EEE102

C++ Programming and Software Engineering II

Lecture 8 Friendship and Inheritance

Dr. Rui Lin/Dr.Fei Xue

Rui.Lin/Fei.Xue@xjtlu.edu.cn

Room EE512/EE222

Outline

- 1. Friend
  - Friend functions and Friend Class
  - Friend function and operator overloading

- 2. Inheritance
  - Class and subclass
  - Inheritance
    - Specifiers
    - Access specifier and inheritance specifier
    - Example for public/protected/private inheritance
    - What can be inherited
    - Class composition and inheritance

1. Friend

- In principle, **`non-public`** members of a class cannot be accessed from outside the same class in which they are declared.
  - Example: the Point class

```
double x=p1.x-p2.y;
double y=p1.x-p2.y;
```

X

```
double x=p1.GetX()-p2.GetX();
double y=p1.GetY()-p2.GetY();
```

√

- Friends are functions or classes declared with the **`friend`** keyword.
  - Advantages: convenience for data sharing, improving the program's readability and running speed
  - Disadvantages: violation to data hiding

- Types of friendship
  - `friend` functions
  - `friend` classes

1.1 Friend functions

- To declare an external function as friend of a class, declaring a prototype of this external function within the class, and preceding it with the keyword **`friend`**
  - Syntax:

```
class ClassB
{
        friend returnType functionName (parameter list);
public:
        . . . . . .
private:
        . . . . . .
};
returnType functionName (parameter list)
{       . . . . . .
}
```

```
class Point
{
   friend double fDistance(Point p1, Point p2);
    public:
         Point(int xx=0, int yy=0)
         {       x=xx;y=yy;}
         int GetX()      {       return x;}
         int GetY()      {       return y;}
    private:
         int x,y;
};
```

```
double fDistance(Point p1, Point p2)
{
        double x=p1.x-p2.x;
        double y=p1.y-p2.y;
        double len=sqrt(x*x+y*y);
        return len;
}
```

```
double Distance(Point p1, Point p2)
{
        double x=p1.GetX()-p2.GetX();
        double y=p1.GetY()-p2.GetY();
        double len=sqrt(x*x+y*y);
        return len;
}
```

```
int main()
{
        Point myp1(1,1), myp2(4,5);
        cout <<"Distance (friend): "<<fDistance(myp1,myp2) <<endl;
        cout <<"Distance (non-friend): "<<Distance(myp1,myp2) <<endl;
        return 0;
}
```

1.2 Friend Class

- To declare a classA as a friend of classB, declaring the friendship within the classB, and preceding it with the keyword **`friend`**
  - Syntax:

```
class ClassB
{
        friend class ClassA;
public:
        . . . . . .
private:
        . . . . . .
};
```

  - Now, all the members of classA can use all the members of classB
- Notice:
  - Friendship cannot be passed (not-transitive);
  - Friendship is one-way.

Friend Class Example

```
class Line;    // declare Line class
class Point
{
   friend class Line;
   public:
        Point(int xx=0, int yy=0)
        {       x=xx;y=yy;}
        int GetX()      {return x;}
        int GetY()      {return y;}
   private:
        int x,y;
};
```

```
#include "Point.h"
class Line
{
public:
        Line(Point xp1, Point xp2):p1(xp1),p2(xp2)
        {
                double x=p1.x-p2.x;
                double y=p1.y-p2.y;
                len=sqrt(x*x+y*y);
        }
        double GetLen()
        {       return len;}
private:
        Point p1,p2;
        double len;
};
```

```
int main()
{
        Point xp1(1,1), xp2(4,5);
        Line line1(xp1,xp2);
        cout<<"Length: "<<line1.GetLen();
        return 0;
}
```

1.3 Friend function and operator overloading

- Question: How many operands ?
- Two ways to define operator overloading:
  - 1. Define as member functions (lect 5, pp19-21)
  - Example: addition in **`complexClass`**
    - One operand – addend.
    - Syntax for definition:

```
complexClass complexClass::operator + (complexClass a)
{
        . . . . . .
}
```

addend

    - Syntax for use:

```
summand + addend   <=>   summand.operator+(addend)
```

1.3 Friend function and operator overloading

- Two ways to define operator overloading:
  - 2. Define as a friend function to class;
    - Two operands – summand & addend.
    - Syntax for definition:

```
Complex operator +(Complex s, Complex a)
{
        . . . . . .
}
```

summand *addend*

    - Syntax for use:

```
summand + addend   <==>   operator+(summand, addend)
```

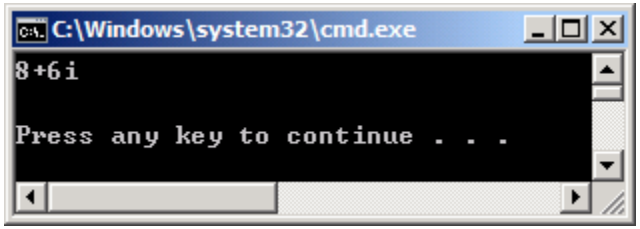
```
class complexClass
{

public:
    complexClass operator +(complexClass a)
    complexClass(double r=0,double i=0)     {real=r;img=i;};
    void display();
private:
    double real;
    double img;
};
```

```
complexClass complexClass::operator +(complexClass a)
{   complexClass temp(real+a.real,img+a.img);
    return temp;
}
```

Method 1: member function

```
int main()
{   Complex a(3,2),b(5,4),result;
    result=a+b;
    result.display();cout <<endl;
    return 0;}
```



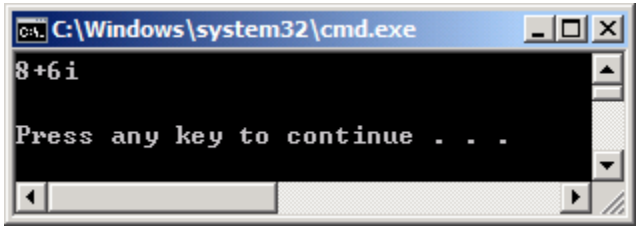
```
class complexClass
{
    friend complexClass operator +(complexClass a, complexClass b);
public:

    complexClass(double r=0,double i=0)     {real=r;img=i;};
    void display();
private:
    double real;
    double img;
};
```

```
complexClass operator +(complexClass s, complexClass a)
{   complexClass temp(s.real+a.real, s.img+a.img);
    return temp;
}
```

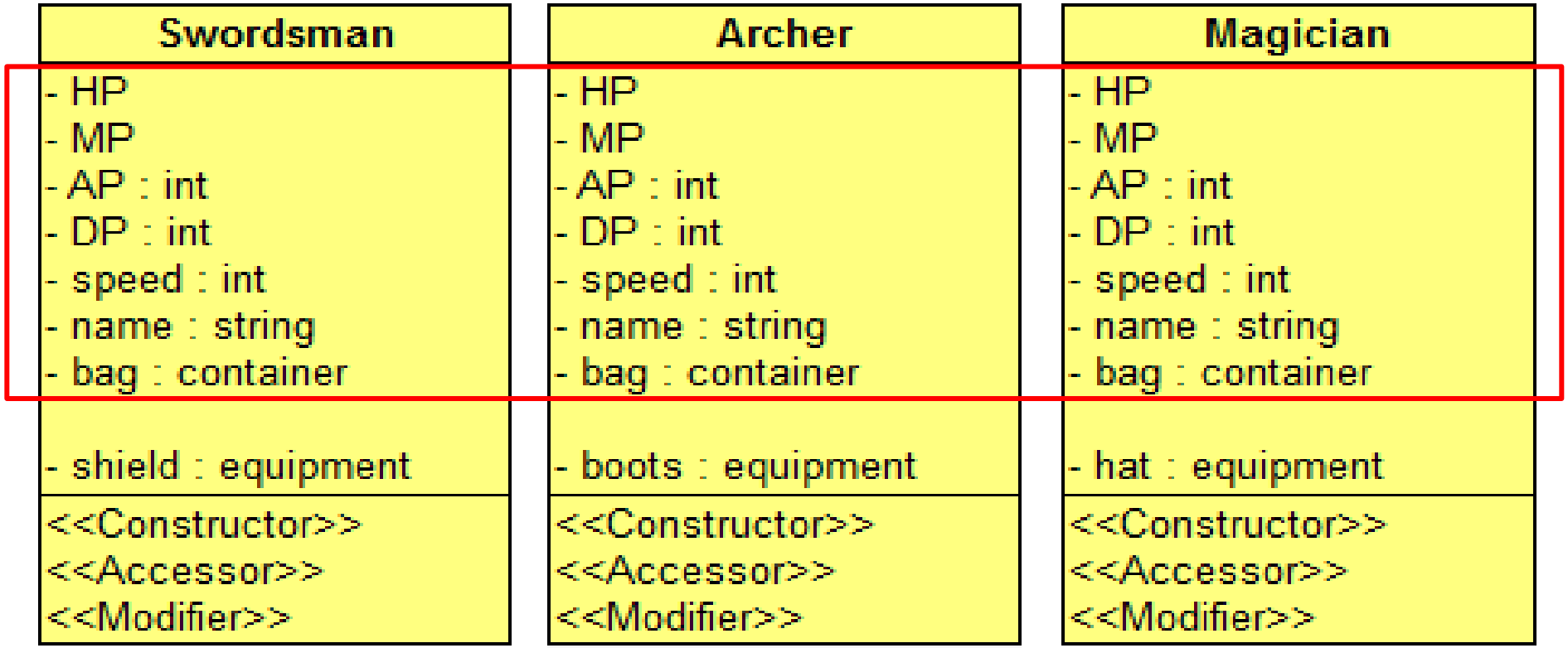
Method 2:
friend function

```
int main()
{   Complex a(3,2),b(5,4),result;
    result=a+b;
    result.display();cout <<endl;
    return 0;}
```



2. Inheritance

- WHY SUB-CLASSES ARE NEEDED?

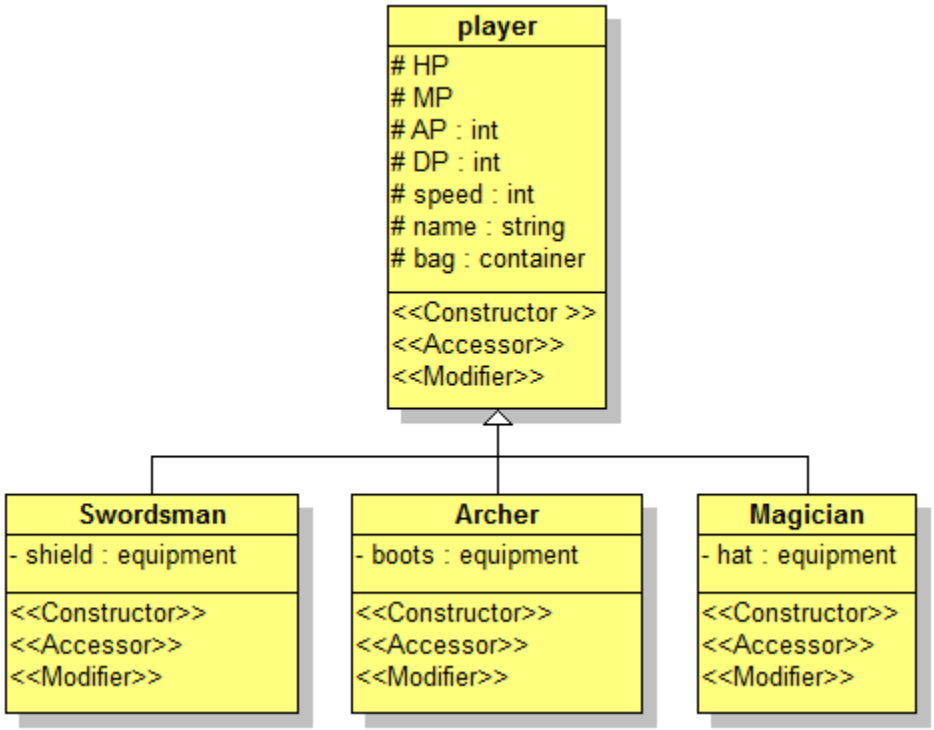


Swordsman Archer Magician

2. Inheritance

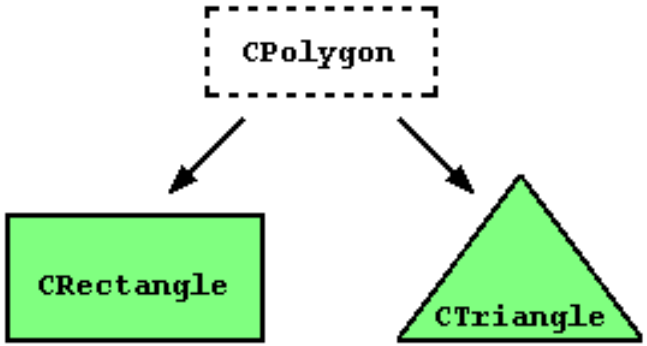
- In reality, there are many objects which have many similar properties and behaviour, but also differ in certain aspects.
- If defining a class for each of the objects, it would be very inefficient from a programming point of view.

- Ideally we want to group all the similar properties and behaviour into an identity and we only need to define once.
- We can then use the defined identity to add more specific feature to form a class that we really need.
- This is the idea of sub-class.



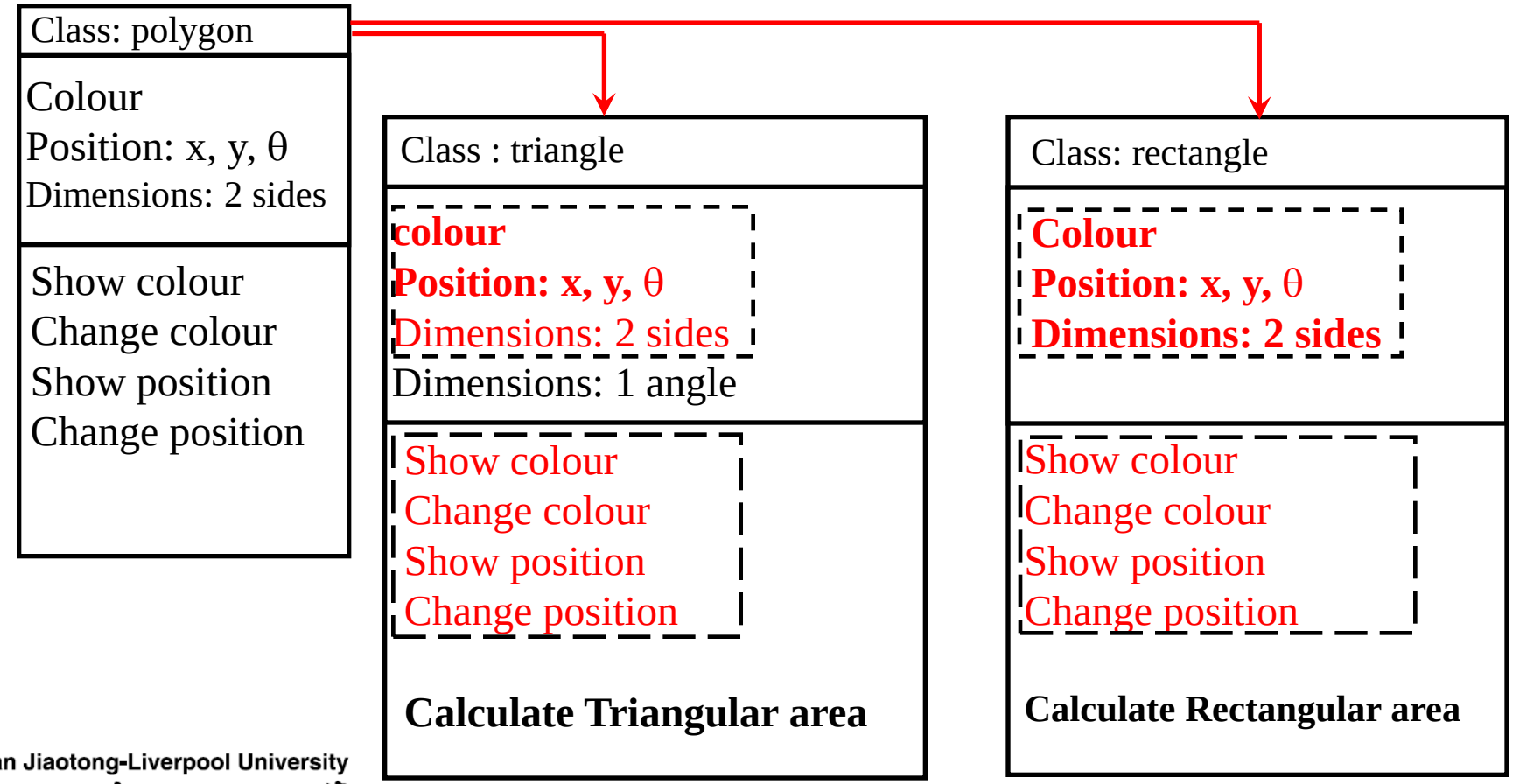
2.1 Class and subclass

- A base class is also called a parent class.
  - A base class defines the common features of a group of objects, such as all primitive shapes.
- A sub-class is also called a child (or derived) class.
  - A sub-class has more specific properties and methods.
  - Example:
    - A rectangle will be a sub-class of polygon.
    - A triangle will also be a sub-class of polygon.



Base Class (Parent Class) and Sub-class (Child Class)

- Data and function members of a parent class (enclosed by the broken boxes) could be used by a child class (under conditions defined later on) without re-defining them in the child class.



Class composition and inheritance

- Question: What is the difference between class composition and inheritance?
  - 1. Class composition: some data members of classA are objects of classB. Such as:
    - A line **has** two points;
    - A quadrilateral **has** three sides and three angles.
  - 2. Inheritance: subclass is derived from base class, belongs to the category of the base class. Such as:
    - A rectangular **is** a quadrilateral.
    - An undergraduate is a student.
  - Make a simple sentence to help your clarify the logic between these concepts.

2.2 Inheritance

- Inheritance allows to create classes (subclasses) which are derived from other classes (base classes), so that they automatically include some of its "parent's" members, plus its own.
- Syntax:

```
class derived_class: specifier base_class
```

  – Example:

```
class CPolygon
{ . . . };
class CTriangle : public CPolygon
class CRectangle : private CPolygon
```

2.2.1 Specifiers

- The access specifier limits the accessible level for the members of a class:

`private`:

– Members cannot be accessed outside an object of the class

`protected`:

– Members cannot be accessed outside an object of the class, but can be inherited by a derived class.

`public`:

– Members can be accessed outside an object of the class and inherited by a derived class.

2.2.2 Access specifier and inheritance specifier

- The inheritance specifier limits the most accessible level for the members inherited from the base class:

1. *Access specifier* : determines the access type to the names that follow it, such as:

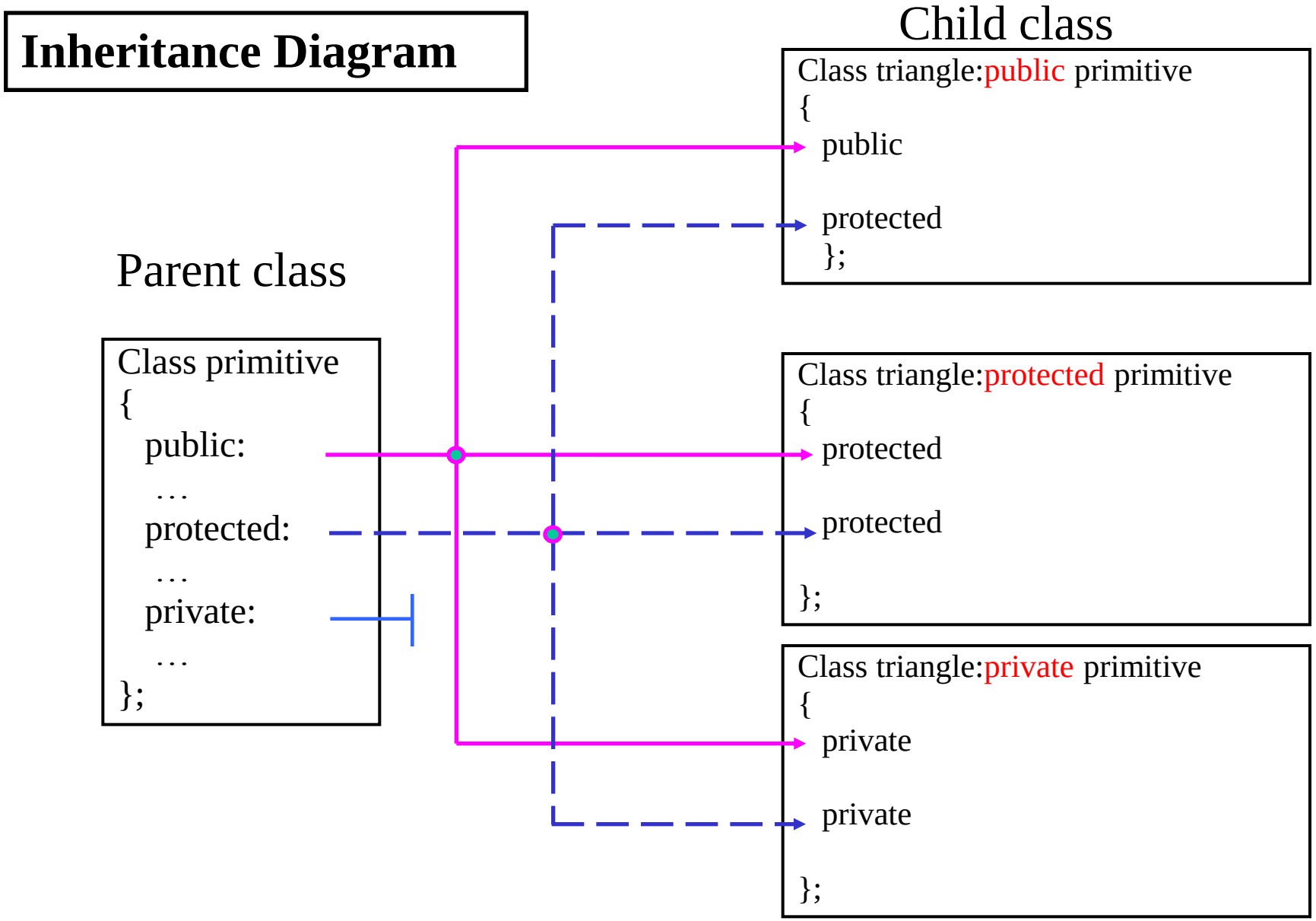
```
class className
{  public:
                .....
   private:
                .....
   protected:
                                .....
};
```

2. *Inheritance specifier* : determines the inherit type of the sub-class, such as:

```
class subclassName: public baseclassName
{
        .....
        .....
        .....
};
```

Can also be: private & protected.

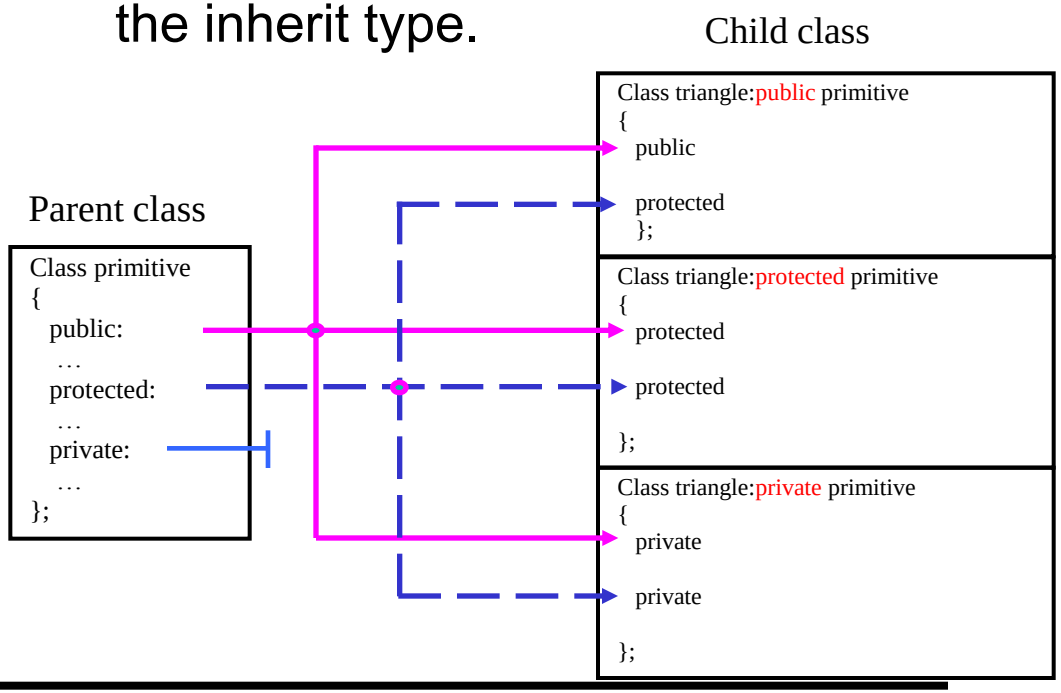
Inheritance Diagram



1. *Access specifier*: determines the access type to the names that follow it, such as:

```
class className
{   public:
                .....
    private:
                .....
    protected:
                .....
};
```

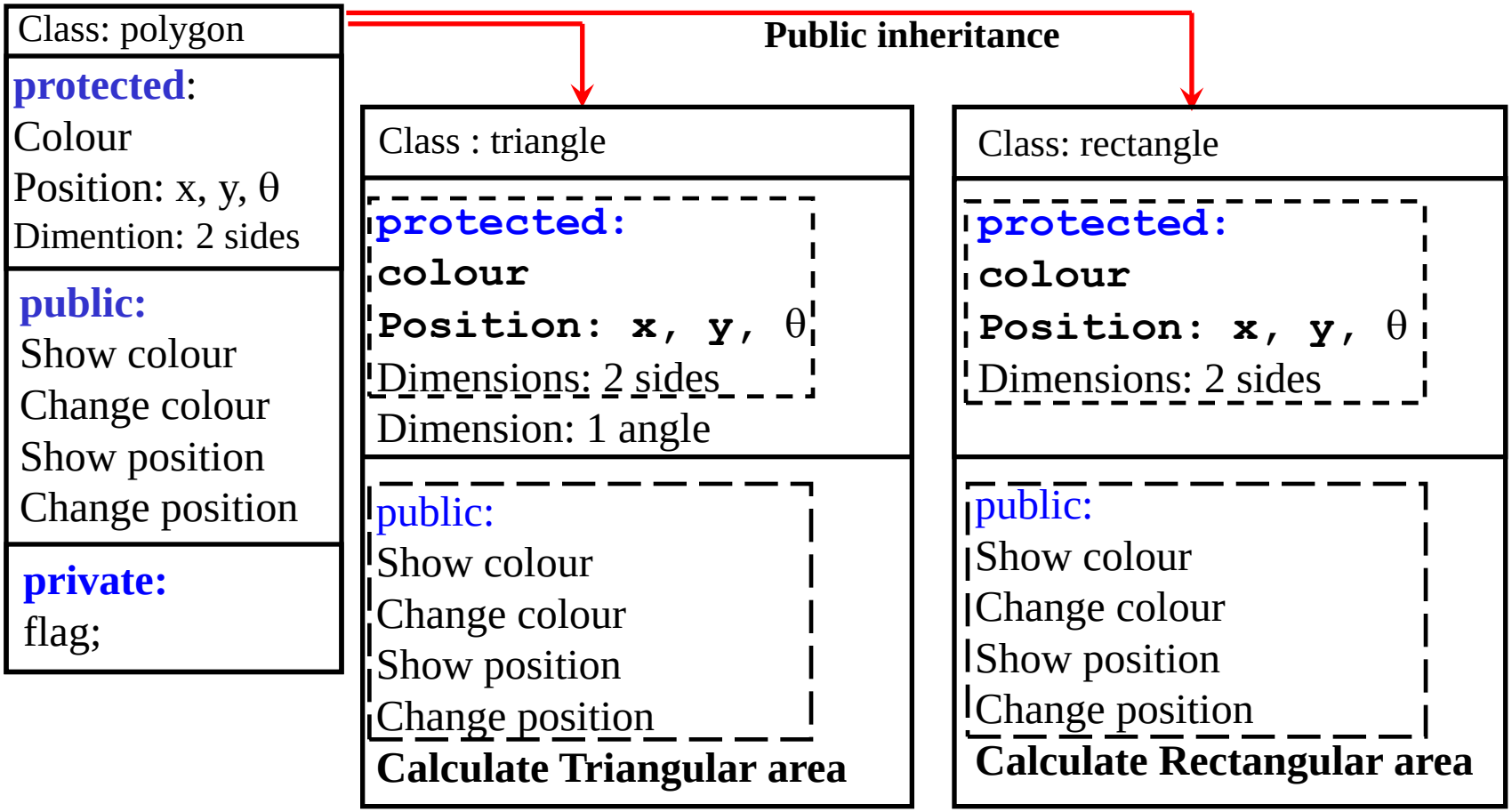
2. *Inheritance specifier*: determines the inherit type.



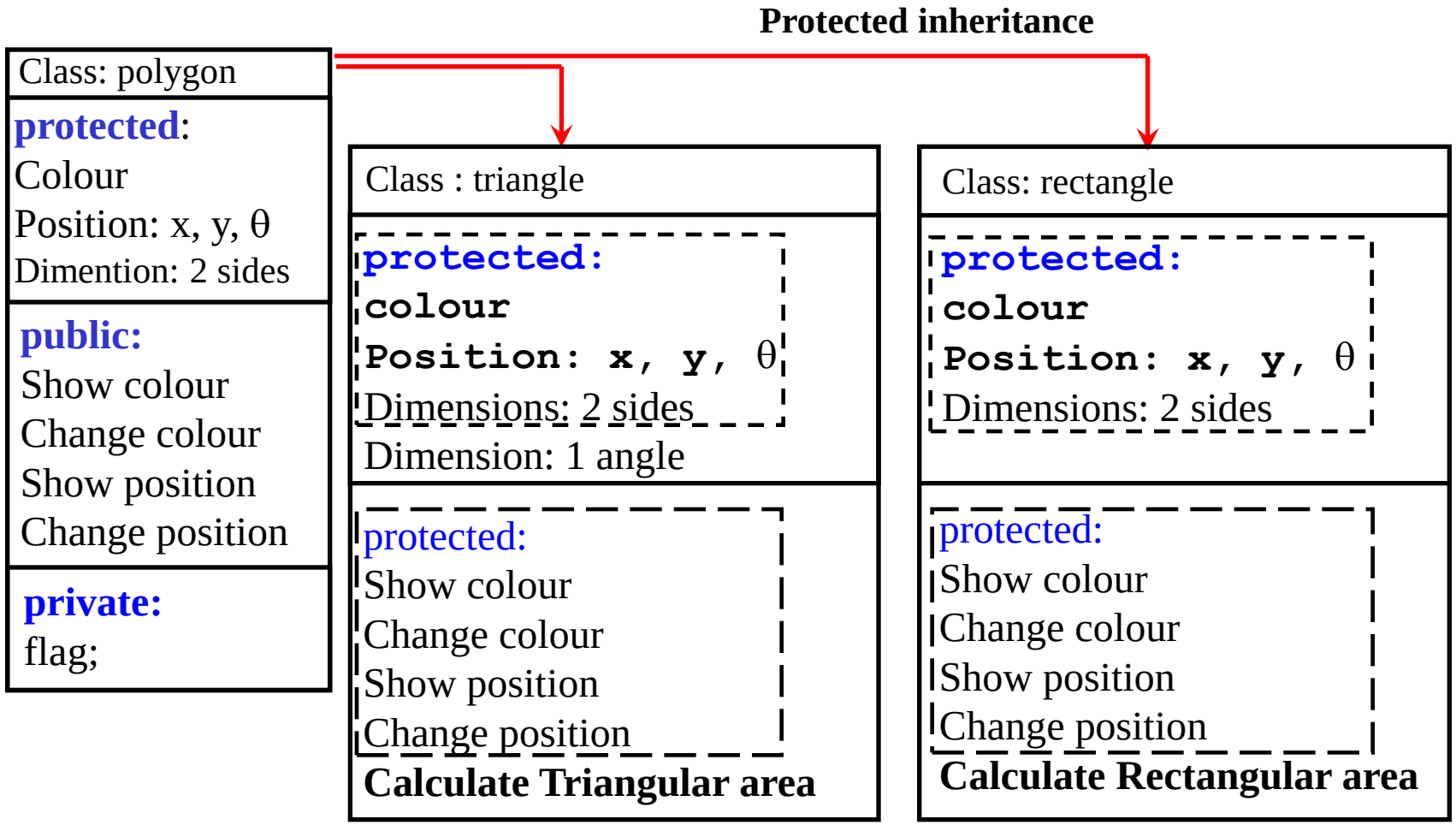
Access From / Access Specifier	Outside	Sub-class	Itself
public	√	√	√
protected	X	√	√
private	X	X	√

Inherit by / Access Specifier	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	X	X	X

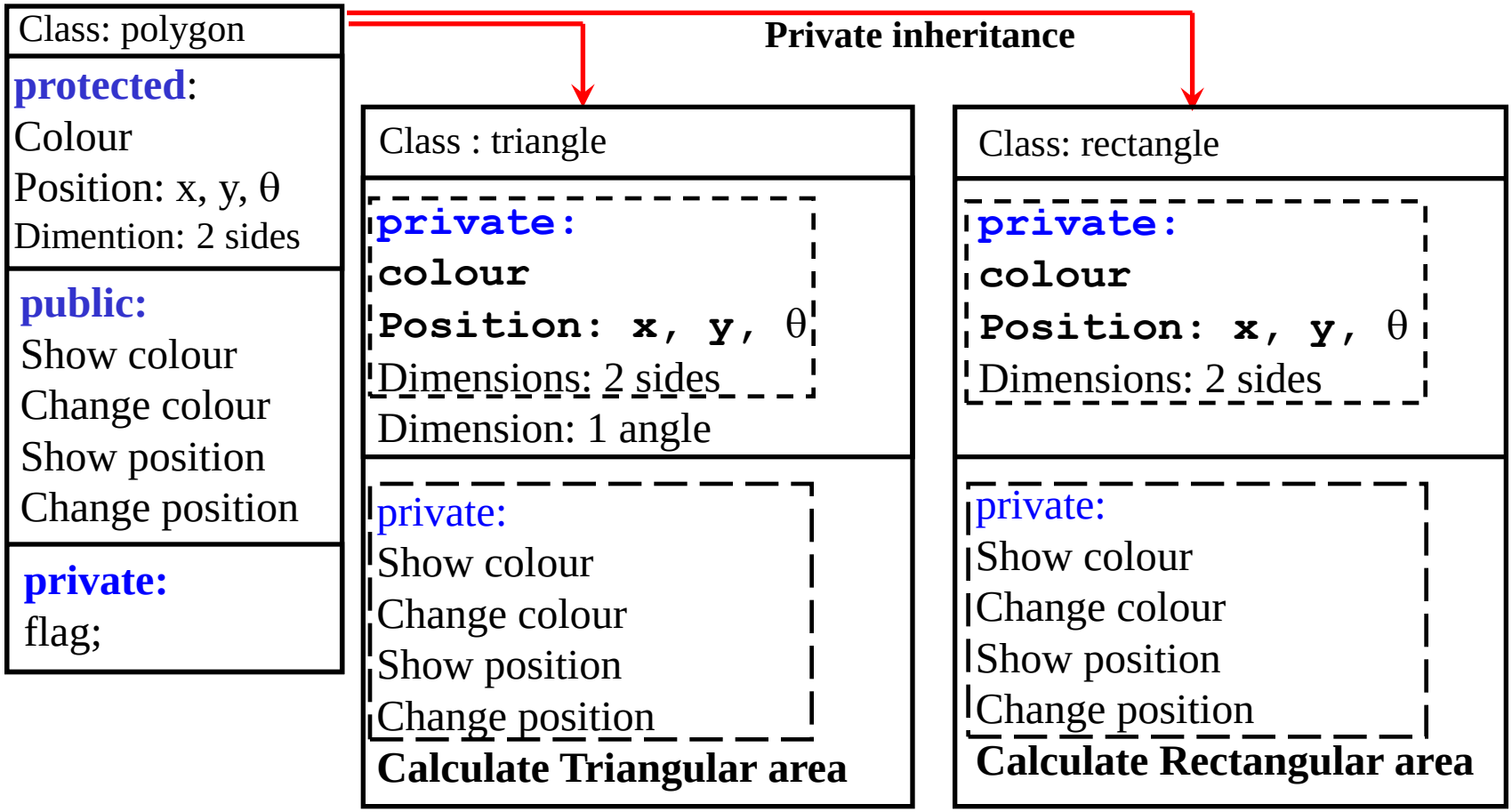
2.2.3 Example for `public` inheritance



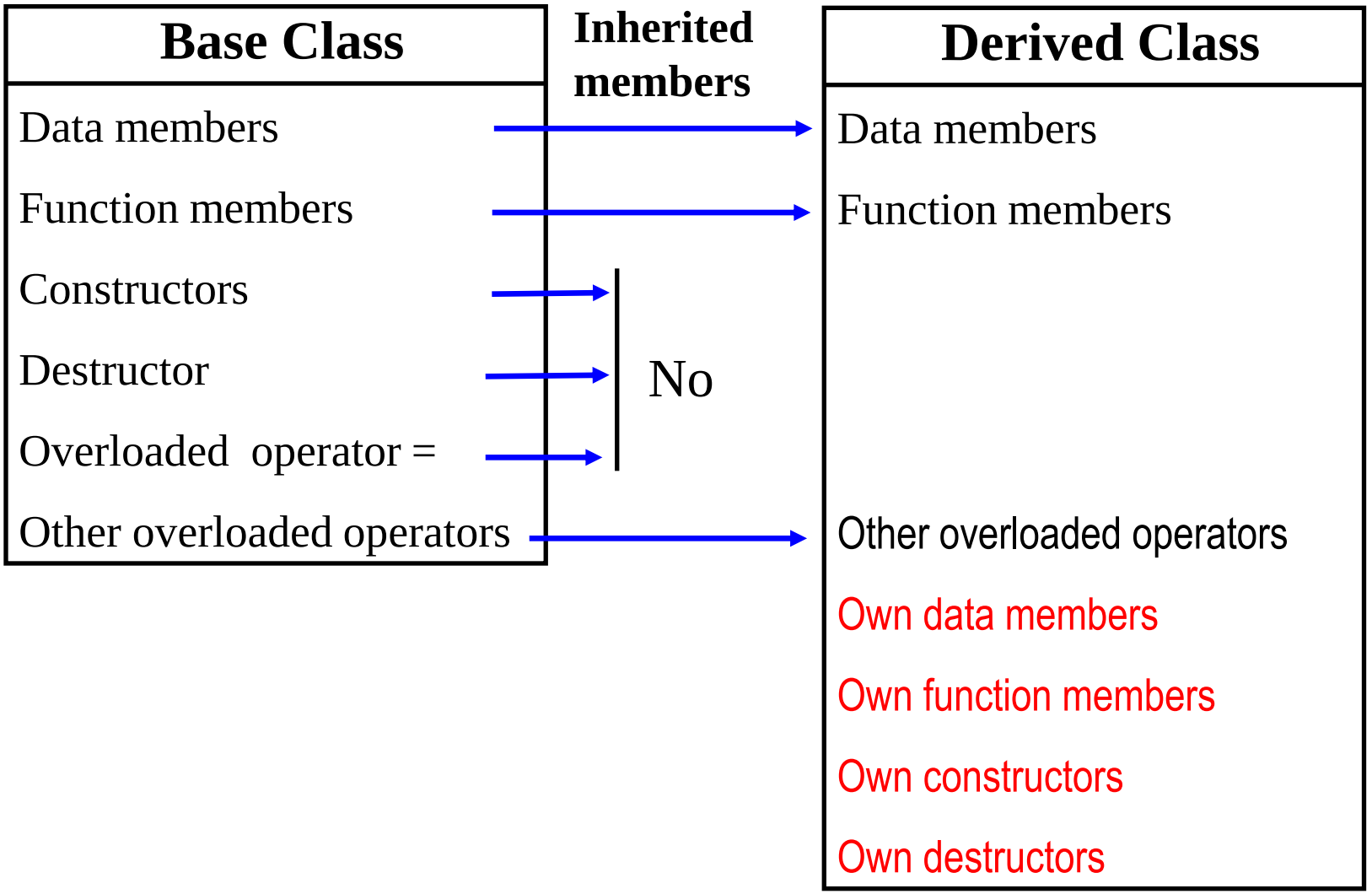
2.2.3 Example for **`protected`** inheritance



2.2.3 Example for **`private`** inheritance



2.2.4 What can be inherited



BASE CLASS

```
class CPolygon {
protected:
        int side1, side2;
public:
        void set_values (int a, int b)
        { side1=a; side2=b;}
};
```

```
int main () {
        CRectangle rect;
        CTriangle trgl;
        rect.set_values (4,5);
        trgl.set_values (4,5);
        trgl.set_angle (30);
        cout << rect.area() << endl;
        cout << trgl.area() << endl;
        return 0;
}
```

SUB-CLASS

```
class CRectangle: public CPolygon
{
public:
        int area ()
        { return (side1 * side2); }
};
```

Inheritance specifier

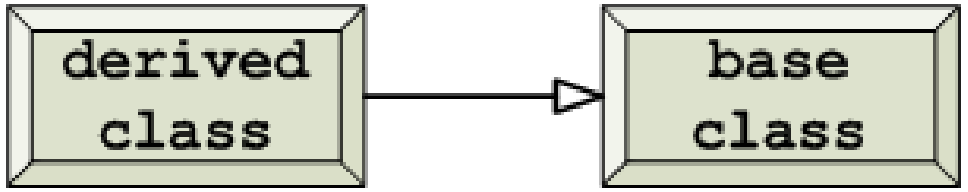
```
class CTriangle: public CPolygon
{
protected:
        double angle;
public:
        void set_angle (double ang)
        { angle = ang/180*3.14; }
        double area ()
        { return
(side1*side2*sin(angle)/2); }
};
```

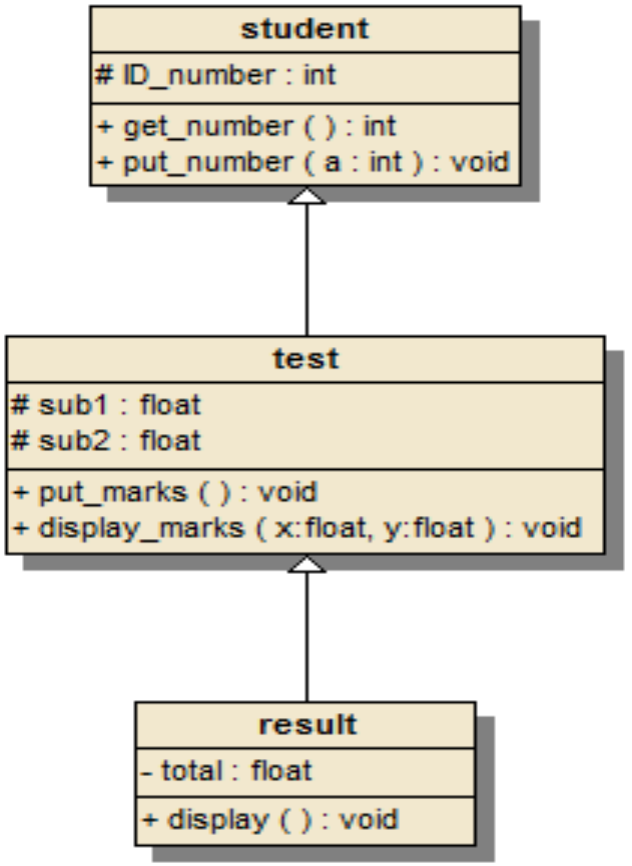
Inheritance specifier

Members (data member and function member) defined in base class can be directly used from derived class !

2.2.5 Class Hierarchy Chart

- To draw the hierarchy chart illustrating the inheritance relationship following UML rules, white arrow like is used to point from derived class to base class





2.2.5 Class Hierarchy Chart