CS 225
Data Structures

January 31 – C++ Overloading and Inheritance
G Carl Evans
<table>
<thead>
<tr>
<th>Operators that can be overloaded in C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
</tr>
<tr>
<td>+  -  *  /  %  ++  --</td>
</tr>
<tr>
<td><strong>Bitwise</strong></td>
</tr>
<tr>
<td>&amp;</td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
</tr>
<tr>
<td>=</td>
</tr>
<tr>
<td><strong>Comparison</strong></td>
</tr>
<tr>
<td>==  !=  &gt;  &lt;  &gt;=  &lt;=</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
</tr>
<tr>
<td>!  &amp;&amp;</td>
</tr>
<tr>
<td><strong>Other</strong></td>
</tr>
<tr>
<td>[ ] ( ) -&gt;</td>
</tr>
</tbody>
</table>
#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_;  // length of the cube
};
}

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() {
    cs252::Cube c(10);
    c = c;
    return 0;
}
```
Example:

```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
In CS 225
Rvalue Reference or Move Semantics

• Rvalue

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

• Move Assignment
  \[
  \text{Cube} \ & \ \text{operator= (const Cube} \&\& \ s) \text{ noexcept}
  \]
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.
Inheritance
#pragma once

#include "Shape.h"

class Square : public Shape {
    public:
        double getArea() const;
    
    private:
        // Nothing!
        // Nothing!
};

class Shape {
    public:
        Shape();
        Shape(double length);
        double getLength() const;
    
    private:
        double length_;
Derived Classes

[Public Members of the Base Class]:

```
5  int main() {
6    Square sq;
7    sq.getLength(); // Returns 1, the length init’d
8    // by Shape’s default ctor
...  ...
... }
```

[Private Members of the Base Class]:

Polymorphism

The idea that a single interface may take multiple types or that a single symbol may be different types.

In Object-Orientated Programming (OOP) a key example is that a single object may take on the type of any of its base types.
Virtual
Cube.cpp

1 Cube::print_1() {
2   cout << "Cube" << endl;
3 }

4 Cube::print_2() {
5   cout << "Cube" << endl;
6 }

7 virtual Cube::print_3() {
8   cout << "Cube" << endl;
9 }

10 virtual Cube::print_4() {
11   cout << "Cube" << endl;
12 }

13 // In .h file:
14 virtual print_5() = 0;
15
16
17
RubikCube.cpp

1 // No print_1() in RubikCube.cpp
2
3
4

5 RubikCube::print_2() {
6   cout << "Rubik" << endl;
7 }

8

9 // No print_3() in RubikCube.cpp
10
11
12

13 RubikCube::print_4() {
14   cout << "Rubik" << endl;
15 }

16

17 RubikCube::print_5() {
18   cout << "Rubik" << endl;
19 }

20
21
22
## Runtime of Virtual Functions

<table>
<thead>
<tr>
<th>virtual-main.cpp</th>
<th>Cube c;</th>
<th>RubikCube c;</th>
<th>RubikCube rc; Cube &amp;c = rc;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c.print_1();</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c.print_2();</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c.print_3();</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c.print_4();</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c.print_5();</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why Polymorphism?
```cpp
class Animal {
  public:
    void speak() {}
};

class Dog : public Animal {
  public:
    void speak() {}
};

class Cat : public Animal {
  public:
    void speak() {}
};
```
Abstract Class:

[Requirement]:

[Syntax]:

[As a result]:
virtual-dtor.cpp

```cpp
15 class Cube {
16     public:
17         ~Cube();
18     };
19
20 class RubikCube : public Cube {
21     public:
22         ~RubikCube();
23     };
```