

#6: C++ Overloading and Inheritance

2 5 January 31, 2022 · *G* Carl Evans

Overloading Operators

C++ allows custom behaviors to be defined on over 20 operators:

Arithmetic	+ - * / % ++
Bitwise	& ^ ~ << >>
Assignment	=
Comparison	== != > < >= <=
Logical	! &&
Other	[] () ->

General Syntax:

Adding overloaded operators to Cube:

	Cube.h	Cube.cpp		
1	#pragma once	/* */		
2		40		
3	class Cube {	41		
4	public:	42		
	//	43		
10		44		
11		45		
12		46		
13		47		
14		48		
	//	/* */		

One Very Powerful Operator: Assignment Operator

Cube.h			
	Cube & operator=(const Cube & other);		
Cube.cpp			
Cube & Cube::operator=(const Cube & other) { }			

Functionality Table:

	Copies an object	Destroys an object
Copy constructor		
Copy Assignment operator		
Destructor		

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

CS 225 and Rule Three/Five/Zero In CS 225 We will:

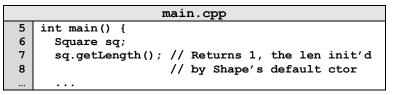
Inheritance

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

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

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



• [Private Members of Shape]:

Virtual

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

Example:

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

	Cube c;	RubikCube c;	RubikCube rc; Cube &c = rc;
	00000 07	nabinease e,	0000000 207
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 can<u>not</u> 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	<pre>Dog & AnimalShelter::adopt() { }</pre>
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	<pre>virtual Cube::print_5() = 0;</pre>

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 due next Monday
- **2.** lab_intro extended deadline Sunday
- 3. new lab released this week also due Sunday
- 4. Daily POTDs