#6: Lifecycle of Classes
February 5, 2021 · G Carl Evans

Using the const keyword
1. Using const in function parameters:

```
joinCubes-by*-const.cpp
15 Cube joinCubes(const Cube &s1, const Cube &s2)
15 Cube joinCubes(const Cube *s1, const Cube *s2)
15 Cube joinCubes(const Cube &s1, const Cube &s2)
```

Best Practice: “All parameters passed by reference must be labeled const.” — Google C++ Style Guide

2. Using const as part of a member functions’ declaration:

```
Cube.h
1 #pragma once
2 namespace cs225 {
3     class Cube {
4         public:
5             Cube();
6             Cube(double length);
7             double getVolume();
8             double getSurfaceArea();
9         private:
10             double length_;
11     }
12 }
```

```
Cube.cpp
11     double Cube::getVolume() { return length_ * length_ * length_; }
14     double Cube::getSurfaceArea() { return 6 * length_ * length_; }
```

Returning from a function
Identical to passing into a function, we also have three choices on how memory is used when returning from a function:

Return by value:

```
Cube joinCubes(const Cube &s1, const Cube &s2)
```

Return by reference:

```
Cube joinCubes(const Cube &s1, const Cube &s2)
...remember: never return a reference to stack memory!
```

Return by pointer:

```
Cube *joinCubes(const Cube &s1, const Cube &s2)
...remember: never return a reference to stack memory!
```

Copy Constructor
When a non-primitive variable is passed/returned by value, a copy must be made. As with a constructor, an automatic copy constructor is provided for you if you choose not to define one:

All copy constructors will:

The automatic copy constructor:

1.

2.

To define a custom copy constructor:

```
cs225/Cube.h
4     class Cube {
5         public:
6             Cube(); // default ctor
7             Cube(double length); // 1-param ctor
8         double getVolume();
9         double getSurfaceArea();
10         private:
11             double length_;
12 }
```

Recall the joinCubes function:

```
joinCubes-(byValue,byReference,byPointer).cpp
15 Cube joinCubes(Cube c1, Cube c2) {
16     double totalVolume = c1.getVolume() + c2.getVolume();
17     double newLength = std::pow( totalVolume, 1.0/3.0 );
18     Cube result(newLength);
19     return result;
22 }
```
### Bringing Concepts Together:

How many times do our different joinCubes files call each constructor?

<table>
<thead>
<tr>
<th></th>
<th>By Value</th>
<th>By Pointer</th>
<th>By Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cube(double)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cube(const Cube &amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Cubes Unite!

Consider a Tower made of three Cubes:

```cpp
#pragma once
#include "cs225/Cube.h"
using cs225::Cube;

class Tower {
public:
    Tower(Cube c, Cube *ptr, const Cube &ref);
    Tower(const Tower & other);

private:
    Cube cube_;  
    Cube *ptr_; 
    const Cube &ref;
};
```

### Automatic Copy Constructor Behavior:

The behavior of the automatic copy constructor is to make a copy of every variable. We can mimic this behavior in our Tower class:

```cpp
Tower::Tower(const Tower & other) {
    cube_ = other.cube_;  
    ptr_ = other.ptr_;    
    ref_ = other.ref_;    
}
```

...we refer to this as a ____________ because:

### Deep Copy via Custom Copy Constructor:

Alternatively, a custom copy constructor can perform a deep copy:

```cpp
Tower::Tower(const Tower & other) {
    // Deep copy cube_:
    cube_ = other.cube_;  
    // Deep copy ptr_:
    ptr_ = other.ptr_;    
    // Deep copy ref_:
    ref_ = other.ref_;    
}
```

### Destructor

The last and final member function called in the lifecycle of a class is the destructor.

#### Purpose of a destructor:

1. ____________

2. ____________

### Custom Destructor:

```cpp
class Cube {
public:
    Cube(); // default ctor
    Cube(double length); // 1-param ctor
    Cube(const Cube & other); // custom copy ctor
    ~Cube(); // destructor, or dtor
...`
```