S. S Jain Subodh P.G. (Autonomous) College

SUBJECT - Object Oriented Programming Concept

TITLE - POLYMORPHISM

Created By:
Shalu J. Rajawat

Signatures

- In any programming language, a signature is what distinguishes one function or method from another
- In c++, two methods have to differ in their *names* or in the *number* or *types* of their parameters
  - foo(int i) and foo(int i, int j) are different
  - foo(int i) and foo(int k) are the same
  - foo(int i, double d) and foo(double d, int i) are different
- In C++, the signature also includes the *return type.*

Polymorphism

- **Polymorphism means *many* (poly) *shapes* (morph)**
- **The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by Inheritance.**
- **There are three types of polymophism**

1. Function overloading

2. Operator Overloading

3. Virtual function

FUNCTION Overloading

We have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number arguments in the argument list. We can not overload function declaration that differ only by return type.

PROGRAM -1

```
#include <iostream.h>
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;
                }
};
int main(void) {
        printData pd;
        pd.print(5); // Call print to print integer
        pd.print(500.263); // Call print to print float
        pd.print("Hello C++"); // Call print to print character
        return 0;
}
```

OUTPUT:
Printing int: 5
Printing float: 500.263
Printing character: Hello C++

OPERATOR OVERLOADING

We can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well. **Overloaded** operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass **two arguments** for each operand as follows:

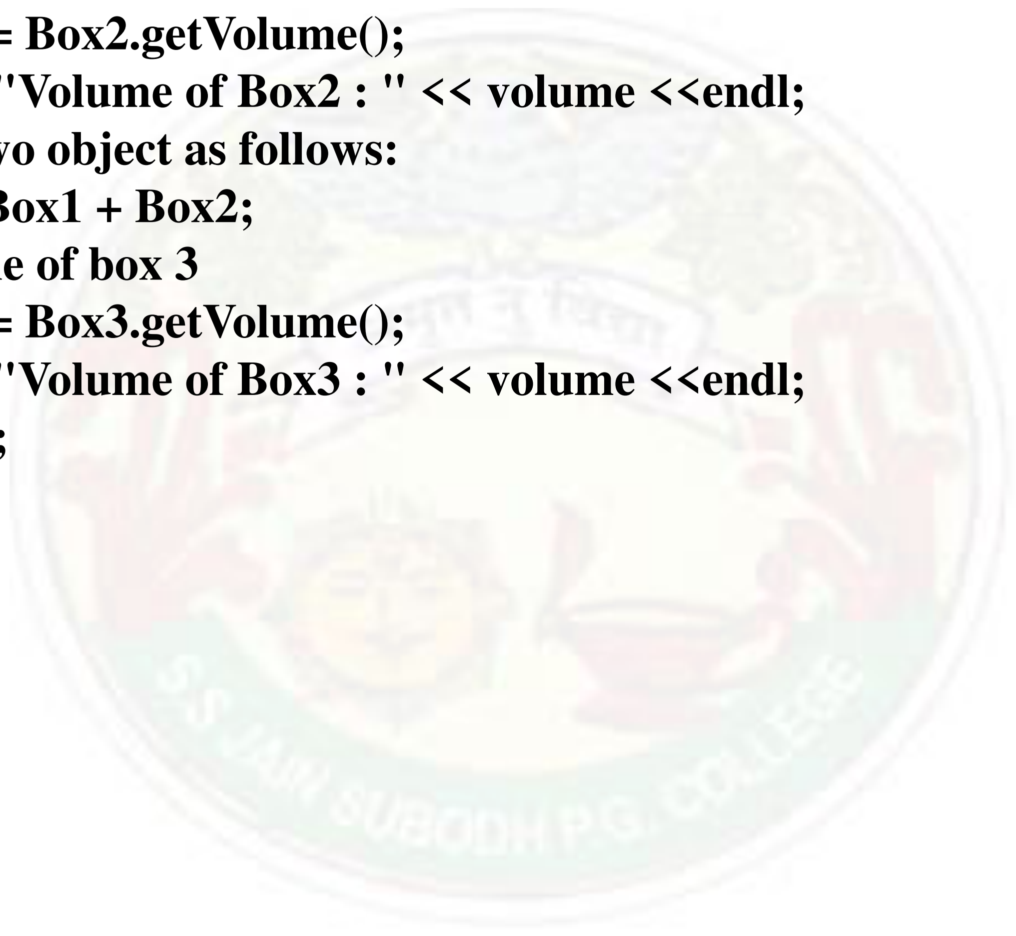
Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below:

```
#include <iostream.h>
Box {
        public:
                double getVolume(void) {
                        return length * breadth * height;
                }
                void setLength( double len ) {
                        length = len;
                 }
                void setBreadth( double bre ) {
                        breadth = bre;
                }
                void setHeight( double hei ) {
                        height = hei;
                }
                 // Overload + operator to add two Box objects.
                Box operator+(const Box& b) {
                        Box box;
                        box.length = this->length + b.length;
                        box.breadth = this->breadth + b.breadth;
                box.height = this->height + b.height;
                        return box;
                }
```

```
    private:
            double length; // Length of a box
            double breadth; // Breadth of a box
            double height; // Height of a box
};


// Main function for the program
int main( ) {
    Box Box1; // Declare Box1 of type Box
    Box Box2; // Declare Box2 of type Box
    Box Box3; // Declare Box3 of type Box
    double volume = 0.0; // Store the volume of a box here
    // box 1 specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);
    // box 2 specification
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);
```



```
        // volume of box 1
        volume = Box1.getVolume();
        cout << "Volume of Box1 : " << volume <<endl;
        // volume of box 2
        volume = Box2.getVolume();
        cout << "Volume of Box2 : " << volume <<endl;
        // Add two object as follows:
        Box3 = Box1 + Box2;
         // volume of box 3
        volume = Box3.getVolume();
        cout << "Volume of Box3 : " << volume <<endl;
        return 0;
}
```

OUTPUT:

Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

Overloadable / Non-overloadable Operators:

Following is the list of operators which can be overloaded:

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td></tr>
<tr><td>&</td><td>|</td><td>~</td><td>!</td><td>,</td><td>=</td></tr>
<tr><td><</td><td>></td><td><=</td><td>>=</td><td>++</td><td>--</td></tr>
<tr><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td></tr>
<tr><td>|=</td><td>*=</td><td><<=</td><td>>>=</td><td>[]</td><td>()</td></tr>
<tr><td>-></td><td>->*</td><td>new</td><td>new []</td><td>delete</td><td>delete []</td></tr>
</table>

Following is the list of operators, which can not be overloaded:

<table>
<tr><td>::</td><td>.*</td><td>.</td><td>?:</td></tr>
</table>

Virtual Function

1. **A virtual function is a member function that is declared within a base class and redefined by a derived class. To create virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.**
2. **Base class pointer can point to derived class object. In this case, using base class pointer if we call some function which is in both classes, then base class function is invoked. But if we want to invoke derived class function using base class pointer, it can be achieved by defining the function as virtual in base class, this is how virtual functions support runtime polymorphism.**

Difference between virtual function and non virtual function through these program:

PROGRAM-1

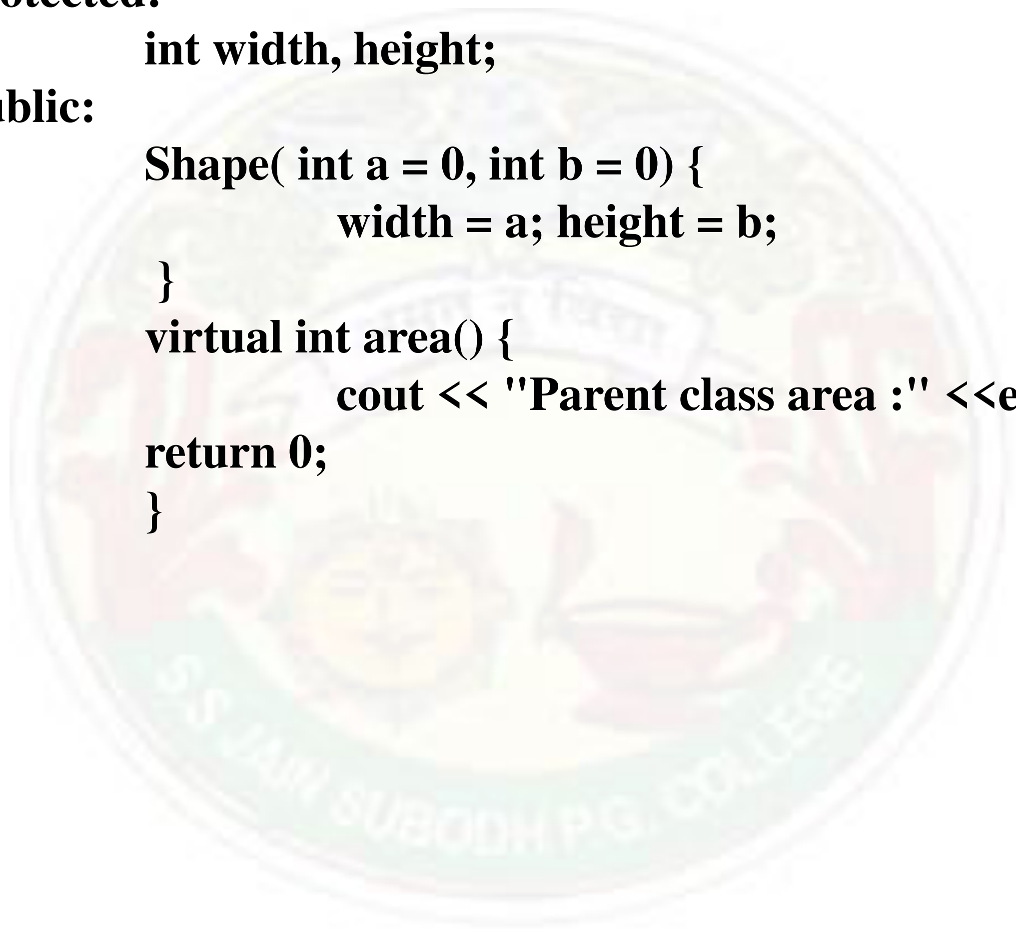
```
#include <iostream.h>
class Shape {
        protected:
                int width, height;
        public:
                Shape( int a = 0, int b = 0) {
                width = a;
                        height = b;
                }
                int area() {
                        cout << "Parent 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);
                }
};
```

```
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;
}
```

OUTPUT:

Parent class area
Parent class area

Now we will only change in shape class like this:



```
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;
                }
    };
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

OUTPUT:
Rectangle class area
Triangle class area

Pure virtual function

- **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.**
- **We can change the virtual function area() in the base class to the following:**

```
class Shape {
        protected:
                int width, height;
        public:
                Shape( int a = 0, int b = 0) {
                        width = a; height = b;
                }
                 // pure virtual function virtual
                int area() = 0;
};
```

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

THANKS