One Very Powerful Operator: Assignment Operator

<table>
<thead>
<tr>
<th>Cube.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube &amp; operator=(const Cube &amp; other);</td>
</tr>
<tr>
<td>Cube.cpp</td>
</tr>
<tr>
<td>Cube &amp; Cube::operator=(const Cube &amp; other) { ... }</td>
</tr>
</tbody>
</table>

Functionality Table:

<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</td>
<td></td>
<td></td>
</tr>
<tr>
<td>operator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destructor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assignment Operator – Self Destruction

- Programmers are sometimes not perfect. Consider the following:

```
#include "Cube.h"

int main() {
    cs225::Cube c(10);
    c = c;
    return 0;
}
```

- Ensure your assignment operator doesn’t self-destroy:

```
#include "Cube.h"

if (&other != this) {
    _destroy();
    _copy(other);
}
return *this;
```

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

Inheritance

In nearly all object-oriented languages (including C++), classes can be extended to build other classes. We call the class being extended the **base class** and the class inheriting the functionality the **derived class**.

```
class Shape {
    public:
        Shape();
        Shape(double length);
        double getLength() const;
    private:
        double length_;
};
```

```
#include "Shape.h"

class Square : public Shape {
    public:
        double getArea() const;
    private:
        // Nothing!
};
```
In the above code, `Square` is derived from the base class `Shape`:

- All **public** functionality of `Shape` is part of `Square`:

```cpp
main.cpp
int main() {
    Square sq; // Returns 1, the len init’d by Shape’s default ctor
    ...}
```

- [Private Members of `Shape`]:

Virtual
- The **virtual** keyword allows us to override the behavior of a class by its derived type.

**Example:**

```cpp
Cube.cpp
virtual Cube::print_3() {
    cout << "Cube" << endl;
} // No print_3()
```

```cpp
RubikCube.cpp
virtual RubikCube::print_3() {
    cout << "Rubik" << endl;
} // No print_3()
```

```cpp
virtual RubikCube::print_4() {
    cout << "Rubik" << endl;
} // No print_4()
```

```cpp
virtual RubikCube::print_5() {
    cout << "Rubik" << endl;
} // No print_5()
```

```cpp
// In .h file:
virtual print_5() = 0;
```

**Polymorphism**
Object-Orientated Programming (OOP) concept that a single object may take on the type of any of its base types.

- A **RubikCube** may polymorph itself to a Cube
- A Cube cannot polymorph to be a **RubikCube** (base types only)

**Why Polymorphism?** Suppose you’re managing an animal shelter that adopts cats and dogs:

**Option 1 – No Inheritance**

```cpp
animalShelter.cpp
Cat & AnimalShelter::adopt() { ... }
Dog & AnimalShelter::adopt() { ... }
```

**Option 2 – Inheritance**

```cpp
animalShelter.cpp
Animal & AnimalShelter::adopt() { ... }
```

**Pure Virtual Methods**
In `Cube`, `print_5()` is a **pure virtual** method:

```cpp
Cube.h
virtual Cube::print_5() = 0;
```

A pure virtual method does not have a definition and makes the class and abstract class.

**CS 225 – Things To Be Doing:**

1. mp_stickers EC deadline Sep. 13 (12 days).
2. Lab Extra Credit → Lab attendance is automatic this week.
3. Daily POTDs