Polymorphism, Virtual Functions and Abstract Class

In C++, a pointer variable of a base class type can point to an object of its derived class. There are situations when this feature of C++ can be used to develop generic code for a variety of applications.

Pointer of base class

Consider the following program to understand pointer compatibility property

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

class Shape
{
  protected:
    double width, height;
  public:
    void set_data (double a, double b)
    {
        width = a;
        height = b;
    }
};

class Rectangle: public Shape
{
  public:
    double area ()
    {
        return (width * height);
    }
};


int main ()
{
    Shape *sPtr;    //declare pointer variables of type Shape
    Rectangle Rect; //create the object rect of type Rectangle
    sPtr = &Rect;   //make sPtr point to the object rect.

    sPtr->set_data (5,3); //set length and width of object rect
    cout << sPtr -> area() << endl;  //Compile Error !!

    return 0;
}
```

Notice that even though rectPtr is pointing to rect (object of type Rectangle), when the program executes, the statement sets length and width of rectangle. If you tried to access area function of class Rectangle with sPtr it will give you compiler error.

```
sPtr -> area()
```

is a compiler error !

It means **base class pointer can not access the additional member function of its derived class**. If we want to do this we need to type cast the base class pointer.

Using Type Casts with Base Class Pointers

We can use a type cast to get the compiler to accept the statement:

```
static_cast <Rectangle *> (sPtr)->area()
```

so we should write the statment

```
cout << static_cast <Rectangle *> (sPtr) -> area() << endl;
```

The type cast informs the compiler that sPtr is actually pointing to a Rectangle object derived from the Shape base class. In general, **a pointer to a base class that actually points to a derived class object must first be appropriately cast before the additional features of the derived class can be used.**

Virtual Function and Polymorphism

Virtual functions are used in C++ to support polymorphic behavior. We are modifing the above program and will introduce you the concept of virtual function by following example:

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

class Shape
{
  protected:
    double width, height;
  public:
    void set_data (double a, double b)
    {
        width = a;
        height = b;
    }
    virtual double area()
    {return 0;}
```

```
};

class Rectangle: public Shape
{
  public:
    double area ()
    {
        return (width * height);
    }
};

int main ()
{
    Shape *sPtr;
    Rectangle Rect;
    sPtr = &Rect;

    sPtr -> set_data (5,3);
    cout << sPtr -> area() << endl;

    return 0;
}
```

```
Output :
15
```

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual. The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. **The advantage of having virtual function is that we are able to access** area **function of derived class by pointer variable of base class.**

Pure Virtual Function and Abstract Class

In above example, base class Shape member function area do not need any implementation because it is overriding in derived class. If this is the case, the C++ language permits the programmer to declare the function a pure virtual function. The C++ way of declaring a pure virtual function is to put the expression = 0 in the class declaration. For example, if a member function double area() is being declared pure virtual, then its declaration in its class looks like

```
virtual double area() = 0;
```

A pure virtual function is sometimes called an abstract function, and a class with at least one pure virtual function is called an abstract class. The C++ compiler will not allow you to instantiate an abstract class. Abstract classes can only be subclassed: that is, you can only use them as base classes from which to derive other classes.

A class derived from an abstract class inherits all functions in the base class, and will itself be an abstract class unless it overrides all the abstract functions it inherits. The usefulness of abstract classes lies in the fact that they define an interface that will then have to be supported by objects of all classes derived from it.

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

class Shape
{
  protected:
    double width, height;
  public:
    void set_data (double a, double b)
    {
        width = a;
        height = b;
    }
    virtual double area() = 0;
};

class Rectangle: public Shape
{
public:
    double area ()
    {
        return (width * height);
    }
};

class Triangle: public Shape
{
public:
    double area ()
    {
        return (width * height)/2;
    }
};



int main ()
{
```

```
    Shape *sPtr;

    Rectangle Rect;
    sPtr = &Rect;

    sPtr -> set_data (5,3);
    cout << "Area of Rectangle is " << sPtr -> area() << endl;

    Triangle Tri;
    sPtr = &Tri;

    sPtr -> set_data (4,6);
    cout << "Area of Triangle is " << sPtr -> area() << endl;
    return 0;
}
```

```
Output :
Area of Rectangle is 15
Area of Triangle is 12
```