

## **Lecture 5: Object Orientation**

Curtin FIRST Robotics Club (FRC) Pre-season Training

---

Scott Day

265815F@curtin.edu.au

December 3, 2016

Curtin University

# Table of contents

1. Object Orientation
2. Classes and Objects
3. Inheritance
4. Overloading
5. Polymorphism
6. Abstraction
7. Encapsulation
8. Interfaces

## Object Orientation

---

Core of Object Oriented Programming (OOP) is to create objects, in code, that have certain properties and methods.

While designing C++ modules, we try to see the whole world in the form of objects.

For example, a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few principle concepts that form the foundation of OOP:

**Object** Is the basic unit of OOP. Both data and function that operate on data are bundled as a unit called an **Object**.

**Class** Defines the blueprint for an object.

**Abstraction** refers to, providing only essential information to the outside world and hiding their background details.

**Encapsulation** is placing data and functions that work on that data in the same place.

**Inheritance** is the process off forming a new class from an existing class that is from the existing class called a base class. The new class formed is called the derived class.

**Polymorphism** is the ability to use a function in different ways.

**Overloading** is also a branch of polymorphism. It allows you to specify more than one definition for a **function name**.

## Classes and Objects

---

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. Then a semicolon or a list of declarations.

```
1 class Box
2 {
3     public:
4         double length; // Length of a box
5         double breadth; // Breadth of a box
6         double height; // Height of a box
7 }
```

The keyword **public** determines the access attributes of the members of the class that follow it. You can also specify it as either **private** or **protected**.

We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

```
1 Box box1;  
2 Box box2;
```



# Accessing Data Members

Public data members of objects of a class can be accessed using the direct member access operator (.).

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Box {
6     public:
7         double length; // Length of a box
8         double breadth; // Breadth of a box
9         double height; // Height of a box
10 };
11
12 int main() {
13     Box box1;           // Declare Box1 of type Box
14     Box box2;           // Declare Box2 of type Box
15     double volume = 0.0; // Store the volume of a box
16     ↪ here
17
18     // box 1 specification
19     box1.height = 5.0;
20     box1.length = 6.0;
21     box1.breadth = 7.0;
22
23     // box 2 specification
24     box2.height = 10.0;
25     box2.length = 12.0;
26     box2.breadth = 13.0;
27
28     // volume of box 1
29     volume = box1.height * box1.length *
30     ↪ box1.breadth;
31     cout << "Volume of Box1 : " << volume << endl;
32
33     // volume of box 2
34     volume = box2.height * box2.length *
35     ↪ box2.breadth;
36     cout << "Volume of Box2 : " << volume << endl;
37
38     return 0;
39 }
```

So far, we have covered the very basics of C++ Classes and Objects. Here are some further concepts we will discuss at some point.

<b>Class Member Functions</b>	Functions with it's definition/prototype within the class definition like any other variable.
<b>Class Access Modifiers</b>	Class members defined as public, private or protected.
<b>Constructor</b>	Special function in a class that's called when a new object of the class is created.
<b>Destructor</b>	Special function which is called when a created object is deleted.
<b>Copy Constructor</b>	Creates an object by initializing it with an object of the same class, which has been created previously.

### **Friend Functions**

Function that is permitted full access to private and protected members of a class.

### **Inline Functions**

The compiler tries to expand the code in the body of the function in place of a call to the function.

### **this pointer in C++**

Every object has a special pointer **this** which points to the object itself.

### **Pointer to C++ Classes**

A pointer to a class done in the same way a pointer to a structure is.

### **Static Members of a Class**

Both data and function members of a class can be declared as static.

## Inheritance

---

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship.

For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

## Blarg blarg blarg

```
1 #include <iostream>
2 using namespace std;
3
4 // Base class
5 class Shape
6 {
7     public:
8         void setWidth(int width)
9         {
10             this.width = width;
11         }
12
13         void setHeight(int height)
14         {
15             this.height = height;
16         }
17
18     protected:
19         int width, height;
20 };
21
22 // Derived class
23 class Rectangle: public Shape
24 {
25     public:
26         int getArea()
27         {
28             return width * height;
29         }
30 };
```

```
1 int main()
2 {
3     Rectangle rect;
4
5     rect.setWidth(5);
6     rect.setHeight(7);
7
8     // Print the area of the object.
9     cout << "Total area: " << rect.getArea() <<
10     "\n";
11
12     return 0;
13 }
```

Blarg blarg blarg the result is 35

A derived class can access all the non-private members of its base class.

Access	Public	Protected	Private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

t



# Multiple Inheritance

A C++ class can inherit members from more than one class.

```
1 #include <iostream>
2 using namespace std;
3
4 // Base class Shape
5 class Shape
6 {
7     public:
8         void setWidth(int width)
9         {
10             this.width = width;
11         }
12
13         void setHeight(int height)
14         {
15             this.height = height;
16         }
17
18     protected:
19         int width, height;
20 };
21
22 // Base class PaintCost
23 class PaintCost
24 {
25     public:
26         int getCost(int area)
27         {
28             return area * 70;
29         }
30 };
31
32 // Derived class
33 class Rectangle: public Shape, public PaintCost
34 {
35     public:
36         int getArea()
37         {
38             return width * height;
39         }
40 };
41
42 int main()
43 {
44     Rectangle rect;
45     int area;
46
47     rect.setWidth(5);
48     rect.setHeight(7);
49
50     area = rect.getArea();
51
52     // Print the area of the object.
53     cout << "Total area: " << rect.getArea() <<
54         << endl;
55
56     // Print the total cost of painting
57     cout << "Total paint cost: $" <<
58         << rect.getCost(area) << endl;
59
60     return 0;
61 }
```

## Overloading

---

## C++ Overloading (Operator and Function)

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

# Function Overloading in C++

```
1 #include <iostream>
2 using namespace std;
3
4 class PrintData
5 {
6     public:
7         void print(int i)
8         {
9             cout << "Printing int: " << i << endl;
10        }
11
12        void print(double f)
13        {
14            cout << "Printing float: " << f << endl;
15        }
16
17        void print(char* c)
18        {
19            cout << "Printing character: " << c <<
20                ↵ endl;
21        }
22    };
23
```

```
1 int main()
2 {
3     PrintData printdata;
4
5     // Call print to print integer
6     printdata.print(5);
7
8     // Call print to print float
9     printdata.print(500.263);
10
11    // Call print to print character
12    printdata.print("Hello C++");
13
14    return 0;
15 }
```

## Polymorphism

---

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

# Example

```
1 #include <iostream>
2 using namespace std;
3
4 class Shape
5 {
6     protected:
7         int width, height;
8
9     public:
10        Shape(int a = 0, int b = 0)
11        {
12            width = a;
13            height = b;
14        }
15
16        virtual int area()
17        {
18            cout << "Parent class area: " << endl;
19            return 0;
20        }
21 };
22
23 class Rectangle: public Shape
24 {
25     public:
26        Rectangle( int a = 0, int b = 0):Shape(a, b)
27        { }
28
29        int area ()
30        {
31            cout << "Rectangle class area: " <<
32                << endl;
33            return width * height;
34 };
35
36 class Triangle: public Shape
37 {
38     public:
39        Triangle(int a = 0, int b = 0):Shape(a, b)
40        { }
41
42        int area ()
43        {
44            cout << "Triangle class area : " << endl;
45            return (width * height / 2);
46        }
47 };
48
49 // Main function for the program
50 int main( )
51 {
52     Shape *shape;
53     Rectangle rec(10,7);
54     Triangle tri(10,5);
55
56     // store the address of Rectangle
57     shape = &rec;
58
59     // call rectangle area.
60     shape->area();
61
62     // store the address of Triangle
63     shape = &tri;
64
65     // call triangle area.
66     shape->area();
67
68     return 0;
69 }
```

Blarg blarg blarg



# Pure Virtual Functions

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

```
1 class Shape
2 {
3     protected:
4         int width, height;
5
6     public:
7         Shape(int a = 0, int b = 0)
8         {
9             width = a;
10            height = b;
11        }
12
13        // pure virtual function
14        virtual int area() = 0;
15 };
16
```

The `= 0` tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

## Abstraction

---

t

t

t

t

t

## Encapsulation

---



t

t

t

## Interfaces

---

t

t

t

