GE3 Computer Science

C and C ++ Lecture series *for*

B.SC 3rd semester *by*

Subhadip Mukherjee

Department of computer science

Kharagpur College

LECTURE 15

Inheritance Concept

Polygon

Rectangle

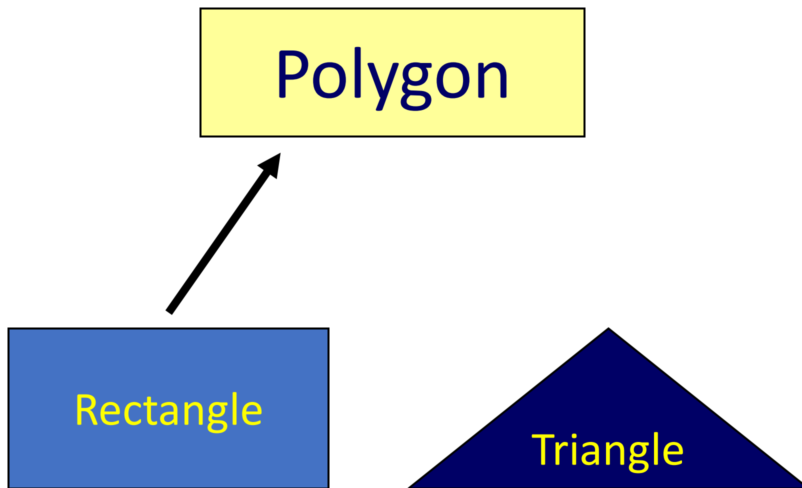
Triangle

```
class Rectangle{
    private:
      int numVertices;
      float *xCoord, *yCoord;
    public:
      void set(float *x, float *y, int nV);
      float area();
};
```

```
class Polygon{
  private:
    int numVertices;
    float *xCoord, *yCoord;
  public:
    void set(float *x, float *y, int nV);
};
```

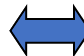
```
class Triangle{
    private:
      int numVertices;
      float *xCoord, *yCoord;
    public:
      void set(float *x, float *y, int nV);
      float area();
};
```

Inheritance Concept



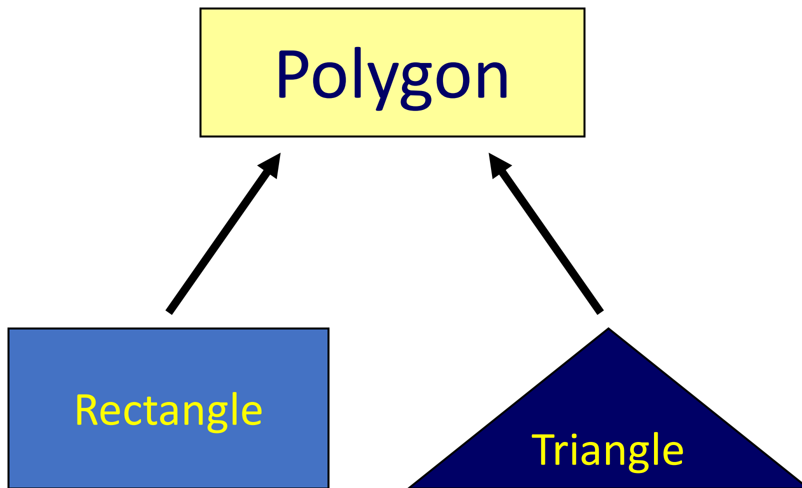

```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
};
```

```
class Rectangle : public Polygon{
    public:
       float area();
};
```



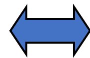
```
class Rectangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
       float area();
};
```

Inheritance Concept



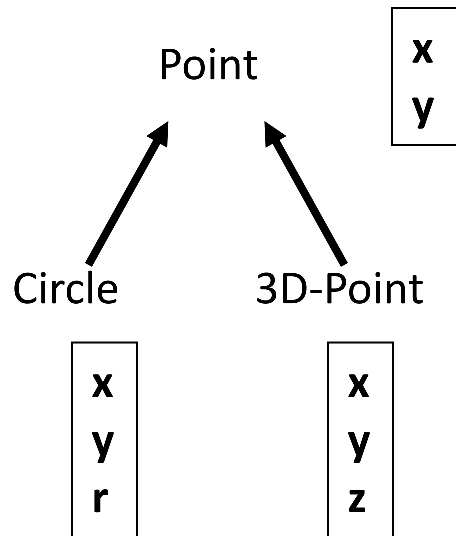

```
class Polygon{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
};
```

```
class Triangle : public Polygon{
    public:
     float area();
};
```



```
class Triangle{
    protected:
      int numVertices;
      float *xCoord, float *yCoord;
    public:
       void set(float *x, float *y, int nV);
       float area();
};
```

Inheritance Concept



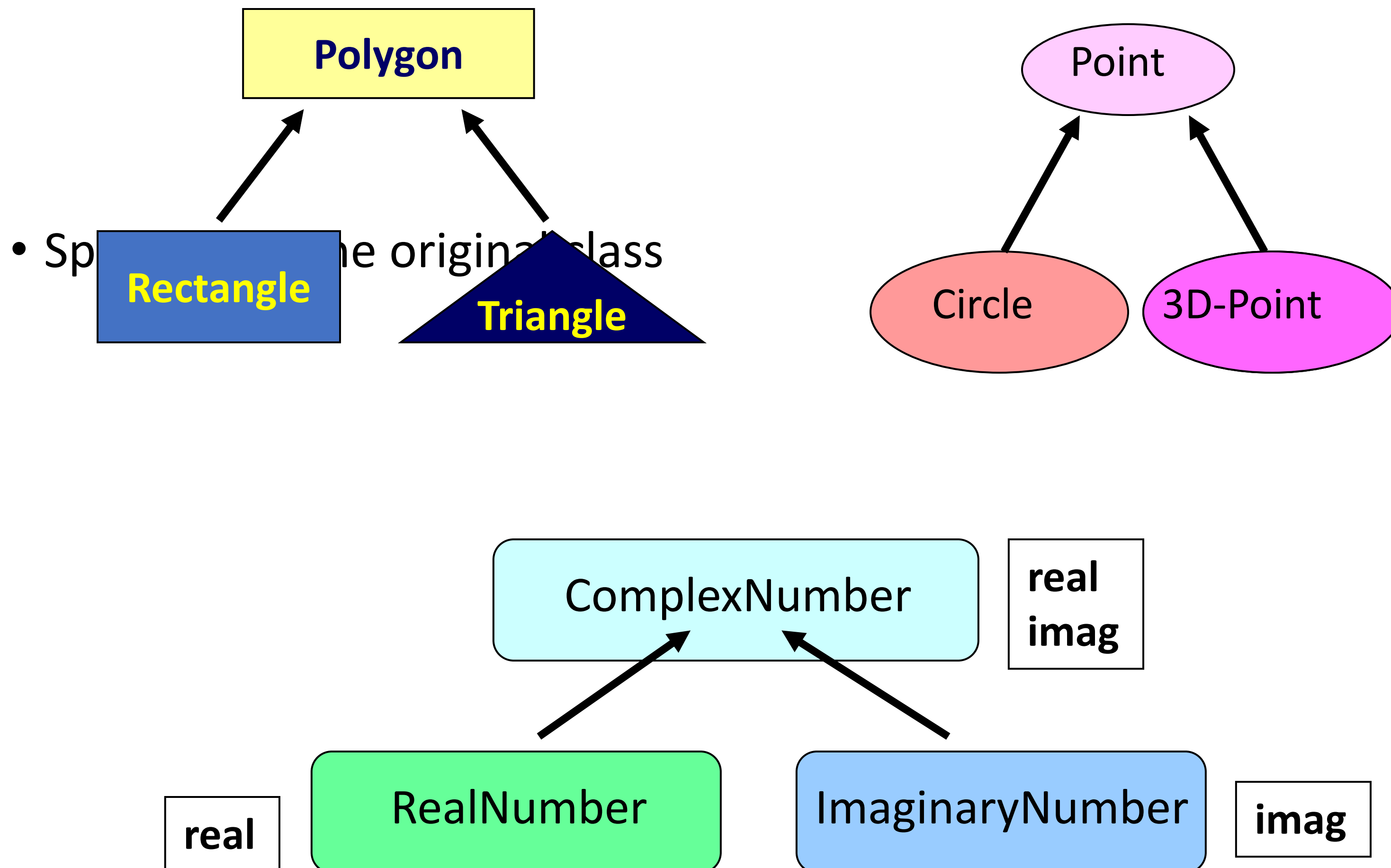

```
class Point{
    protected:
      int x, y;
    public:
      void set (int a, int b);
};
```

```
class Circle : public Point{
    private:
        double r;
};
```

```
class 3D-Point: public Point{
    private:
        int z;
};
```

Inheritance Concept

- Augmenting the original class



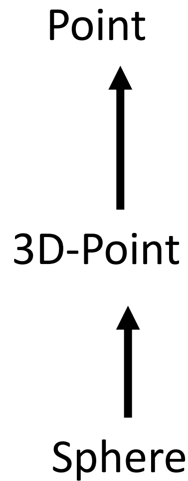
Why Inheritance ?

Inheritance is a mechanism for

- building class types from existing class types
- defining new class types to be a
  - specialization
  - augmentation

  of existing types

Class Derivation



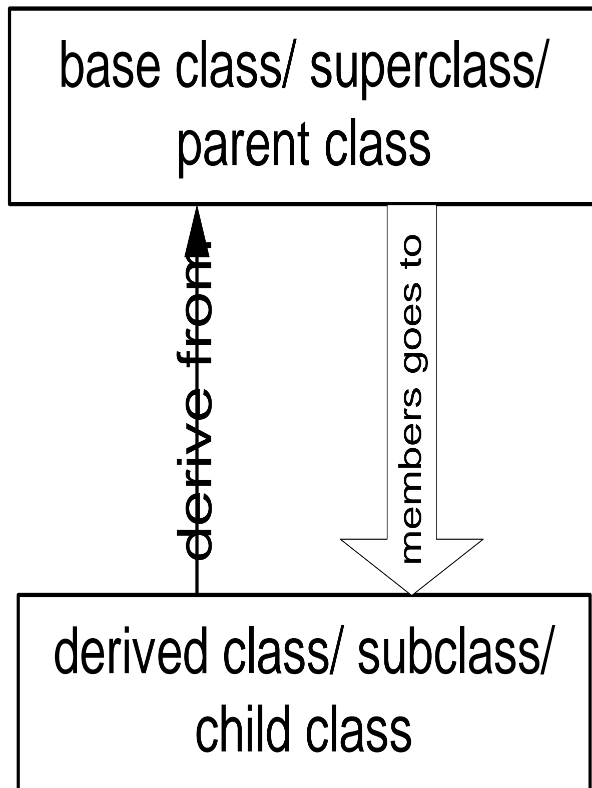

```
class Point{
    protected:
        int x, y;
    public:
        void set (int a, int b);
};
```

```
class 3D-Point : public Point{
    private:
        double z;
    … …
};
```

```
class Sphere : public 3D-Point{
    private:
        double r;
    … …
};
```

Point is the base class of 3D-Point, while 3D-Point is the base class of Sphere

Access Control Over the Members



- Two levels of access control over class members
  - class definition
  - inheritance type

```
class Point{
    protected: int x, y;
    public: void set(int a, int b);
};
```

```
class Circle : public Point{
    … …
};
```

Access Rights of Derived Classes

Type of Inheritance

Access Control for Members

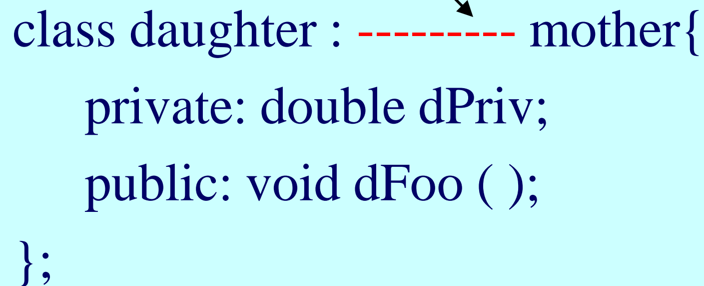
	private	protected	public
private	-	-	-
protected	private	protected	protected
public	private	protected	public

- The type of inheritance defines the access level for the members of derived class that are inherited from the base class

Class Derivation

```
class mother{
    protected: int mProc;
    public: int mPubl;
    private: int  mPriv;
};
```

private/protected/public




```
void daughter :: dFoo ( ){
    mPriv = 10;   //error
    mProc = 20;
};
```

```
class grandDaughter : public daughter {
    private: double gPriv;
    public: void gFoo ( );
};
```

```
int main() {
    /*….*/
}
```

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission
- However, there are exceptions for
  - constructor and destructor
  - operator=() member
  - friends

Since all these functions are class-specific

Constructor Rules for Derived Classes

The default constructor and the destructor of the base class are always called when a new object of a derived class is created or destroyed.

```
class A {
  public:
    A ( )
      {cout<< "A:default"<<endl;}
    A (int a)
      {cout<<"A:parameter"<<endl;}
};
```

```
class B : public A
{
  public:
    B (int a)
        {cout<<"B"<<endl;}
};
```

```
B test(1);
```

output:

```
A:default
B
```

Constructor Rules for Derived Classes

You can also specify an constructor of the base class other than the default constructor

```
DerivedClassCon ( derivedClass args ) : BaseClassCon ( baseClass args )
    {  DerivedClass constructor body   }
```

```
class A {
  public:
    A ( )
      {cout<< "A:default"<<endl;}
    A (int a)
      {cout<<"A:parameter"<<endl;}
};
```

```
class C : public A {
  public:
    C (int a) : A(a)
        {cout<<"C"<<endl;}
};
```

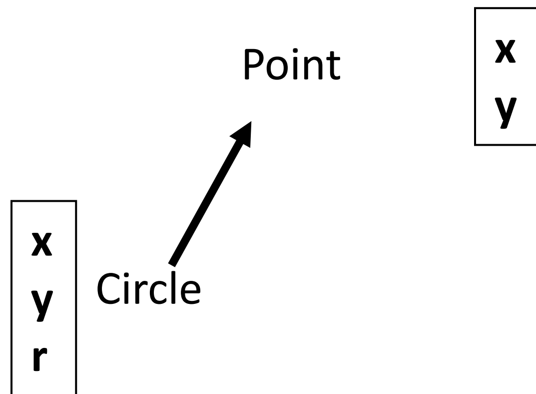
```
C test(1);
```

output:

```
A:parameter
C
```

Define its Own Members

The derived class can also define its own members, in addition to the members inherited from the base class




```
class Circle : public Point{
    private:
        double r;
    public:
        void set_r(double c);
};
```

```
class Point{
    protected:
        int x, y;
    public:
        void set(int a, int b);
};
```

```
class Circle{
    protected:
        int x, y;
    private:
        double r;
    public:
        void set(int a, int b);
        void set_r(double c);
};
```

Even more …

- A derived class can override methods defined in its parent class. With overriding,
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

```
class A {
  protected:
    int x, y;
  public:
    void print ()
          {cout<<"From A"<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
          {cout<<"From B"<<endl;}
};
```

Access a Method

```
class Point{
    protected:
       int x, y;
    public:
       void set(int a, int b)
            {x=a; y=b;}
       void foo ();
       void print();
};
```

```
class Circle : public Point{
  private:  double r;
  public:
     void set (int a, int b, double c) {
         Point :: set(a, b); //same name function call
         r = c;
     }
     void print();  };
```

```
Point A;
A.set(30,50);  // from base class Point
A.print(); // from base class Point
```

```
Circle C;
C.set(10,10,100);   // from class Circle
C.foo ();  // from base class Point
C.print(); // from class Circle
```

Polymorphism – An Introduction

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance

- Allows run-time interpretation of object type for a given class hierarchy
  - Also Known as "Late Binding"

- Implemented in C++ using virtual functions

Static Binding

- When the type of a formal parameter is a parent class, the argument used can be:

  the same type as the formal parameter,

  or,

  any derived class type.

- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter

- When pass-by-value is used, static binding occurs

Dynamic Binding

- Is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference

Virtual Functions

- Virtual Functions overcome the problem of run time object determination
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- How ?
  - Define a virtual function in the base class. The word virtual appears only in the base class
  - If a base class declares a virtual function, it must implement that function, even if the body is empty
  - Virtual function in base class stays virtual in all the derived classes
  - It can be overridden in the derived classes
  - But, a derived class is not required to re-implement a virtual function. If it does not, the base class version is used

Abstract Classes & Pure Virtual Functions

- Some classes exist logically but not physically.
- Example : Shape
  - `Shape s; // Legal but silly..!! : "Shapeless shape"`
  - Shape makes sense only as a base of some classes derived from it. Serves as a "category"
  - Hence instantiation of such a class must be prevented

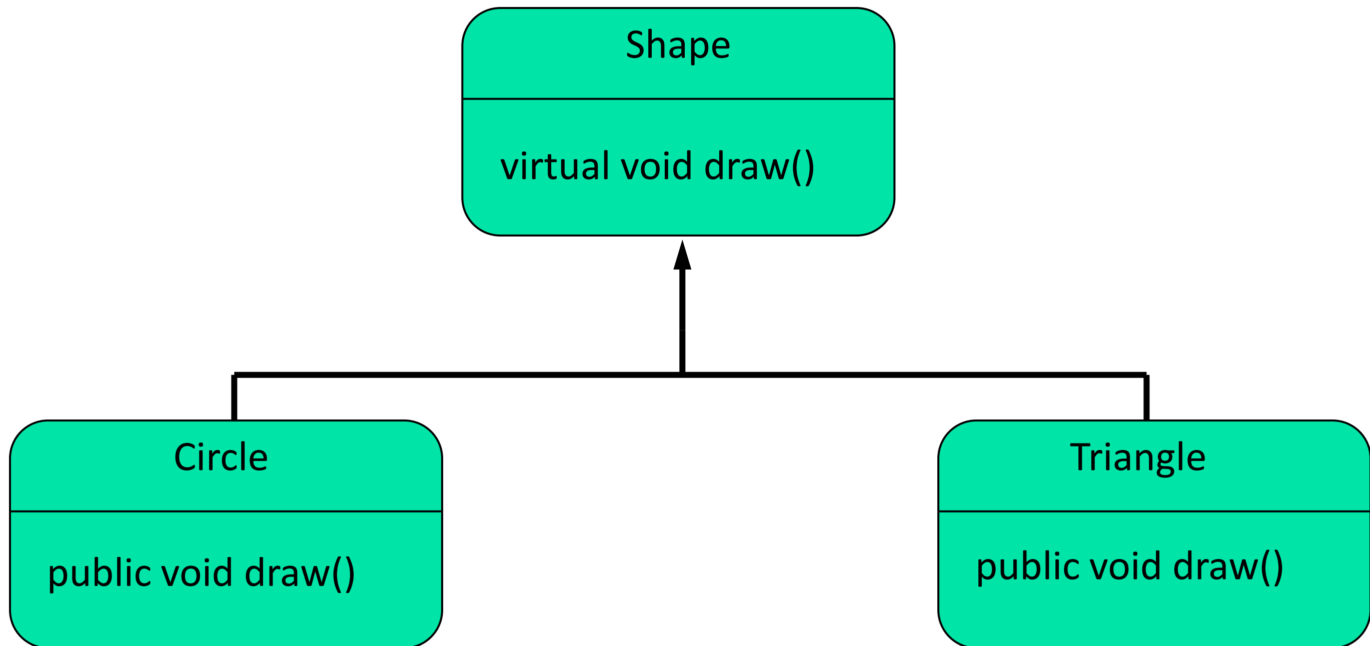
```
class Shape    //Abstract
{
  public :
  //Pure virtual Function
  virtual void draw() = 0;
}
```

A class with one or more pure virtual functions is an Abstract Class

Objects of abstract class can't be created

```
Shape s; // error : variable of an abstract class
```

Example



- A pure virtual function not defined in the derived class remains a pure virtual function.
- Hence derived class also becomes abstract

```
class Circle : public Shape { //No draw() - Abstract
   public :
   void print(){
     cout << “I am a circle” << endl;
   }
class Rectangle : public Shape {
   public :
   void draw(){ // Override Shape::draw()
     cout << “Drawing Rectangle” << endl;
   }
```

```
Rectangle r; // Valid
Circle c; // error : variable of an abstract class
```

Thank You