CS 225
Data Structures

September 8– Overloading
G Carl Evans
```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_;  // Your description here
    Cube *ptr_;  // Your description here
    const Cube &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) {
    cube_ = other.cube_;  
    ptr_ = other.ptr_;  
    ref_ = other.ref_;  
}
Tower::Tower(const Tower & other) {
    // Deep copy cube_:

    // Deep copy ptr_:

    // Deep copy ref_:
}

Tower.cpp
Destructor

[Purpose]:
Destructor

[Purpose]: Free any resources maintained by the class.

Automatic Destructor:
1. Exists only when no custom destructor is defined.

2. [Invoked]:

3. [Functionality]:
```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_;    
};
}
```

```cpp
namespace cs225 {

Cube::Cube() {
  length_ = 1;
  cout << "Default ctor" << endl;
}

Cube::Cube(double length) {
  length_ = length;
  cout << "1-arg ctor" << endl;
}
```

```cpp
... // ...
```
### 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_;  
    };
}
```
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:
# 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
In CS 225
Rvalue Reference or Move Semantics

• Rvalue

• Move

\[ \text{Cube(const Cube&& s) noexcept} \]

• Move Assignment

\[ \text{Cube & operator=(const Cube&& s) noexcept} \]
The “Rule of Five”

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

1.

2.

3.

4.

5.