CS 225

Data Structures

January 28 – C++ Copying and Overloading

G Carl Evans

Returning Pointers and References

A variable containing an instance of an object:

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

A reference variable of a Cube object:

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

A variable containing a pointer to a Cube object:

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

Copy Constructor

Copy Constructor

Automatic Copy Constructor

Custom Copy Constructor

Tower.h

```
 1 #pragma once
 2
 3 #include "cs225/Cube.h"
 4 using cs225::Cube;
 5
 6 class Tower {
 7   public:
 8     Tower(Cube c, Cube *ptr, const Cube &ref);
 9     Tower(const Tower & other);
10
11   private:
12     Cube cube_;
13     Cube *ptr_;
14     const Cube &ref;
15 };
16
17
```

Tower.cpp

```
10 Tower::Tower(const Tower & other) {
11   cube_ = other.cube_;
12   ptr_ = other.ptr_;
13   ref_ = other.ref_;
14 }
```

Tower.cpp

```
10 Tower::Tower(const Tower & other) {
11   cube_ = other.cube_;
12   ptr_ = other.ptr_;
13   ref_ = other.ref_;
14 }
```

```
waf@siebl-2215-02:/mnt/c/Users/waf/Desktop/cs225/_lecture/06-lifecycle$ make
clang++ -std=c++1y -stdlib=libc++ -O0 -Wall -Wextra -pedantic -lpthread -lm main.cpp cs225/Cube.cpp Tower.cpp -o main
Tower.cpp:10:8: error: constructor for 'Tower' must explicitly initialize the reference member 'ref_'
Tower::Tower(const Tower & other) {
       ^
./Tower.h:14:17: note: declared here
    const Cube &ref_;
                ^
Tower.cpp:20:8: error: no viable overloaded '='
  ref_ = other.ref_;
  ~~~~ ^ ~~~~~~~~~~
```

Tower.cpp

```
10 Tower::Tower(const Tower & other) {
11   cube_ = other.cube_;
12   ptr_ = other.ptr_;
13   ref_ = other.ref_;
14 }
```

Tower.cpp

```
10 Tower::Tower(const Tower & other) : cube_(other.cube_),
11    ptr_(other.ptr_), ref_(other.ref_) { }
12
13
14
```

Constructor Initializer List

Tower.cpp

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




  // Deep copy ptr_:




  // Deep copy ref_:




}
```

Destructor

Cube.h

```
 1  #pragma once
 2
 3  namespace cs225 {
 4    class Cube {
 5      public:
 6        Cube();
 7        Cube(double length);
 8        Cube(const Cube & other);
 9        ~Cube();
10
11        double getVolume() const;
12        double getSurfaceArea() const;
13
14      private:
15        double length_;
16    };
17  }
18
19
20
```

Cube.cpp

```
 7  namespace cs225 {
 8    Cube::Cube() {
 9      length_ = 1;
10      cout << "Default ctor"
             << endl;
11    }
12
13    Cube::Cube(double length) {
14      length_ = length;
15      cout << "1-arg ctor"
             << endl;
16    }
17
18
19
20
21
22
23
24
25
…     // ...
```

Operators that can be overloaded in C++	
Arithmetic	+ - * / % ++ --
Bitwise	& | ^ ~ << >>
Assignment	=
Comparison	== != > < >= <=
Logical	! && ||
Other	[] () ->

Cube.h

```
 1  #pragma once
 2
 3  namespace cs225 {
 4    class Cube {
 5      public:
 6        Cube();
 7        Cube(double length);
 8        Cube(const Cube & other);
 9        ~Cube();
10
11
12
13
14
15        double getVolume() const;
16        double getSurfaceArea() const;
17
18      private:
19        double length_;
20    };
    }
```

Cube.cpp

```
 7  namespace cs225 {
 8    Cube::~Cube() {
 9      cout << "dtor called";
           << endl;
10    }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
…     // ...
```

One Very Special Operator

Definition Syntax (.h):

```
Cube & operator=(const Cube& s)
```

Implementation Syntax (.cpp):

```
Cube & Cube::operator=(const Cube& s)
```

Assignment Operator

Similar to Copy Constructor:

Different from Copy Constructor:

Example:

assignmentOpSelf.cpp

```
1  #include "Cube.h"
2
3  int main() {
4    cs225::Cube c(10);
5    c = c;
6    return 0;
7  }
```

Example:

assignmentOpSelf.cpp

```
 1 #include "Cube.h"
 …
40 Cube& Cube::operator=(const Cube &other) {
41
42   _destroy();
43   _copy(other);
44
45   return *this;
46 }
```

Assignment Operator

	Copies an object	Destroys an object
Copy constructor		
Copy Assignment operator		
Destructor		

The “Rule of Three”

If it is necessary to define any one of these three functions in a class, it will be necessary to define all three of these functions:

1.

2.

3.

The “Rule of Zero”

Corollary to Rule of Five

Classes that **declare** custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership. Other classes should not **declare** custom destructors, copy/move constructors or copy/move assignment operators

–Scott Meyers