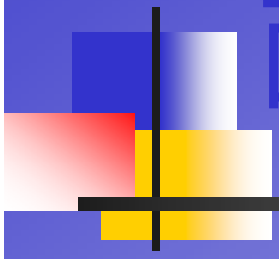
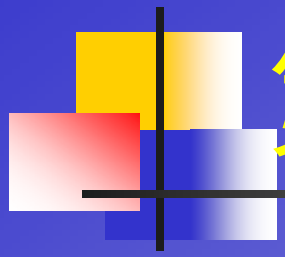


面向对象的程序设计

华中科技大学计算机学院
李瑞轩



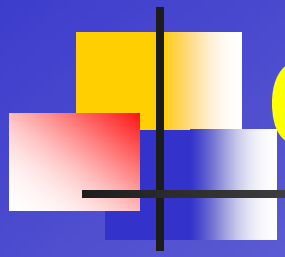
第六章 单继承类

本章内容：

- 6.1单继承类
- 6.2派生控制
- 6.3成员访问
- 6.4构造与析构
- 6.5父类和子类
- 6.6派生类的存储空间

6.1单继承类

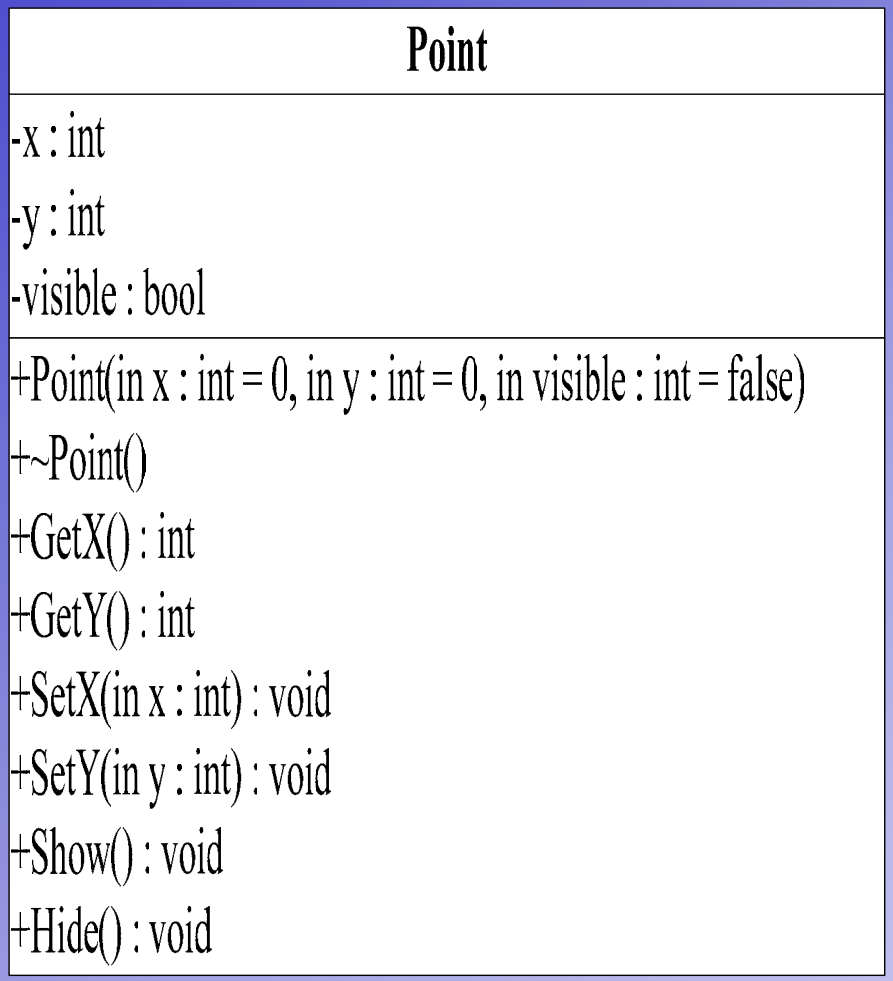
- 继承是C++类型演化的重要机制，常用来表示类属关系而不是构成关系，其实质是建造新的派生类，换句话说，继承就是创建一个具有别的类的属性和行为的新类的能力。
- 通过继承，一个类可以只定义新类只需定义原有类型没有的数据和函数成员，但是具有原有类的属性和行为。使得类之间具备了层次性。

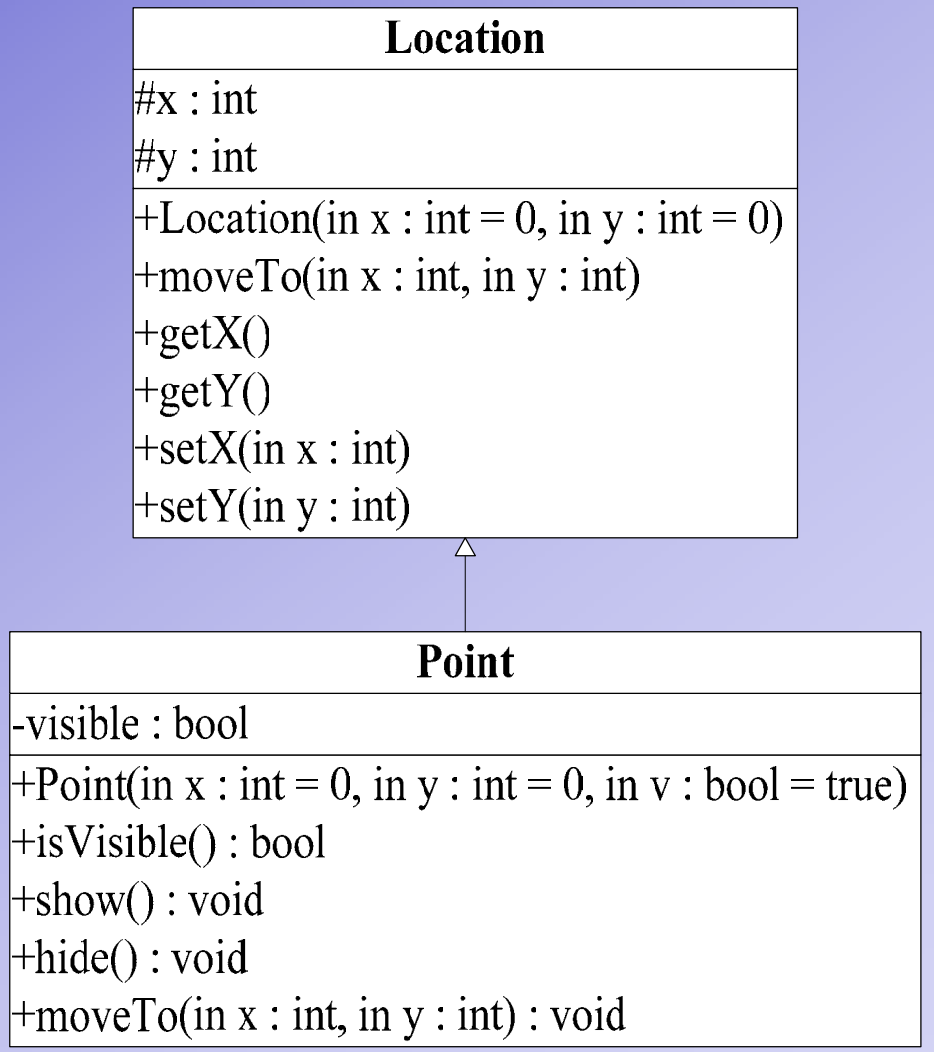


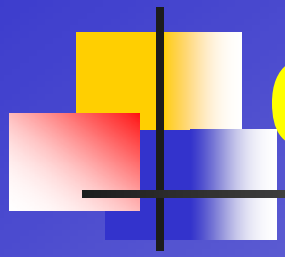
6.1单继承类

- 例子：在一个系统中，需要对点进行操作，点具有可见性，并具有x ,y坐标特性，点可以移动。

6.1单继承类


Point
-x : int
-y : int
-visible : bool
+Point(in x : int = 0, in y : int = 0, in visible : int = false)
+~Point()
+GetX() : int
+GetY() : int
+SetX(in x : int) : void
+SetY(in y : int) : void
+Show() : void
+Hide() : void

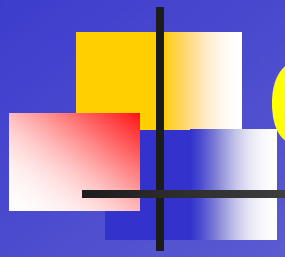

Location
#x : int
#y : int
+Location(in x : int = 0, in y : int = 0)
+moveTo(in x : int, in y : int)
+getX()
+getY()
+setX(in x : int)
+setY(in y : int)
Point
-visible : bool
+Point(in x : int = 0, in y : int = 0, in v : bool = true)
+isVisible() : bool
+show() : void
+hide() : void
+moveTo(in x : int, in y : int) : void



6.1单继承类

- 利用继承的优点：
- 层次分明；
- 可复用性好；

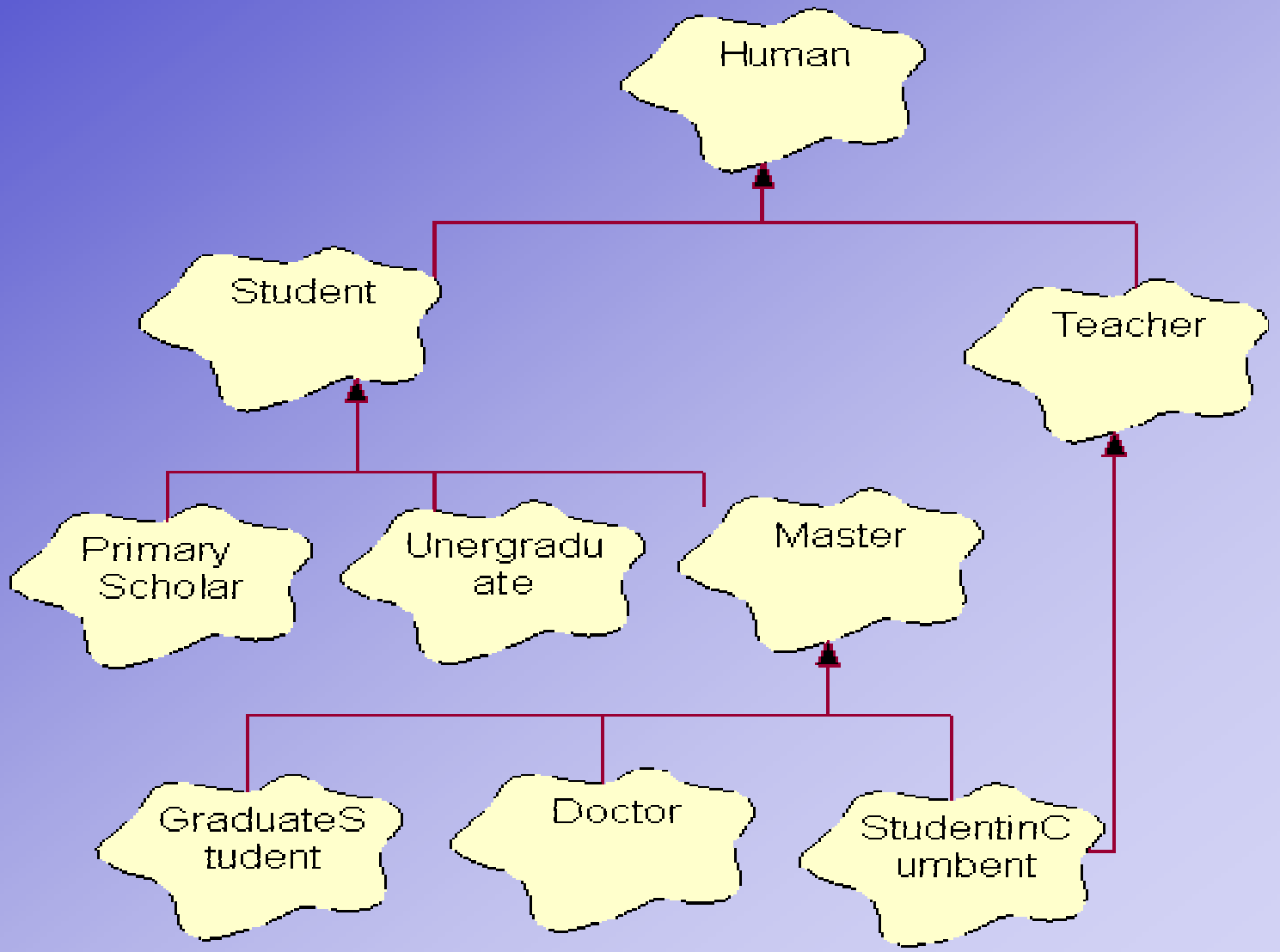
  便于维护性。



6.1单继承类

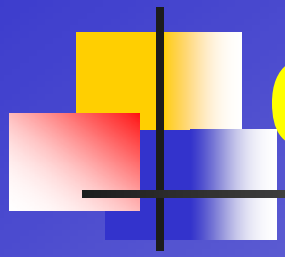
- 派生类：接受成员的新类，如上例中的Point类
- 基类：提供成员的类，如上例中的Location类
- 新类可以接受一个类提供的数据和函数成员，也可以接受多个类提供的数据和函数成员，这两种继承形式分别称为单继承和多继承。

6.1单继承类


Human
Student
Teacher
Primary Scholar
Unergraduate
Master
GraduateStudent
Doctor
StudentinCumbent

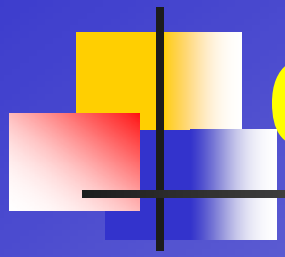
6.1单继承类

- 单继承的声明形式：
- **class** 派生类名称:访问控制符　基类名称
- {
- **private**：
- 成员说明列表;
- **protected:**
- 成员说明列表;
- **public**：
- 成员说明列表;
- }



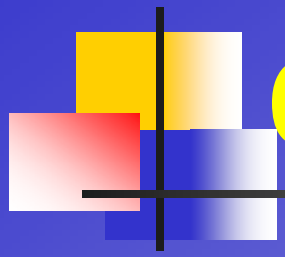
6.1单继承类

- 说明：
- 派生类名称是要定义的新类的名字；
- 基类名称是指从哪个类派生出来的；
- 访问控制符指当基类成员继承到派生类时，基类成员在派生类中的访问权限，其值可为private，public和protected（见6.2）；



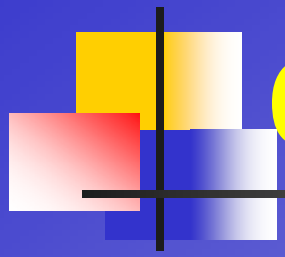
6.1单继承类

- 说明：
- 用class声明的类的派生控制缺省为private，因此，声明class POINT: private LOCATION等价于声明class POINT: LOCATION。
- 派生类也可以用struct声明，用class和struct声明的不同之处在于：用class声明的派生控制和访问权限缺省为private，用struct声明的派生控制和访问权限缺省为public。注意，用union声明的类既不能作基类，也不能作任何基类的派生类。



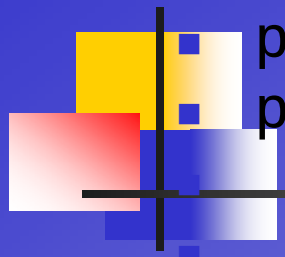
6.2派生控制

- 类的私有成员可以被类自身的成员和友元访问，但不能被包括派生类在内的其他任何类和任何普通函数访问；
- 类的保护成员除了可以被类自身的成员和友元访问外，还可以被派生类的函数成员访问，但是，类的保护成员不能被任何非友元的普通函数访问；
- 类的公有成员可以被任何普通函数和任何函数成员访问。



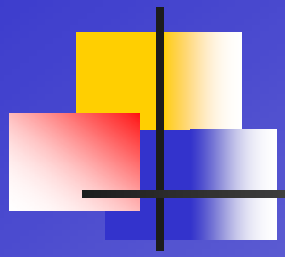
6.2派生控制

- 由此可知，派生类函数成员可以访问基类的保护和公有成员。除非定义为基类的成员友元，否则派生类函数成员不能访问基类的私有成员。
- 例题：指出下面程序的错误。

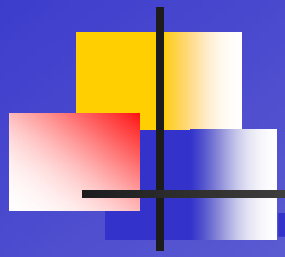


```
class LOCATION{
    int  x;
protected: int y;
public:
    int getx( ) { return x;      }
    int gety( ) { return y;      }
    void moveto(int x, int y)
      {
          LOCATION::x=x;
          LOCATION::y=y;
      }
    LOCATION(int x, int y)
      {
          LOCATION::x=x;
          LOCATION::y=y;
      }
    ~LOCATION( ){ }
};
```

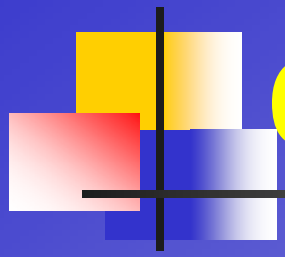
```
class POINT: public
LOCATION{
    int visible;
public:
    int  isvisible( ) { return
visible; };
void show( ){ visible=1; }
void hide( ){ visible=0;  }
void offset(int dx, int dy){
     int v=isvisible( );
     if(v) hide( );
     x += dx;
     y += dy;
    if(v) show( );
}
POINT(int x, int
y) :LOCATION(x, y)
{ visible=0; };
    ~POINT( ) { hide( ); };
};
```



```
void main(void)
{
    POINT p(3, 6);
    p.LOCATION::moveto(7, 8);
    p.x = 10;
     p.y = 20;
}
```



- 错误：
- void offset(int dx, int dy){
- int v=isvisible( );
- if(v) hide( );
- x += dx；//x为基类的私有成员，派生类不能访问
- y += dy;//y为基类的保护成员，派生类可以访问
- if(v) show( ); }
- void main(void)
- {
- POINT p(3, 6);
- p.LOCATION::moveto(7, 8);
- p.x = 10; //对象不能直接访问私有成员
- p.y = 20; //对象不能直接访问受保护成员
- }

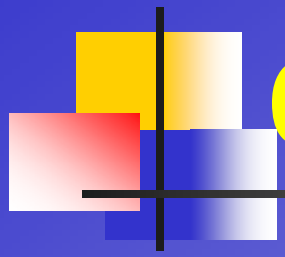


6.2派生控制

- 基类成员继承到派生类时，其访问权限的变化同派生控制有关。对于派生类继承的基类成员，其访问权限的变化情况如下表 ：

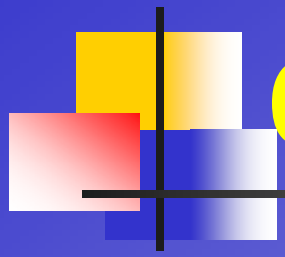
6.2派生控制

派生控制 / 基类成员	private	protected	public
protected	private	protected	protected
public	private	protected	public



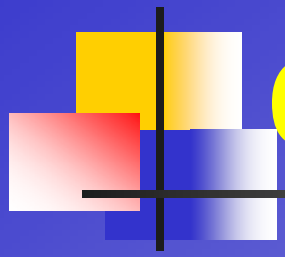
6.2派生控制

- 实际上，可以用一个更简洁的办法记忆上表。假定访问权限和派生控制满足private<protected<public，如果基类成员的访问权限高于派生控制，则派生后基类成员的访问权限和派生控制一样，否则，派生后基类成员的访问权限保持不变。由此可见，当派生控制为public时，基类的所有非私有成员派生后访问权限不变。



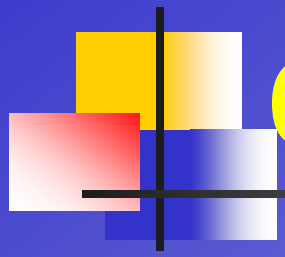
6.2派生控制

- 类POINT的派生控制为public，基类函数getx、gety派生后的访问权限仍为public对类POINT来说这是合理的，因为，对类POINT来说则类POINT需要这样的函数成员；但是，基类函数成员moveto派生后的访问权限为public对类POINT来说则是不合理的，因为类POINT自己定义了public函数成员moveto，在以下程序中，主函数还能调用基类函数成员LOCATION::moveto。



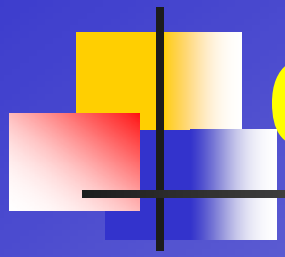
6.2派生控制

```
void main(void)
{
    POINT p(3, 6);
    p.moveto(9, 18);
     //访问POINT::moveto
    p.LOCATION::moveto(7, 8);
     //指定访问LOCATION::moveto
}
```



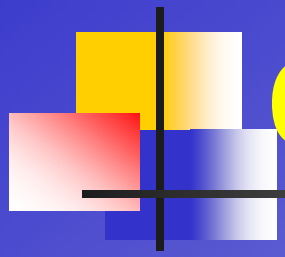
6.2派生控制

- 执行上述程序就会发现，点p从(9, 18)移动到(7, 8)后，屏幕上点的显示位置并未改变，因为LOCATION::moveto不能显示任何图形。为了防止main调用基类函数成员，POINT的派生控制应为private或protected。



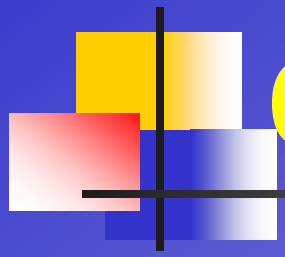
6.2派生控制

- class POINT: LOCATION{//class定义的派生类的派生控制缺省为private
- 　　int visible;
- public:
- 　　int isvisible( ) { return visible; };
- 　　void show( );
- 　　void hide( );
- 　　void moveto(int x, int y);
- 　　POINT(int x, int y) :LOCATION(x, y) { visible=0; };
- 　　~POINT(int x, int y) { hide( ); };
- };



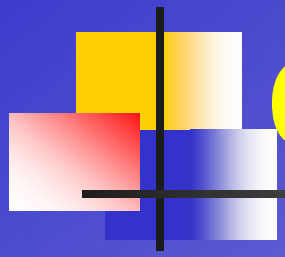
6.2派生控制

- C++提供了恢复基类成员访问权限的方法，恢复是将基类成员的访问权限复原，而不是改变成其他种类的访问权限。为了恢复基类getx和gety的访问权限public，类POINT的声明可修改如下。



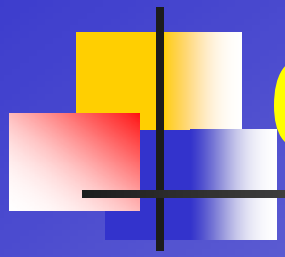
6.2派生控制

```
class POINT: private LOCATION{
    int visible;
public:
    LOCATION::getx;                 //恢复权限
    LOCATION::gety;                 //恢复权限
    int  isvisible( ) { return visible; };
    void show( );
    void hide( );
    void moveto(int x, int y);
    POINT(int x, int y) :LOCATION(x, y) { visible=0; };
    ~POINT( ) { hide( ); };
};
```



6.2派生控制

- 需要指出的是，选用private作派生控制通常不是最好的选择。如果派生类POINT选用private作派生控制，却又未恢复LOCATION::getx的访问权限，则getx在POINT类中的访问权限将变为private，从而使以POINT为基类的派生类无法访问private的POINT::getx。

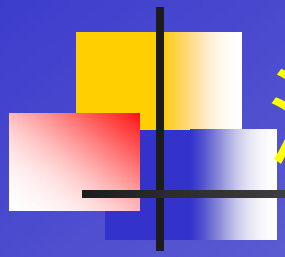


6.3 成员访问

- 标识符的作用范围可分为从小到大四种级别：
- ①作用于函数成员内;
- ②作用于类或者派生类内;
- ③作用于基类内;
- ④作用于虚基类内。
- 标识符的作用范围越小，被访问到的优先级越高。如果希望访问作用范围更大的标识符，则可以用类名和作用域运算符进行限定。

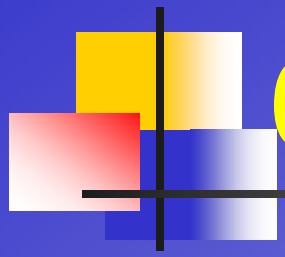
6.3 成员访问:同名成员

```
#include <iostream.h>
class  Point{
protected:  int x,y;
public:
   Point(int cx=0,int cy=0){x=cx;y=cy;}
   int getx(){return x;}
};
class  Point3D:public Point{
 int x,y,z;
public:
Point3D(int cx,int cy,int cz){
 x=cx; y = cy; z = cz;
}
int  GetX(){ return x; }
 };
void main(){
  Point3D a(100,150,200);
  int x = a.getx();
  int y = a.GetX();
  cout<<"x="<<x<<" y="<<y<<endl;
 }
```



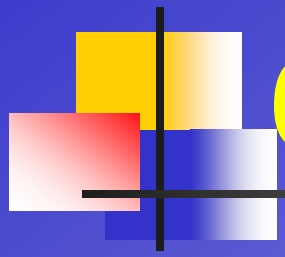
派生类成员对基类私有成员的访问

```
class B;
class A{
    int a, b;
public:
    A(int x) { a=x; };
    friend B;
};
class B: A{    //派生控制缺省为private
    int b;
public:
    B(int x):A(x) { b=x;  A::b=x; };
};
void main(void) { B x(7); }
```



6.4 构造与析构

- 先调用虚基类的构造函数，接着调用基类的构造函数，然后按照数据成员的声明顺序，依次调用数据成员的构造函数或初始化数据成员，最后执行派生类构造函数的函数体。

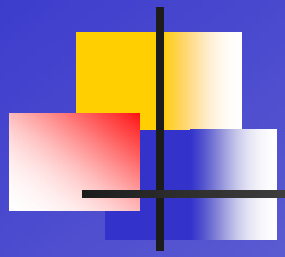


6.4 构造与析构

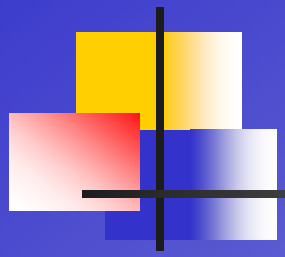
- 如果虚基类和基类已定义了带参数的构造函数，或者派生类已定义了引用成员、只读成员或者需要用带参数的构造函数初始化的对象成员，则派生类必须定义自己的构造函数。派生类只在构造虚基类或基类时调用它们的构造函数，此后，派生类不再调用或访问虚基类或基类的构造函数。

6.4 构造与析构

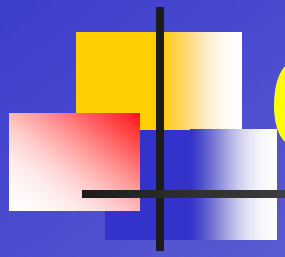
```
#include <iostream.h>
class A{
    int  a;
public:
    A(int  x):a(x)
{ cout<<a; }
    ~A( ) { cout<<a; }
};
class B: A{
    int  b, c;
    const int  d;
    A  x, y;
public:
    B(int v):
b(v),y(b+2),x(b+1),d(b
),A(v) {
        c=v;
        cout<<b<<c<<d;
        cout<<'C';
    }
    ~B( ) { cout<<'D'; }
};
void main(void) { B
z(1); }
```



- 输出结果：
- 1　//A(1)的输出结果
- 2　//A x(b+1)的输出结果
- 3　//A y(b+2)的输出结果
- 111C　//B(v)的输出结果

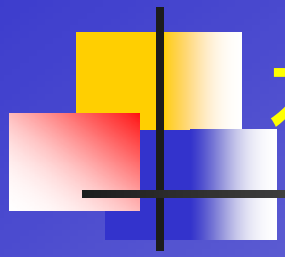


- 输出结果：
- D //~B(v)的输出结果
- 3 // y的析构函数
- 2 // x的析构函数
- 1 // B的基类的构造函数



6.4 构造与析构

- 基类构造函数不带参数的情况：
- 如果基类有缺省构造函数或者有不带参数的构造函数，则基类的构造函数可以不再派生类的构造函数的定义中出现

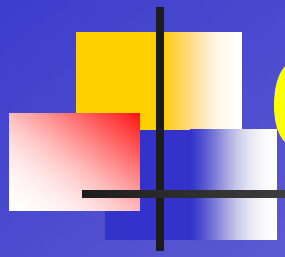


被引用的对象的析构

- 如果引用变量r引用的是一个对象变量v，则对象的构造和析构由对象变量v完成，而不应该由引用变量r完成。如果被引用的对象是用new生成的，则引用变量r必须用delete &r析构对象，否则被引用的对象将因无法完全释放空间而产生内存泄漏。

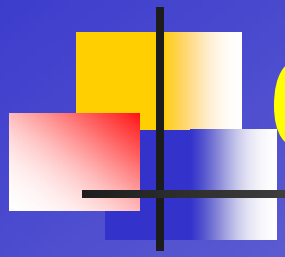
被引用的对象的析构

```
#include <iostream.h>
class A{
    int  i;
    int *s;
public:
    A(int x) { s=new int[i=x]; cout<<"(C): "<<i<<"\n"; }
    ~A( ) {delete s; cout<<"(D): "<<i<<"\n"; }
};
void sub1(void) {  A &p=*new A(1);  }
void sub2(void) {  A *q=new A(2); }
void sub3(void) {  A &p=*new A(3); delete &p; }
void sub4(void) {  A *q=new A(4);  delete q;  }
void main(void)
{
    sub1( );
    sub2( );
    sub3( );
    sub4( );}
```



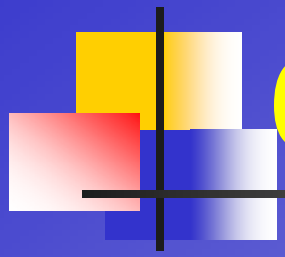
6.5 父类和子类

- 输出：
- (C): 1
- (C): 2
- (C): 3
- (D): 3
- (C): 4
- (D): 4



6.5 父类和子类

- 如果派生类的派生控制为public，则这样的派生类称为基类的子类，而相应的基类则称为派生类的父类。
- C++允许父类指针直接指向子类对象，也允许父类引用直接引用子类对象。
- 父类指针实际指向的对象的类型不同，通过父类指针调用的虚函数的行为就不同，从而产生多态。

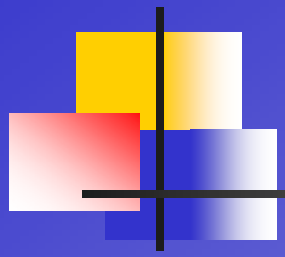


6.5 父类和子类

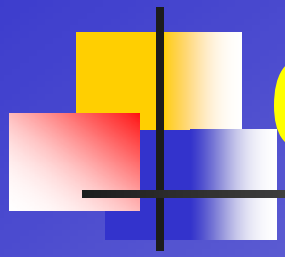
- 编译程序只能根据类型定义静态地检查语义。由于父类指针可以直接指向子类对象，而到底是指向父类对象还是子类对象只能在运行时确定，故编译时只能把父类指针指向的对象都当作父类对象。因此，在访问这些对象的数据成员或函数成员时，不能超越父类对象为相应成员规定的访问权限。

6.5 父类和子类

```
#include <iostream.h>
class POINT{
    int  x, y;
public:
    int getx( ) { return  x; }
    int gety( ) { return  y; }
    void show( )
{ cout<<"Show a point\n"; };
    POINT(int x, int y)
{ POINT::x=x;
POINT::y=y; };
};
class CIRCLE: public POINT{
    int r;
public:
    int getr( ) { return  r; }
    void show( ) { cout<<"Show
a circle\n"; };
    CIRCLE(int x, int y, int
r):POINT(x, y)
{ CIRCLE::r=r; };
};
void main(void)
{
    CIRCLE c(3, 7, 8);
    POINT *p=&c;
    cout<<"The circle with
radius "<<c.getr( );
    cout<<" is at ("<<p-
>getx( )<<", "<<p-
>gety( )<<")\n";
    p->show( );
}
```



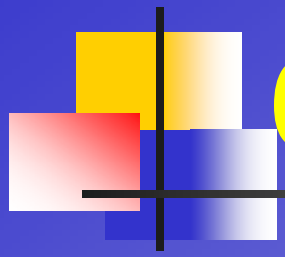
- 输出结果：
  - The circle with radius 8 is at (3, 7)
  - Show a point //p->show()调用的是p的类型定义的对象的函数



6.5 父类和子类

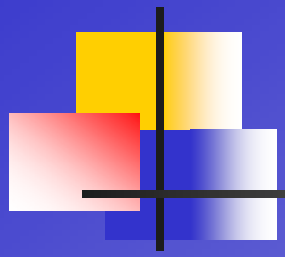
- 如果基类和派生类没有构成父子关系，则普通函数定义的基类指针不能直接指向派生类对象，而必须通过强制类型转换才能指向派生类对象。同理，普通函数定义的基类引用也不能直接引用派生类对象，而必须通过强制类型转换才能引用派生类对象。

```
#include <iostream.h>
class A{
    int a;
public:
    int getv( ) { return a; }
    A( ) { a=0; }
    A(int x) { a=x; }
};
class B: A{
    int b;
public:
    int getv( ) { return b+A::getv( ); }
    B( ) { b=0; }
    B(int x):A(x) { b=x; }
};
class C: public A{
    int c;
public:
    int getv( ) { return c+A::getv( ); }
    C( ) { c=0; }
    C(int x):A(x) { c=x; }
};
void main(void)
{
    A &p=*new C(3);
                    //直接引用C类对象
    A &q=*(A *)new B(5);
                                //强制转换引用B类对象

    cout<<"p.getv( )="<<p.getv( )<<"\n";

    cout<<"q.getv( )="<<q.getv( )<<"\n";
    delete &p;
                //析构C(3)的父类A而非子类C
    delete &q;
                        //析构B(5)的父类A 而非子类B
}
```

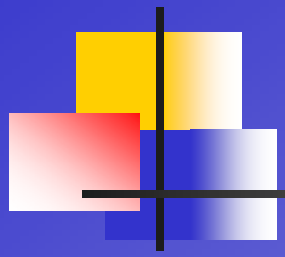


6.5 父类和子类

- 输出：
- p.getv( )=3
- q.getv( )=5



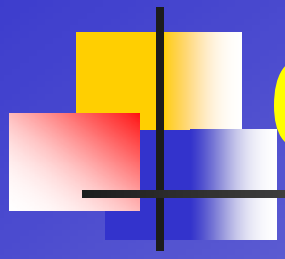
- 上述普通函数是指开工函数、收工函数以及没有定义为父类友元的非成员函数。如果上述普通函数main定义为类A的友元，则main定义的A &q可直接引用类A的派生类类B的对象，即不必通过强制类型转换就允许引用A &q=*new B(5)。



- 对于同类A没有派生关系的类X的函数成员mf来说，如果mf没有定义为类A的成员友元，则mf定义的A类指针不能直接指向类X的对象，mf定义的A类引用也不能直接引用类X的对象。
- 对于从基类A派生的类X的函数成员mf来说，无论类X的派生控制是private、protected还是public，mf定义的A类指针都可以直接指向类X的对象，mf定义的基类引用也可以直接引用类X的对象。也就是说，对于派生类函数成员来说，基类被等同地当作父类。

```
class VEHICLE{
    int speed, weight, wheels;
public:
    VEHICLE(int spd, int wgt, int whl);
};
VEHICLE::VEHICLE(int spd, int wgt, int whl){
    speed=spd;
    weight=wgt;
    wheels=whl;}
class CAR: private VEHICLE{
    int seats;
public:
    VEHICLE *who( );
 CAR(int sd, int wt, int st);
};
```

```
CAR::CAR(int sd, int wt, int st):VEHICLE(sd, wt, 4)
{
    seats=st;
}
VEHICLE *CAR::who( )
{
    VEHICLE *p=this;
                    //基类指针直接指向派生类对象
    VEHICLE &q=*this;
      //基类引用直接引用子类对象
    return  p;
}
void main(void) { }
```

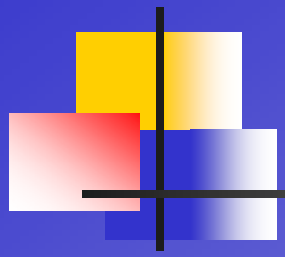


6.6 派生类的存储空间

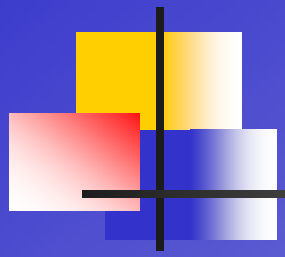
- 基类对象的存储空间是派生类对象存储空间的一部分。因此，在计算派生类对象所占用的存储空间时，除了不考虑基类及派生类的静态数据成员外，必须考虑基类及派生类的所有其他数据成员。

6.6 派生类的存储空间

```
#include <iostream.h>
class A{
    int  h, i, j;
    static int k;
};
class B: A{
    int  m,n,p;
    static int q;
};
int A::k=0;
int B::q=0;
void main(void)
{
    cout<<"Size of int="<<sizeof(int)<< "\n";
    cout<<"Size of A="<<sizeof(A)<<"\n";
    cout<<"Size of B="<<sizeof(B)<<"\n";
}
```



- 输出：
- Size of int=2
- Size of A=6
- Size of B=12



- 此可见，派生类的存储空间等于基类的存储空间与派生类新增成员的存储空间之和，但基类和派生类的所有静态数据成员必须除外。上例的派生类的存储空间如图6.1所示。