CS 225

Data Structures

January 31 – C++ Overloading and Inheritance

G Carl Evans

Operators that can be overloaded in C++	
Arithmetic	+ - * / % ++ --
Bitwise	& | ^ ~ << >>
Assignment	=
Comparison	== != > < >= <=
Logical	! && ||
Other	[] () ->

Cube.h

```cpp
#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_;
  };
}
```

Cube.cpp

```cpp
namespace cs225 {
  Cube::~Cube() {
    cout << "dtor called";
         << 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:

assignmentOpSelf.cpp

```
#include "Cube.h"

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

Example:

assignmentOpSelf.cpp

```
#include "Cube.h"

Cube& Cube::operator=(const Cube &other) {

  _destroy();
  _copy(other);

  return *this;
}
```

Assignment Operator

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

The "Rule of Three"

If it is <u>necessary to define any one</u> of these three functions in a class, it will be <u>necessary to define all three</u> 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**

```
Cube(const Cube&& s)noexcept
```

- **Move Assignment**

```
Cube & operator=(const Cube&& s)noexcept
```

The “Rule of Five”

If it is <u>necessary to define any one</u> of these five functions in a class, it will be <u>necessary to define all five</u> of these functions:

1.

2.

3.

4.

5.

Inheritance

Square.h

```cpp
#pragma once

#include "Shape.h"

class Square : public Shape {
  public:
    double getArea() const;

  private:
    // Nothing!
};
```

Shape.h

```cpp
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;

  private:
    double length_;
};
```

Square.cpp

...

Derived Classes

[Public Members of the Base Class]:

main.cpp

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

[Private Members of the Base Class]:

Polymorphism

The idea that a single interface my 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

```cpp
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;
```

RubikCube.cpp

```cpp
// No print_1() in RubikCube.cpp



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

// No print_3() in RubikCube.cpp



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

RubikCube::print_5() {
  cout << "Rubik" << endl;
}
```

Runtime of Virtual Functions

virtual-main.cpp	`Cube c;`	`RubikCube c;`	`RubikCube rc;` `Cube &c = rc;`
`c.print_1();`			
`c.print_2();`			
`c.print_3();`			
`c.print_4();`			
`c.print_5();`			

Why Polymorphism?

animalShelter.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

```
class Cube {
  public:
    ~Cube();
};

class RubikCube : public Cube {
  public:
    ~RubikCube();
};
```