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

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

    // Deep copy ptr_

    // Deep copy ref_
}
Destructor
```cpp
#pragma once

namespace cs225 {
    class Cube {
    public:
        Cube();
        Cube(double length);
        Cube(const Cube & other);
        ~Cube();
        double getVolume() const;
        double getSurfaceArea() const;
    private:
        double length_;  // Other private members can be added here.
    }
}

namespace cs225 {
    Cube::Cube() {
        length_ = 1;
        cout << "Default ctor" << endl;
    }
    Cube::Cube(double length) {
        length_ = length;
        cout << "1-arg ctor" << endl;
    }
    ... // Other implementations
```
## Operators that can be overloaded in C++

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+  -  *  /  %  ++  --</td>
</tr>
<tr>
<td>Bitwise</td>
<td>&amp;</td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
</tr>
<tr>
<td>Comparison</td>
<td>==  ! =  &gt;  &lt;  &gt;=  &lt;=</td>
</tr>
<tr>
<td>Logical</td>
<td>!  &amp;&amp;</td>
</tr>
<tr>
<td>Other</td>
<td>[ ]  ()  -&gt;</td>
</tr>
</tbody>
</table>
```cpp
#pragma once

namespace cs225 {

    class Cube {
        public:
            Cube();
            Cube(double length);
            Cube(const Cube & other);
            ~Cube();

            double getVolume() const;
            double getSurfaceArea() const;

        private:
            double length_;  
    };
}

namespace cs225 {

    Cube::~Cube() {
        cout << "dtor called";
        cout << endl;
    }
}
```
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:

```cpp
#include "Cube.h"

int main() {
    cs225::Cube c(10);
    c = c;
    return 0;
}
```
```cpp
#include "Cube.h"
...
Cube& Cube::operator=(const Cube &other) {
    _destroy();
    _copy(other);
    return *this;
}
```
# Assignment Operator

<table>
<thead>
<tr>
<th></th>
<th>Copies an object</th>
<th>Destroys an object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy constructor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy Assignment operator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destructor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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