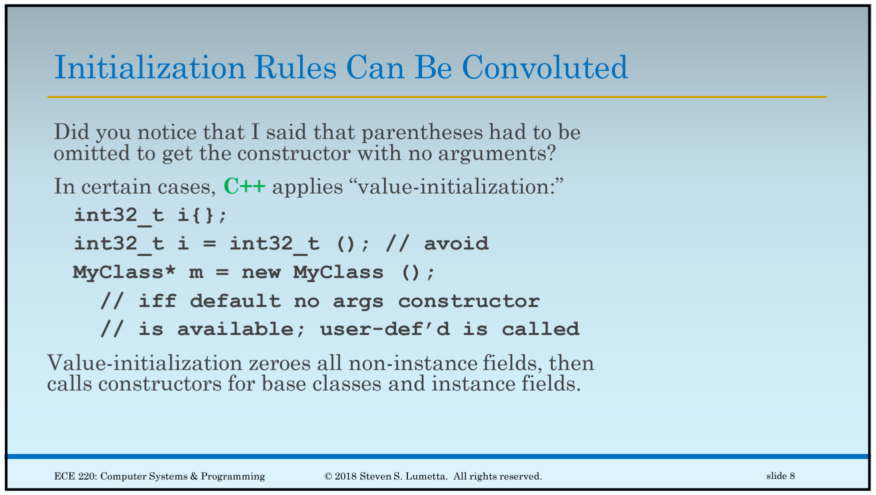


Use `delete` to Deallocate Instances, `delete[]` for Arrays

Given `MyClass* m`,
- `delete m; // deletes an instance`
- `delete[] m; // deletes an array`

Before the memory is freed, **destructors** (with no arguments) **are called** on all instances.

As with modern **C**,
- **deleting NULL has no effect**, but
- deleting a "pointer" of uninitialized bits is problematic.



Initialization Rules Can Be Convoluted

Did you notice that I said that parentheses had to be omitted to get the constructor with no arguments?

In certain cases, **C++** applies "value-initialization:"

```
int32_t i{};
int32_t i = int32_t (); // avoid
MyClass* m = new MyClass ();
  // iff default no args constructor
  // is available; user-def'd is called
```

Value-initialization zeroes all non-instance fields, then calls constructors for base classes and instance fields.