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

Object Orientation

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 ccertain methods such as accelerate, brake, and so on.

Overview

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

C++ Class Definitions

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**.

Define C++ Objects

We declare objects of a class with exatly the same sort of declaration that we declare variables of basi 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 (.).

```cpp
#include <iostream>

using namespace std;

class Box {
   public:
      double length;  // Length of a box
      double breadth; // Breadth of a box
      double height;  // Height of a box
};

int main() {
   Box box1;            // Declare Box1 of type Box
   Box box2;            // Declare Box2 of type Box
   double volume = 0.0; // Store the volume of a box here

   // box 1 specification
   box1.height  = 5.0;
   box1.length  = 6.0;
   box1.breadth = 7.0;

   // box 2 specification
   box2.height  = 10.0;
   box2.length  = 12.0;
   box2.breadth = 13.0;

   // volume of box 1
   volume = box1.height * box1.length * box1.breadth;
   cout << "Volume of Box1 : " << volume << endl;

   // volume of box 2
   volume = box2.height * box2.length * box2.breadth;
   cout << "Volume of Box2 : " << volume << endl;

   return 0;
}
```

Classes & Objects in Detail I

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

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

Classes & Objects in Detail II

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

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.

Base & Derived Classes

Blarg blarg blarg

```cpp
#include <iostream>
using namespace std;

// Base class
class Shape
{
    public:
        void setWidth(int width)
        {
            this.width = width;
        }

        void setHeight(int height)
        {
            this.height = height;
        }

    protected:
        int width, height;
};

// Derived class
class Rectangle: public Shape
{
    public:
        int getArea()
        {
            return width * height;
        }
};
```

```cpp
int main()
{
    Rectangle rect;

    rect.setWidth(5);
    rect.setHeight(7);

    // Print the area of the object.
    cout << "Total area: " << rect.getArea() <<
    ↪    endl;

    return 0;
}
```

Blarg blarg blarg the result is 35

Access Control and Inheritance

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.

Type of Inheritance

t

Multiple Inheritance

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

```cpp
#include <iostream>
using namespace std;

// Base class Shape
class Shape
{
    public:
        void setWidth(int width)
        {
            this.width = width;
        }

        void setHeight(int height)
        {
            this.height = height;
        }

    protected:
        int width, height;
};

// Base class PaintCost
class PaintCost
{
    public:
        int getCost(int area)
        {
            return area * 70;
        }
};
```

```cpp
// Derived class
class Rectangle: public Shape, public PaintCost
{
    public:
        int getArea()
        {
            return width * height;
        }
};

int main()
{
    Rectangle rect;
    int area;

    rect.setWidth(5);
    rect.setHeight(7);

    area = rect.getArea();

    // Print the area of the object.
    cout << "Total area: " << rect.getArea() <<
    ↪    endl;

    // Print the total cost of painting
    cout << "Total paint cost: $" <<
    ↪    rect.getCost(area) << endl;

    return 0;
}
```

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++

```cpp
#include <iostream>
using namespace std;

class PrintData
{
    public:
        void print(int i)
        {
            cout << "Printing int: " << i << endl;
        }

        void print(double  f)
        {
            cout << "Printing float: " << f << endl;
        }

        void print(char* c)
        {
            cout << "Printing character: " << c <<
            ↪    endl;
        }
};
```

```cpp
int main()
{
    PrintData printdata;

    // Call print to print integer
    printdata.print(5);

    // Call print to print float
    printdata.print(500.263);

    // Call print to print character
    printdata.print("Hello C++");

    return 0;
}
```

Polymorphism

Polymorphism in C++

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

```cpp
#include <iostream>
using namespace std;

class Shape
{
    protected:
        int width, height;

    public:
        Shape(int a = 0, int b = 0)
        {
            width = a;
            height = b;
        }

        virtual int area()
        {
            cout << "Parent class area: " << endl;
            return 0;
        }
};

class Rectangle: public Shape
{
    public:
        Rectangle( int a = 0, int b = 0):Shape(a, b)
        { }

        int area ()
        {
            cout << "Rectangle class area: " <<
            ↪    endl;
            return width * height;
        }
};
```

```cpp
class Triangle: public Shape
{
    public:
        Triangle(int a = 0, int b = 0):Shape(a, b)
        { }

        int area ()
        {
            cout << "Triangle class area :" << endl;
            return (width * height / 2);
        }
};

// Main function for the program
int main( )
{
    Shape *shape;
    Rectangle rec(10,7);
    Triangle  tri(10,5);

    // store the address of Rectangle
    shape = &rec;

    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;

    // call triangle area.
    shape->area();

    return 0;
}
```

Virtual Function

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 };
```

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

Abstraction

Data Abstraction in C++

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

In C++, we use classes to define our own abstract data types (ADT). You can use the cout object of class ostream to stream data to standard output like this:

```cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello C++" << endl;
    return 0;
}
```

Access Labels Enforce Abstraction

Blarg blarg blarg

Benefits of Data Abstraction

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction.

Designing Strategy

Blarg blarg blarg

Encapsulation

Data Encapsulation in C++

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

Data Encapsulation Example

```
1  class Box
2  {
3      public:
4          double getVolume()
5          {
6              return length * breadth * height;
7          }
8
9      private:
10         double length;  // Length of a box
11         double breadth; // Breadth of a box
12         double height;  // Height of a box
13 };
```

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Designing Strategy

Blarg blarg blarg

Interfaces

Interfaces in C++ (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes. A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
1  class Box
2  {
3      public:
4          // Pure virtual function
5          virtual double getVolume() = 0;
6
7      private:
8          double length;  // Length of a box
9          double breadth; // Breadth of a box
10         double height;  // Height of a box
11 };
```

The purpose of an abstract class is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Abstract Class Example

```cpp
#include <iostream>
using namespace std;

// Base class
class Shape
{
    public:
        // Pure virtual function providing interface framework.
        virtual int getArea() = 0;

        void setWidth(int width)
        {
            this.width = width;
        }

        void setHeight(int h)
        {
            this.height = height;
        }

    protected:
        int width;
        int height;
};

// Derived classes
class Rectangle: public Shape
{
    public:
        int getArea()
        {
            return width * height;
        }
};
```

```cpp
class Triangle: public Shape
{
    public:
        int getArea()
        {
            return (width * height) / 2;
        }
};

int main(void)
{
    Rectangle rect;
    Triangle  tri;

    rect.setWidth(5);
    rect.setHeight(7);

    // Print the area of the object.
    cout << "Total rectangle area: " << rect.getArea() << endl;

    tri.setWidth(5);
    tri.setHeight(7);

    // Print the area of the object.
    cout << "Total triangle area: " << tri.getArea() << endl;

    return 0;
}
```

Designing Strategy

Blarg blarg blarg

References I