

Assignment Operator – Self Destruction

- Programmers are sometimes not perfect Consider the following:

```

assignmentOpSelf.cpp
1 #include "Cube.h"
2
3 int main() {
4     cs225::Cube c(10);
5     c = c;
6     return 0;
7 }
    
```

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

```

Cube.cpp
1 #include "Cube.h"
...
40 Cube& Cube::operator=(const Cube &other) {
41     if (&other != this) {
42         _destroy();
43         _copy(other);
44     }
45     return *this;
46 }
    
```

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**.

Shape.h	Square.h
<pre> class Shape { public: Shape(); Shape(double length); double getLength() const; private: double length_; }; </pre>	<pre> #include "Shape.h" class Square : public Shape { public: double getArea() const; private: // Nothing! }; </pre>

In the above code, **square** is derived from the base class **Shape**:

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

```

main.cpp
5 int main() {
6     Square sq;
7     sq.getLength(); // Returns 1, the len init'd
8                     // 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:

Cube.cpp	RubikCube.cpp
<pre> Cube::print_1() { cout << "Cube" << endl; } Cube::print_2() { cout << "Cube" << endl; } virtual Cube::print_3() { cout << "Cube" << endl; } virtual Cube::print_4() { cout << "Cube" << endl; } // In .h file: virtual print_5() = 0; </pre>	<pre> // No print_1() RubikCube::print_2() { cout << "Rubik" << endl; } // No print_3() RubikCube::print_4() { cout << "Rubik" << endl; } RubikCube::print_5() { cout << "Rubik" << endl; } </pre>

	Cube c;	RubikCube c;	RubikCube rc; Cube &c = rc;
c.print_1();			
c.print_2();			
c.print_3();			
c.print_4();			
c.print_5();			

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

animalShelter.cpp	
1	Cat & AnimalShelter::adopt() { ... }
2	Dog & AnimalShelter::adopt() { ... }
3	...

Option 2 – Inheritance

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

Pure Virtual Methods

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

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):

List ADT - Purpose	Function Definition

List Implementation

What types of List do we want?

Templates in C++

Two key ideas when using templates in C++:

- 1.
- 2.

Templated Functions:

functionTemplate1.cpp	
1	
2	T maximum(T a, T b) {
3	T result;
4	result = (a > b) ? a : b;
5	return result;
6	}

CS 225 – Things To Be Doing:

1. mp_stickers due Sept. 21 (10 days), EC deadline today!
2. Lab Extra Credit → Attendance in your assigned lab section!
3. Daily POTDs