Virtual functions & Polymorphism

Lecture 19

Concrete classes

- Classes that can be used to instantiate objects are called concrete classes
- Such classes provide implementations of every member function they define

Example

```
class point
{  int x,y,z;
  public:
  point(int d,int e, int f)
 { x=d; y=e; f=z; }
void display()
{cout<<x<<y<<z; }
};
```

```
void main()
{
  point a(10,20,15);
  a.display();
}
```

Abstract classes

- Sometimes it is useful to define classes from which programmer never intends to instantiate any objects
- Such classes are called ***abstract classes***

contd..

- These classes are normally used as base classes in inheritance hierarchies
- Generally referred to as ***abstract base classes***

Abstract base classes

- Cannot be used to instantiate objects as these are incomplete
- Derived classes must define/complete the missing part
- Good object oriented programming practice

Pure virtual function

- A class is made abstract by declaring one or more of its virtual functions to be “pure”

- A pure virtual function is specified by placing “=0” in its declaration

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

Pure virtual function

- Pure virtual functions do not provide implementations
- Every concrete derived class must override all base-class pure virtual functions

Virtual function vs. pure virtual function

- Virtual function has an implementation and gives the derived class the ***option*** of overriding the function

- A pure virtual function does not provide implementation and ***requires*** the derived class to override the function

Example

```
class quadrilateral
{  public:
   virtual void area() =0 };
```

```
void main()
{   quadrilateral *q=new
    square(3);
   q->area();   delete q;
   q=new rectangle(2,4);
   q->area();}
```

```
class square : public quadrilateral
{  int side;
  public:  square(int i=1) { side=i; }
  void area() { cout<<"\n Area of square is : "<<(side*side); }};
```

```
class rectangle : public quadrilateral
{ int side1;  int side2;
  public:  rectangle(int i,int j) { side1=i; side2=j; }
  void area() { cout<<"\n Area of rectangle is : <<(2*side1*side2);}
```

Pointers to abstract base classes

- Though we cannot instantiate objects of an abstract base class, we can use the abstract base class to declare pointers and references that can refer to objects of any concrete class derived from the abstract class

Virtual destructors

```
class quadrilateral
{ public:    virtual ~quadrilateral() {
    cout<<"\n Base class " ;}
};
```

```
void main()
{   quadrilateral *q=new
    square(3);
   delete q;
   q=new rectangle(2,4);
   delete q; }
```

```
class square : public quadrilateral
{  int side;
  public:  square(int i=1) { side=i; }
    ~square() { cout<<"\n Square class " ; }};
```

```
class rectangle : public quadrilateral
{  int side1;  int side2;
  public:  rectangle(int i,int j) { side1=i; side2=j; }
  ~rectangle() { cout<<"\n Rectangle class " ; }};
```

Class assignment

Consider the following class definition

```
class father {
protected : int age;
public;
father (int x) {age = x;}
virtual void iam ( )
{ cout < < “I AM THE FATHER, my age is : ”<< age<< end1:}
};
```

Derive the two classes son and daughter from the above class and for each, define iam ( ) to write our similar but appropriate messages. You should also define suitable constructors for these classes. Now, write a main ( ) that creates objects of the three classes and then calls iam ( ) for them.

Declare pointer to father. Successively, assign addresses of objects of the two derived classes to this pointer and in each case, call iam ( ) through the pointer to demonstrate polymorphism in action.