Inheritance/access control

Members of a class can be	A base class can be inherited as	
• public • protected • private **class** B { **public**: **protected**: **private**: }	• public • protected • private Definition of **D**erived class D class D: public B{…} class D: protected B{…} class D: private B{…}	• In Java • not in Java • not in Java

Some base class members are not inherited to the derived class:

- private members
- constructors, destructor
- operator=

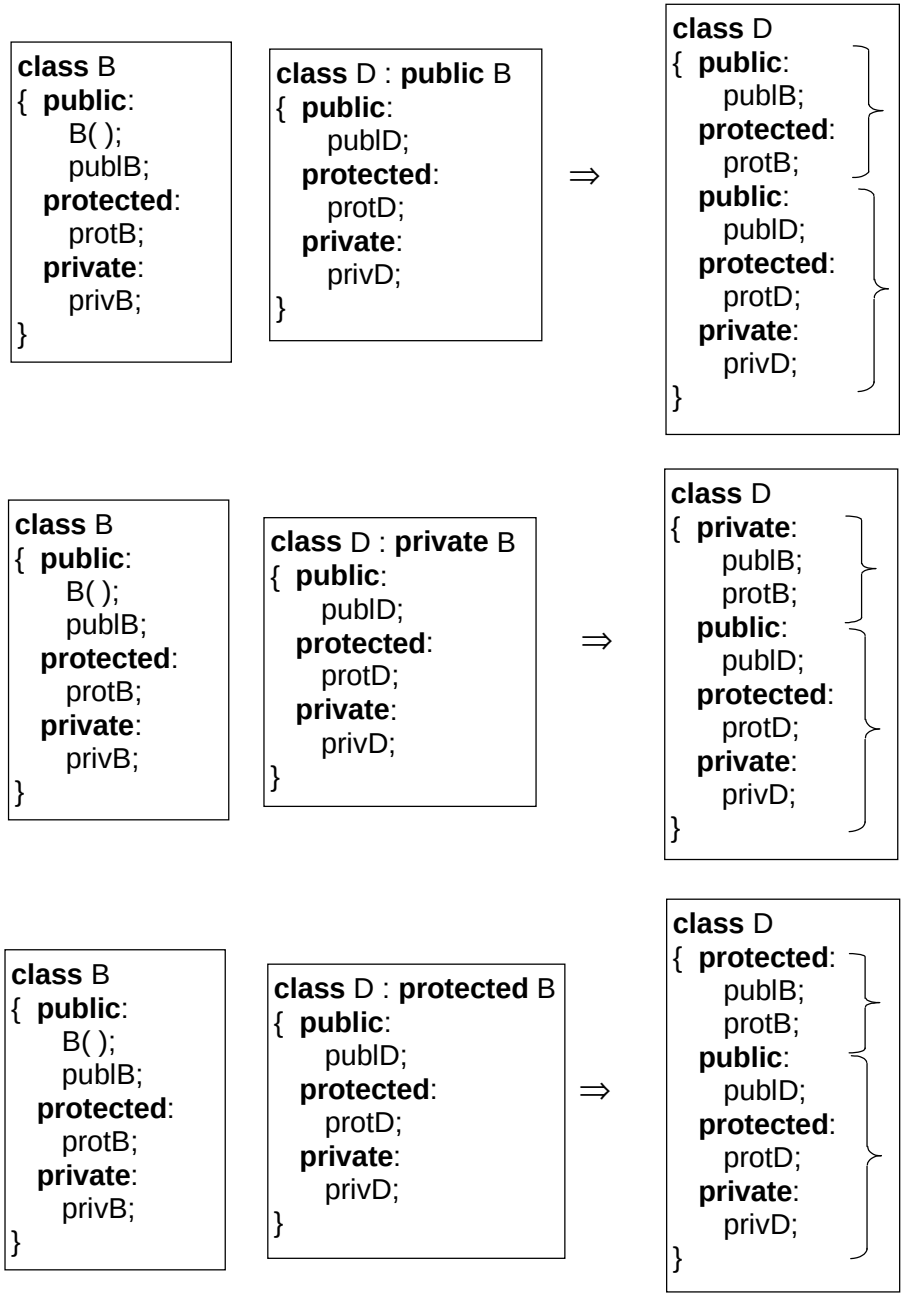
Members of B are inherited to the derived class D as public or protected as shown in the table below:

class B{	class D: public B{}	class D: protected B{}	class D: private B{}
public →	public	protected	private
protected →	protected	protected	private
private →	not visible	not visible	not visible

}

C++	Java
class D: public B{…}	class B extends B{…}

Class Derivation/ Inheritance, Access Control



Inheritance/access control

public derivation:

class D: **public** B{…}

Implements a subtype-relationship called

the *is-a* / *is-a-kind-of* relation or *interface inheritance*.
D is-a-kind-of B
D is a subtype of B
a D-object may be used as a B-object.

private derivation:

class D: **private** B{…}

May be used when the base class makes the derived class easier to write, and no base class members should be visible in the derived class. Example a queue-class D implemented by a list-class B, (but D could just as well have a B-object as a private member).

protected derivation:

class D: **protected** B{…}

As with private derivation, but access from further derived classes to the base class is desirable

Public Inheritance, student class

```
class student
{  public:
      enum year { fresh, soph, junior, senior, grad };
      student(char* nm, int id, double g, year x);
      void  print() const;

   protected:
      int   student_id;
      double   gpa;          // grade point average
      year y;
      char    name[30];
};

student ::student(char* nm, int id, double g, year x)
   :student_id(id), gpa(g), y(x)
   {  strcpy(name, nm);}         //from <cstring>

void student::print() const
   {  cout    << "\nstudent: " << name << " , " << student_id
              << " , " << y << " , " << gpa << endl;
   }
```

sophomore:
second year undergraduate
junior:
third year undergraduate

```
class grad_student : public student {
public:
   enum support { ta, ra, fellowship, other };
   grad_student(char* nm, int id, double g, year x,    support t, char* d, char* th);
   void  print() const;
protected:
   support  s;
   char     dept[10];
   char     thesis[80];
};
grad_student::grad_student
   (char* nm, int id,    double g, year x, support t, char* d, char* th)
   :  student(nm, id, g, x),
      s(t)
   {  strcpy(dept, d); strcpy(thesis, th);
   }

void grad_student::print() const
   {  student::print(); //base class info is printed
      cout << dept << " , " << s << '\n' << thesis << endl;
   }
```

a graduate student is a kind of student

print: overridden/ redefined

extra information for graduate student

call of base class constructor

Public Inheritance

student class(2)

```
#include <iostream>

//Test pointer conversion rules

#include  "student.h"       //include decls of student and graduate student

int main()
{ student  s("Mae Pohl", 100, 3.425, student::fresh);
  student* ps = &s;

  grad_student  gs("Morris Pohl", 200, 3.2564,student::grad,
                   grad_student::ta, "Pharmacy","Retail Pharmacies");
  grad_student* pgs;


  ps -> print();          //student::print
  ps = pgs = &gs;

  // now both ps      (type student*) and pgs  (type grad_student*)
  // points to a grad_student

  pgs -> print();         // grad_student::print

  ps ->print();           // student::print !!!
}
```

output:

```
student: Mae Pohl , 100 , 0 , 3.425

student: Morris Pohl , 200 , 4 , 3.2564
Pharmacy , 0
Retail Pharmacies

student: Morris Pohl , 200 , 4 , 3.2564
```

The method applied is determined from the pointer type,

not from the type of the object actually pointed to.

STATIC binding!

Different from Java!!

Public Inheritance, Virtual Functions

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

const double PI = 3.14159;

class shape {
  public:
    shape():x(0.0),y(0.0){}
    void setpos(double nx, double ny){x=nx;y=ny;} // common for all shapes

    virtual double  area() const { return 0; }
    virtual string get_name(){return " NOSHAPE ";}

  protected:
    double  x, y;    // common for all shapes
};
```

to be redefined for different shapes

```
class rectangle : public shape {
  public:
    rectangle(double h, double w): height(h), width(w) {}
    double  area() const { return (height * width); }
    string get_name() { return (" RECTANGLE "); }
  private:
    double  height, width;
};


class circle : public shape {
  public:
    circle(double r): radius(r) {}
    double  area() const { return (PI * radius * radius);}
    string get_name() { return (" CIRCLE "); }
  private:
    double   radius;
};
```

virtual attribute inherited to area & get_name

Public Inheritance, Virtual Functions

```
class polygon : public shape {
  public:
    polygon(int n, double s): number(n), side(s) {}
    double  area() const
        { return (side*side*number/(4.0*tan(PI/number)));}
    string get_name() { return (" POLYGON "); }
  private:
    int number; // of sides
    double side;
};
```

```
int main()
{ shape*     shps[]=
    {new shape(), new circle(2.2), new polygon(14,1.3), new
rectangle(5.2,3.0)};
  for(int i=0; i<4;++i)
    {cout<< shps[i]->get_name()<<"area= "<< shps[i]->area() <<endl;}
}
```

shape*

output:

```
NOSHAPE area= 0
 CIRCLE area= 15.2053
 POLYGON area= 25.9153
 RECTANGLE area= 15.6
```

The method applied is determined from the type of the object actually pointed to.

DYNAMIC binding! As in Java!!

Inheritance Polymorphism.

In C++ terminology: all Java Methods are virtual .

Public Inheritance, Abstract Base Class

```
class shape {
  public:
    shape():x(0.0),y(0.0){}
    void setpos(double nx, double ny){x=nx;y=ny;} // common for all shapes
    virtual double  area() const =0;
    virtual string get_name() const = 0;
  protected:
    double  x, y;    // common for all shapes
};
```

pure virtual functions

A class containing a pure virtual method is *an abstract base class*.

An abstract base class cannot be instantiated.

Java:

```
abstract class shape {
  public shape():{x=0.0;y= 0.0;}
  public void setpos(double nx, double ny){x=nx; y=ny;}
  public abstract double  area();
  public abstract string get_name();
  protected double  x;
  protected double  y;
};
```

Inheritance Polymorphism, Only via pointer and &

```
class shape {
  public:
    shape():x(0.0),y(0.0){}
    void setpos(double nx, double ny){x=nx;y=ny;} // common for all shapes
    virtual double  area() const {return 0;}
    virtual string get_name() const{return " NOSHAPE";}
  protected:
    double  x, y;    // common for all shapes
};
class rectangle : public shape {
  public:
    rectangle(double h, double w): height(h), width(w) {}
    double  area() const { return (height * width); }
    string get_name() { return (" RECTANGLE "); }
  private:
    double  height, width;
};
…
int main()
{ shape  shps[]=
    {circle(2.2), polygon(14,1.3), rectangle(5.2,3.0)};
  for(int i=0; i<3;++i)
    {cout<< shps[i].get_name()<<"area= "<< shps[i].area() <<endl;}
}
```

array of shapes !!, not shape*

shape

output:
NOSHAPE area= 0
NOSHAPE area= 0
NOSHAPE area= 0

Inheritance polymorphism only works for objects accessed via pointers or refences !!

Multiple Inheritance

Not in Java

```
class student
{ public:
    const int soc_sec;
    const string name;
    …
};
class worker
{ public:
    string name;
    …
};
class student_worker: public student, public worker
{ public:
    void print()
    { cout   << "ssn: " << soc_sec << "\n"
             << "name: " << student::name << endl;
      // << "name: " << name << endl; would be ambiguous !!
    }
…
};
```

```
class Interface{
  public: virtual type1 method1(…)=0; virtual type2 method2(…)=0
}
class BaseClass{
…
}
class MyClass: public BaseClass, public Interface
{…}
```

~ **Java**:	**class** MyClass **extends** BaseClass **implements** Interface {…}

Public Inheritance, Subtyping Form

```
class B //Base class
{  public:
      B(){};       // constructors
      B(…){}
      virtual ~B(){…;}    // virtual destructor, may be pure virtual
      virtual method1(…){…;}
      …
      virtual methodn(…)=0;
   protected:
      …
   public: //often empty
}
class D1: public B
{  public:
      D1(){}                  // constructors
      D1(…):B(…){…;}      // constructors
      ~D1(){…;}               // destructor
      method1(…){…;}      // special implementation of method1
      …
}
class D2: public B
{  public:
      …
}
B* x= new D1(…);
…
delete x;   // destructor execution: 1) ~D1(), 2) ~B()
```

Notice: if ~B() not virtual then ~D1() will not be executed!!!

Runtime Type Identification (RTTI)

```
class shape {
   public:
      shape():x(0.0),y(0.0){}
      void setpos(double nx, double ny){x=nx;y=ny;} // common for all shapes
      virtual double  area() const { return 0; }
      virtual string get_name(){return " NOSHAPE ";}
   protected:
      double  x, y;    // common for all shapes
};

…

class polygon : public shape {
   public:
      polygon(int n, double s): number(n), side(s) {}
      double  area() const
           { return (side*side*number/(4.0*tan(PI/number)));}
      string get_name() { return (" POLYGON "); }
      int polNumb(){return number;} // special polygon method ←
   private:
      int number; // of sides
      double side;
};


int main()
{  shape* shps[]=
      {new shape, new circle(2.2), new polygon(14,1.3), new rectangle(5.2,3.0)};
   for(int i=0; i < 4;++i)
      { polygon* pp= dynamic_cast<polygon*>(shps[i]); //down-casting!!
        if (pp){cout<< "polygon number of sides= " << pp->polNumb();}
      }
}
```

output:
polygon number of sides= 14