CS 225

#6: Lifecycle of Classes

February 5, 2021 · *G Carl Evans*

Using the `const` keyword

1. Using `const` in function parameters:

joinCubes-by*-const.cpp

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

```
#pragma once

namespace cs225 {
  class Cube {
    public:
      Cube();
      Cube(double length);
      double getVolume()          ;
      double getSurfaceArea()          ;

    private:
      double length_;
  };
}
```

Cube.cpp

```
  double Cube::getVolume()                    {
    return length_ * length_ * length_;
  }

  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

```
class Cube {
  public:
    Cube();              // default ctor
    Cube(double length); // 1-param ctor


    double getVolume();
    double getSurfaceArea();

  private:
    double length_;
};
```

Recall the joinCubes function:

joinCubes-{byValue,byReference,byPointer}.cpp

```
Cube joinCubes(Cube ___ c1, Cube ___ c2) {
  double totalVolume = c1.getVolume() + c2.getVolume();

  double newLength = std::pow( totalVolume, 1.0/3.0 );

  Cube result(newLength);
  return result;
}
```

Bringing Concepts Together:

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

	By Value	By Pointer	By Reference
`Cube()`			
`Cube(double)`			
`Cube(const Cube &)`			

Cubes Unite!

Consider a Tower made of three Cubes:

Tower.h

```
#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:

Tower.cpp

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

```
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:

Tower.cpp

```
Tower::Tower(const Tower & other) {
  // Deep copy cube_:



  // Deep copy ptr_:



  // Deep copy ref_:


}
```

Destructor

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

Purpose of a **destructor**:

The **automatic destructor**:

1.

2.

Custom Destructor:

cs225/Cube.h

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

CS 225 – Things To Be Doing:

1. lab_intro and lab_debug due Sunday@ 11:59pm
2. Mp_intro is due Monday@11:59pm
3. Daily POTDs every M-F for daily extra credit!