#8: Templates
February 1, 2018 · Fagen-Ulmschneider, Zilles

Assignment Operator – Self Destruction
- Programmers are sometimes not perfect. Consider the following:

```cpp
#include "Cube.h"
int main() {
    cs225::Cube c(10);
    c = c;
    return 0;
}
```

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

```cpp
#include "Cube.h"
Cube& Cube::operator=(const Cube &other) {
    if (&other != this) {
        _destroy();
        _copy(other);
    }
    return *this;
}
```

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**.

```cpp
#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;
    sq.getLength(); // 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:**

<table>
<thead>
<tr>
<th>Cube.cpp</th>
<th>RubikCube.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube::print_1() {</td>
<td>Cube::print_1() {</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>Cube::print_2() {</td>
<td>Cube::print_2() {</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
<td>cout &lt;&lt; &quot;Rubik&quot; &lt;&lt; endl;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>virtual Cube::print_3() {</td>
<td>virtual Cube::print_3() {</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>virtual Cube::print_4() {</td>
<td>virtual Cube::print_4() {</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
<td>cout &lt;&lt; &quot;Rubik&quot; &lt;&lt; endl;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>virtual Cube::print_5() {</td>
<td>virtual Cube::print_5() {</td>
</tr>
<tr>
<td>cout &lt;&lt; &quot;Cube&quot; &lt;&lt; endl;</td>
<td>cout &lt;&lt; &quot;Rubik&quot; &lt;&lt; endl;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shape.h</th>
<th>Square.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Shape {</td>
<td>class Square : public Shape {</td>
</tr>
</tbody>
</table>
|    public:               |    public:
|        Shape();         |        double getArea() const;
|        Shape(double length); double getLength() const; |
|    private:             |    private:
|        double length_;  |        // Nothing!
| };                      | };
```

```cpp
RubikCube::print_2() {   cout << "Rubik" << endl; }
RubikCube::print_3() {   cout << "Rubik" << endl; }
RubikCube::print_4() {   cout << "Rubik" << endl; }
RubikCube::print_5() {   cout << "Rubik" << endl; }
```
**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
1 Cat & AnimalShelter::adopt() { ... }
2 Dog & AnimalShelter::adopt() { ... }
```

**Option 2 – Inheritance**

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

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

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

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

**Abstract Class:**

1. [Requirement]:
2. [Syntax]:
3. [As a result]:

---

**Abstract Class Animal**
In our animal shelter, Animal is an abstract class:

---

**Abstract Data Types (ADT):**

<table>
<thead>
<tr>
<th>List ADT - Purpose</th>
<th>Function Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**List Implementation**
What types of List do we want?

---

**Templates in C++**
Two key ideas when using templates in C++:

1. 
2. 

**Templated Functions:**

```cpp
functionTemplate1.cpp
1 T maximum(T a, T b) {
2     T result;
3     result = (a > b) ? a : b;
4     return result;
5 }
```

---

**CS 225 – Things To Be Doing:**

1. Theory Exam #1 is ongoing; ensure you take it!
2. MP2 due Feb. 11 (10 days), EC deadline in 3 days!
3. Lab Extra Credit → Attendance in your registered lab section!
4. Daily POTDs