A Tutorial:
Object-Oriented Programming/C++

Mehdi Raoufi

Department of Information Systems and Computer Programming
Purdue University Calumet
Hammond, Indiana 45323, USA

Abstract

Often the transition from procedural programming to object-oriented programming is painful for many students who have extensive experience in procedural programming with no exposure to object-oriented concepts. In this tutorial I will show how object-oriented programming promotes a new level of abstraction and reusability using inheritance and polymorphism. The C++ is used for presentation.

Keywords: Object-Oriented Programming (OOP), class, object, inheritance, polymorphism, virtual function, abstract class

Content:

1. Introduction

In **Procedural programming,** a program is constructed using functions. In **Object-Oriented Programming,** a program is constructed using interactive objects. Every object-oriented programming language must have facility to define:

a. **Class**
b. **Inheritance**
c. **Polymorphism**

Abstract Data Type (ADT)

ADT refers to any user-defined type. In C and later in C++ with some modification a mechanism to define ADT is given using the keyword struct (structure).

The keyword struct is used to define a record. The syntax to define a struct is:

```
struct Type-name
{
   variable declaration1;
   variable declaration2;
            .
            .
            .
};
```

Example 1.1: Use a struct in C++ to define an ADT, which represents a point in a rectangular coordinate system. Then define a function to rotate a given point 90 degrees clockwise.

```
// File name:point1.cpp
// This program will use struct to
// demonstrate a simple ADT Point1
#include<iostream>
using namespace std;
// Point1 is defined as an ADT.
```

```
// It has two data members x, and y.
// Both x, and y are public by default,
// it means that they are directly
// accessible from main program.
// Syntax to define a struct Point1

struct Point1
{
      double x;
      double y;
};
// prototype for function rotate90():
// Function rotate90 will accept a
// variable of type
// Point1 (ADT) and will rotated
//  it 90 degrees clockwise.
void rotate90( Point1 &p);
int main()
{
      Point1 point;
      // point is a variable of type Point1.
      // Both x, and y are directly
      // accessible from main program.
      point.x = 3;
      point.y = 4;
      // Output coordinates of point.
      cout << "Coordinates of point:"
       << endl;
      cout << "The x coordinate is: "
      << point.x << endl;
      cout << "The y coordinate is: "
      << point.y << endl;
      cout << endl;
      rotate90(point);
      // Output coordinates of point after it
      // is rotated 90 degrees clockwise.
      cout <<
      "Coordinates of point after it is rotation"
      << endl;
      cout << "The x coordinate is: " <<
      point.x << endl;
      cout << "The y coordinate is: " <<
      point.y << endl;
      return 0;
}
// Definition of function rotate90()
void rotate90( Point1 &p)
{
      double x1, y1;
      // Rotate p 90 degrees clockwise.
      x1 = p.y;
      y1 = -p.x;

      p.x = x1;
      p.y = y1;
}
```

Program 1.1

2. The class as an ADT

Object-oriented programming language C++ has a mechanism to define a class where we can easily incorporate encapsulation and information hiding. In the definition of an ADT using class, the data members and member functions are presented as a package and public functions (interfaces) are used to manipulate private data members.

The syntax to define a class in C++ is similar to a struct. The only difference is that the data members and member functions are **private** by default. That is, members are not accessible directly from the main program. If we want a member to be **public** we should specify it as public. Public data members are directly accessible from the main program (see example 2.1).

```
class ClassName
{
   variable declaration1;
   variable declaration2;
            .
            .
public:
   function prototype1;
   function prototype2;
            .
            .
};
```

Information Hiding

If the body of the function rotate90() is hidden from us, and we are only concerned with what task rotate90() performs, then this is referred as **information hiding**.

Encapsulation

Combining variables of ADT with associated functions as a package and using functions to manipulate the variables is sometimes referred to as **encapsulation**.

Example 2.1: We can rewrite the program of example 1.1 using class instead of struct. As we said, C++ makes data members of class private by default. Note that x and y are not directly accessible, and that is why we have defined two public member functions (public interfaces), getX() and getY(), to access these data members.

```
// File name:point2.cpp
// This program will use class to
// demonstrate a simple ADT
// Data members are encapsulated
// with associated member functions
#include<iostream>
```

```
using namespace std;
// Point2 defined as an ADT.
// It has two data members x, and y.
// Both x, and y are private by
// default, it means that they are
// not accessible directly
// from main program.
// Only public member functions
// can be used to access private
// data members.

class Point2
{
      double x;
      double y;
public:
      // Function Point2() is a constructor.
      // Constructor  functions
      // have the same name as class
      // and do not return any
      // values. Constructors are
      // used to initialize data
      // members at declaration time.
      // If no argument is expressed
      // for Point2 at the time of
      // declaration both x, and y
      // will be initialized to zeros
      // prototype for point2()
      Point2( double = 0, double = 0);
      // Function to access x
      double getX();
      // Function to access y
      double getY();
      // Function to rotate point 90
      // degrees clockwise
      void rotate90();
};
// Definition of Point2()
 //  The scope resolution operator ::
// specifies that the function
//  Point2() belongs to the class Point2.
Point2 :: Point2( double x1, double y)

{
      x = x1;
      y = y1;
}
double Point2 :: getX()
{
      return x;
}
double Point2 :: getY()
{
      return y;
}
void Point2 ::rotate90()
{
      double x1, y1;
      x1 = y;
      y1 = -x;
      x = x1;
      y = y1;
}
int main()
{
      Point2 point(3,4);
      // Both x, and y are not directly accessible
      // from main program.
      // Output coordinates of point.
      cout << "Coordinates of point:" << endl;
      cout << "The x coordinate is: "
      << point.getX() << endl;
      cout << "The y coordinate is: "
      << point.getY() << endl;
      point.rotate90();
      // Output coordinates of point
      // after it is rotated 90 degrees
      // clockwise.
      cout <<
      "Coordinates of point after it is rotated 90"
      << endl;
      cout <<
      "The x coordinate is: " << point.getX()
      << endl;
      cout <<
      "The y coordinate is: " << point.getY()
      << endl
      return 0;
}
```

Program 2.1

3. Inheritance and Reuse of Existing Implementation

Inheritance: Inheritance is a mechanism by which one class acquires the data members and member functions of another class.

Let's define class A with two data members i and j, and two member functions, doSomeThing1() and doSomeThing2(). A() is default constructor for class A.

```
class A
{
   int i,j;
public:
   A();
   void doSomeThing1();
   void doSomeThing2();
};
```

Fig 3.1

We define class B where it is inherited from class A, Fig 3.2. It means that class B has all properties of class A. Class B inherits members i and j, and two member functions, doSomeThing1() and doSomeThing2(). Here, since we have redefined the function doSomeThing2() in class B, the new definition of doSomeThing2() overrides the early definition in class B. The definition of doSomeThing1() remain exactly the same for both of the classes. Note that the class B also has more properties which do not belong to class A; one extra data member

k, and one more member function doSomeThingElse(). The class B also inherits the default constructor of class A, meaning that each time we instantiate an object of class B the default constructor of class A will be invoked automatically, and then the default constructor of B will be executed (given that a non-default constructor or class A is not called earlier). The class A is called the **base** class, and the class B a **derived** class of A.

```
class B : public A
{
   int k;
public:
    B();
    void doSomeThing2();
    void doSomeThingElse();
};
```

Fig 3.2

The qualifier **public** preceding A means that the private data members of class A are inherited as private members of class B and public functions of class A remain as public for derived class B.

If we replace the keyword public (preceding A) with **private** in the derived class definition

```
class B : private A
```

then the public functions of base class A become private functions for derived class B, and we will not be able to invoke these functions directly for an object of class B, but public functions of class A are still accessible within the implementation of class B.

Reuse of Existing Implementation Using Inheritance

Here we learn how to define a new class using an existing class without modifying any of the member functions or data members of the existing class. Note that most often the implementation of existing class is hidden from us.

In the following example we use the **template** classes. The type of a data member of a template class is determined in the declaration of object of the class in the main program. For example if we use the template class **vector** from **STL** (Standard Template Library), the declaration of an object of class vector where the type of data members is int should be as,

```
vector<int> v;
```

Example 3.1 In this example we define the class Stack as a derived class of **vector**. The class Stack has three member functions. The function empty() returns true if the stack is empty, false otherwise. The function pop() removes the top element and returns its value. The function push() inserts the content of the argument onto the stack. Here we are using an existing class vector to create the class Stack. Note that the private derived class makes the functions of vector inaccessible from an object of class Stack, and the functions empty(), pop(), and push() are the only functions we may invoke for an object of type Stack. The definition of Stack is given in header file and implementation in implementation file.

```
// File name stack.h
// Stack is defined here
#include <vector>
// The class vector is defined in C++ STL.
#ifndef STACK_H
#define STACK_H
using namespace std;
// Stack is a template class. The type of its
// items is determined in main program.
// For example if we want
// to declare s to be Stack of type int,
// the declaration will look as
// Stack<int> s;.
// The type of Stack's items follows
// the name of the Stack between two
// angle brackets "<" and " >".
template <class Item>
class Stack : private vector<Item>
// The class Stack as a derived class of vector.
{
// Member functions of vector will become
// private member functions
// for the class  Stack, and an object of type
// Stack will not be able to invoke them.
      public:
      void push( const Item& entry);
      // Pushes the item entry onto stack.
      Item pop();
      // Removes and returns the item on the top.
      bool empty() const;
};
#include "stack.tem"
#endif

// File name stack.tem
// Implementation of class Stack
// as a derived class of vector.

template <class Item>
void Stack<Item> :: push( const Item& entry)
{
      vector<Item> :: push_back( entry );
}
template <class Item>
Item Stack<Item> :: pop()
{
      Item i = (vector<Item> :: back());
      vector<Item> :: pop_back();
      return i;
}
template <class Item>
```

```
bool Stack<Item> :: empty() const
{
      return (vector<Item> :: size() == 0 );
}

// File name: test.cpp
// Here we test the class Stack.
#include <iostream>
#include "stack.h"
using namespace std;
int main()
{
      Stack<int> s;
      s.push(3);
      s.push(19);
      s.push(8);
      s.push(43);
      s.push(90);

      while( ! s.empty())
            cout << s.pop() << endl;

      return 0;
}

90
43
8
19
3
Press any key to continue
```

Program 3.1

4. Polymorphism and Reuse of Public Interface

The static Binding: The compile-time determination of what function is bound with what object is referred as static binding.

The Dynamic Binding (Late Binding): The run-time determination of what function is bound with what object is referred as dynamic binding.

Polymorphism: A polymorphic function is a function that has a different meaning depending on what kind of object it is bound to at run-time.

Virtual Function: A virtual function is a function that allows dynamic or late binding. In C++ the keyword virtual is used to define a function as virtual. The syntax of a virtual function is demonstrated in example 4.1.

Abstract class: Abstract class is a class that cannot be instantiated.

Pure Virtual Function: Defining a member function of a class pure virtual makes the class an abstract class. To define a pure virtual function we need to add the syntax "= 0" to the end of a virtual function prototype. This is also demonstrated in example 4.1. The derived class of an abstract class will remain abstract if it does not override all pure virtual functions of the base class.

The following example will demonstrate the concept of late or dynamic binding using a virtual function, and polymorphism.

Example 4.1: In this example we define a class called Figure with two data members, x and y, and a member function, draw(). Then we define two derived classes of Figure named Circle and Square.

```
#include <iostream>
using namespace std;
class Figure
{
 protected:              // protected data members behave
like private, except that they
                         // are accessible from any derived
                class of the base class Figure.
   double x, y;
public:
   Figure( double = 0.0, double = 0.0);
   void draw();
};

Figure :: Figure( double a, double b)
{
   x = a ;
   y = b ;
}

void Figure :: draw()
{
   cout << " I will draw a figure." << endl;
}
class Circle : public Figure
{
protected:
   double radius;
public:
   Circle( double = 0, double =0, double =0);
   void draw() const ;
};
Circle ::
Circle( double a, double b, double c) : Figure (a,b)
{
   radius = c ;
}
void Circle ::draw() const
{
   cout <<  "I will draw a circle." << endl;
}
class Square : public Figure
{
protected:
   double side;
public:
```

```
   Square( double = 0, double =0, double =0);
   void draw() const ;
};
Square ::
Square( double a, double b, double c) : Figure (a,b)
{
   side = c ;
}
void Square ::draw() const
{
   cout <<  "I will draw a square." << endl;
}
int main()
{
            // Array of objects of class Figure.
            Figure  *pointersToFigure[100];

            pointersToFigure[0] = new Circle;
            pointersToFigure[1] = new Square;
            pointersToFigure[2] = new Circle;
            pointersToFigure[3] = new Square;
            for ( int i = 0; i <= 3; i++)
                        pointersToFigure[i]->draw();
            return 0;
}
```

Output:

```
I will draw a figure.
I will draw a figure.
I will draw a figure.
I will draw a figure.
Press any key to continue
```

Program 4.1

In the preceding program the function draw() did not utilize the late binding. To utilize the late binding all we need to do is to replace the prototype of the draw() function in class Figure by:

```
virtual void draw() const;
```

We need to examine the output of the program 4.1 after making this change to understand the run-time binding. Run-time binding takes place if we define function draw() as a virtual function. This is a very powerful mechanism that a pointer to the object of base class Figure can point either to an object of Square or Circle. This will enable us to override and execute the draw() function in any class, which is derived from Figure using a pointer to the base class Figure. Note that such a class usually is defined as an abstract class. To define a class as an abstract class in C++, we need to define at least one pure virtual member function. In our example to make our function to be pure virtual we need to replace the definition of draw() function in Figure class with:

```
virtual void draw()  const = 0;
```

Please note that even though we cannot instantiate an object of an abstract class, but we can define an array of pointers, where each element points to an abstract class. After changing the draw() function to virtual or pure virtual the output of Program 4.1 looks as:

```
I will draw a circle.
I will draw a square.
I will draw a circle.
I will draw a square.
Press any key to continue
```

Please note that the draw() function performs the overridden functions of Square and Circle classes based on the type of the object that it points.

REFERENCES

Dale, N., C. Weems, and M. Headington,, 2000, Programming and Problem Solving with C++, Jonesand Bartlett, Boston, MA.

Deitel, H. M. and P. J. Deitel, 1998, C++ How to Program, Prentice Hall, Upper Saddle River, NJ.

Lafore, R., 1999, Object-Oriented Programming in C++, SAMS, Indianapolis, IN.

Lippman, S. B. and J. Lajoie, 1998, C++ Primer, Addison Wesley, Reading, MA.

Main, M. and W. Savitch,, 1997, Data Structures and other Objects using C++, Addison Wesley, MA.

Savitch , W., 1999, Problem Solving with C++ The Object of Programming, Addison Wesley, Reading, MA.

In Conclusion

This tutorial has tried to demonstrate, how object-oriented programming promotes a new level of reusability and abstraction, which is not possible in procedural programming languages.