# UEE1303(1070) S12: Object-Oriented Programming

## Polymorphism

### *What you will learn from Lab 11*

In this laboratory, you will learn the concept of polymorphism in object-oriented programming.

## TASK 11-1 VIRTUAL DESTRUCTOR

✓

```
// lab11-1.cpp
#include <iostream>
using std::cout;   using std::endl;

class Point2D
{
private:
    int *x;
    int *y;
public:
    Point2D(){x = new int (0);y=new int (0); cout << "New X and Y" << endl;}
    ~Point2D(){delete x; delete y; cout << "Delete X and Y" << endl;}
};
class Point4D : public Point2D
{
private:
    int *z;
    int *t;
public:
    Point4D() : Point2D()
        {z = new int (0); t = new int (0);cout << "New Z and T " << endl;}
    ~Point4D(){delete z; delete t; cout << "Delete Z and T" << endl;}
};
int main()
{
    Point2D *pt = new Point4D;
    delete pt;
    return 0;
}
```

- The above program will produce the output

  `New X and Y`

  `New Z and T`

  `Delete X and Y`

- Note that, it is valid to create a `Point2D` pointer by `Point4D` object since every `Point4D`

is a `Point2D`. It is invalid to write as "`Point4D *pt = new Point2D;`" cannot down-casting

- In this example, proper cleanup is not achieved here. That is because the `new` operator constructed an object of type `Point4D`, but the `delete` operator cleaned up an object of type `Point2D` pointed to by `pt`.
- Please declare the destructor of the base class `Point2D` to be `virtual`, and execute the program again.

## TASK 11-2 VIRTUAL FUNCTION

✓ Virtual destructor provides run-time polymorphism to proper cleanups. Similarly, virtual functions are also used to allow run-time polymorphism as the following example.

```
// lab11-2.cpp
#include <iostream>
using std::cout;   using std::endl;

/*  The Point2D and Point4D defined in lab11-1                          */
/*  Add declarations of display() in Point2D and Point4D, respectively. */

void Point2D::display() const
{
   cout << *x << "," << *y;
}
void Point4D::display() const
{
   Point2D::display();
   cout << "," << *z << "," << *t;
}

int main()
{
   Point2D *pt = new Point4D;
   pt->display(); cout << endl;
   delete pt;

   return 0;
}
```

- The above program shows "`0,0`" for `pt` on screen. However, `pt` is created by object of type `Point4D`, and the expected result should be "`0,0,0,0`".
- Please declare the `display` of the base class `Point2D` as a `virtual function`, and execute the program again.
- In general, a class with a virtual function should have a virtual destructor, because run-time polymorphism is expected for such a class.

## TASK 11-3 ABSTRACT CLASSES

✓ A virtual function is called a pure virtual function if it is declared but its definition is not provided. A class with one or more pure virtual functions is called an abstract class.

```
// lab11-3.cpp
#include <iostream>
using std::cout;  using std::endl;

class Shape
{
protected:
    int color;
public:
    virtual void draw() = 0;
    virtual bool is_closed() = 0;
    virtual ~Shape(){}
};
int main()
{
    Shape s;
    return 0;
}
```

```
[Error] cannot declare variable 's' to be of abstract type 'Shape'
[Note] because the following virtual functions are pure within 'Shape':
[Note] virtual void Shape::draw()
[Note] virtual bool Shape::is_closed()
```

➢ The object of an abstract class cannot be defined. It is illegal to define as "Shape s;"

➢ Some virtual functions in a base class can only be declared but cannot be defined, since the base class may not have enough information to do so and such virtual functions are only meant to provide a common interface for the derived classes.

✓ Class `Circle` is derived from the abstract class Shape.

```
// lab11-3-1.cpp
#include <iostream>
using std::cout;  using std::endl;

/* abstract class Shape defined in lab11-3  */
/* general class Point2D defined in lab11-2  */

class Circle: public Shape
{
private:
    Point2D center;
    double radius;
public:
    // constructor of Circle.
    void draw();
    bool is_closed() {return true;}
};
```

```
int main()
{
    Point2D pt(3,4);
    Circle c(pt,5,255);
    c.draw();

    return 0;
}
```

- Please finish the program as `task A` in exercise 11-1. Note that, you can add any member function to the abstract class Shape or class `Point2D` if necessary.

- ✓ Class `Polygon` is also an abstract class since it has pure virtual functions `draw()` and `rotate()`, which are inherited from its base class Shape but have not been defined. Thus objects of `Polygon` cannot be created. However, `Triangle` is no longer an abstract class since it does not contain any pure virtual functions.

```
// lab11-3-2.cpp
#include <iostream>
using std::cout;  using std::endl;

/* abstract class Shape defined in lab11-3  */
/* general class Point2D defined in lab11-2  */

class Polygon: public Shape
{
public:
    bool is_closed() {return true;}
};
class Triangle: public Polygon
{
private:
    Point2D *vertices;
public:
    // constructor for Triangle
    ~Triangle() {delete [] vertices;}
    void draw();
};

int main()
{
    Point2D *vec = new Point2D[3];
    vec[0].setPoint2D(1,1);
    vec[1].setPoint2D(6,1);
    vec[2].setPoint2D(1,8);
```

```
    Triangle t(vec,255);
    delete []vec;

    t.draw();


    return 0;
}
```

- Please finish the program as `task B` in exercise `11-1`. Note that, you can add any member function to the abstract class Shape or class `Point2D` if necessary.

## TASK 11-6 EXERCISE

### 1. *SHAPE

✓ Task A: Please finish the program `lab11-3-1` and show the follows on screen.

```
Color: 255
Center: 3,4
Radius: 5
```

✓ Task B: Please finish the program `lab11-3-2` and show the follows on screen.

```
Color: 255
Vertices:
1,1
6,1
1,8
```

✓ Task C: Define a class Shape which includes a virtual function `area()` to compute the area of a shape. Three classes `Triangle`, `Circle` and `Rectangle` are derived from Shape, respectively, and override the virtual function `area()` in each of the derived classes. In main function, create an array of pointers to Shape that actually point to objects of `Triangle`, `Circle` and `Rectangle`, and create an array of `Point2D`. Call `area()` to calculate the area for each of the Shape objects. The main function is defined as follows.

```
int main()
{
    Point2D pt(3,4);
    Circle cir(pt, 5);

    Point2D *vec = new Point2D [3];
    vec[0].setPoint2D(1,1);    vec[1].setPoint2D(1,6);
    vec[2].setPoint2D(8,1);

    Triangle tri(vec);
    delete []vec;
    vec = new Point2D [4];
    vec[0].setPoint2D(1,1);    vec[1].setPoint2D(6,1);
```

```
    vec[2].setPoint2D(6,6);    vec[3].setPoint2D(1,6);

    Rectangle rect(vec);
    delete [] vec;

    Shape *collection[3];
    collection[0] = &cir;
    collection[1] = &tri;
    collection[2] = &rect;

    cout << "Area of Circle: " << collection[0]->area() << endl;
    cout << "Area of Triangle: " << collection[1]->area() << endl;
    cout << "Area of Rectangle: " << collection[2]->area() << endl;

    return 0;
}
```

## 2. MATRIX

✓ Mtx is an abstract class which defines the interface of derived class. FullMatrix and SymmetricMatrix are two derived class and inherit from abstract class Mtx. Moreover, UpperTriMatrix and LowTriMatrix inherit from SymmetricMatrix. Please finish this program and show correct results. Note that you can add any member if necessary.

```
#include <iostream>
using namespace std;
class Mtx
{
protected:
    int dim;
public:
    virtual int &operator()(int i, int j) =0;
    virtual const int &operator() (int i, int j) const = 0;
    virtual void showMatrix() const = 0;
    virtual ~Mtx(){}
};

class FullMatrix: public Mtx
{
private:
    int **matrix;
public:
    FullMatrix(int n)
    {
        dim = n;
        matrix = new int *[dim];
        for (int i=0;i<dim;i++) matrix[i] = new int [dim];
        for (int i=0;i<dim;i++)
            for (int j=0;j<dim;j++) matrix[i][j] = 0;
    }
```

```
    int &operator()(int i, int j)
    {
        // you may provide boundary checking
        return matrix[i][j];
    }
    // …
};

class SymmetricMatrix: public Mtx
{
private:
    int **matrix;
public:
    SymmetricMatrix(int n)
    {
        dim = n;
        matrix = new int *[dim];
        for (int i=0;i<dim;i++) matrix[i] = new int [i+1];
        for (int i=0;i<dim;i++)
            for (int j=0;j<=i;j++) matrix[i][j] = 0;
    }
    int &operator()(int i,int j)
    {
        // you need provide boundary checking
        if (i>=j) return matrix[i][j];
        else return matrix[j][i];
    }
    // …
};

class LowTriMatrix: public SymmetricMatrix
{
public:
    LowTriMatrix(int n): SymmetricMatrix(n){}
    // …
};

class UpperTriMatrix : public SymmetricMatrix
{

public:
    UpperTriMatrix(int n):SymmetricMatrix(n){}
    // …
};
int main()
{
    FullMatrix A(2);
    A(0,0) = 5; A(0,1) = 4; A(1,0) = 3; A(1,1) = 6; A(100,100) = 10;
    SymmetricMatrix B(2);
    B(0,0) = 5; B(1,0) = 3; B(1,1) = 6; B(100,100) = 10
```

```
    UpperTriMatrix C(2);
    C(0,0) = 5; C(0,1) = 3; C(1,1) = 6; C(100,100) = 10;

    LowTriMatrix D(2);
    D(0,0) = 5; D(1,0) = 3; D(1,1) = 6; D(100,100) = 10;

    // you should not assign A(100,100), B(100,100), C(100,100) and D(100,100)

    UpperTriMatrix E(2);
    E(0,0) = 5; E(1,0) = 3; E(1,1) = 6;   // you should not assign E(1,0)

    LowTriMatrix F(2);
    F(0,0) = 5; F(0,1) = 3; F(1,1) = 6;   // you should not assign F(0,1)

    Mtx *vec[6];
    vec[0] = &A;    vec[1] = &B;
    vec[2] = &C;    vec[3] = &D;
    vec[4] = &E;    vec[5] = &F;

    for (int i=0;i<6;i++)
    {
        vec[i]->showMatrix();       cout << endl;
    }

    return 0;
}
```

✓ The results are,

```
5 4
3 6

5 3
3 6

5 3
0 6

5 0
3 6

5 0
0 6

5 0
0 6
```