



## 第4章 继承

ffh



# 概述

- ◆ **C++** 允许从已存在的类派生新类，所派生的类被称为派生类（**derived class**），又称子类。而已存在的用于派生新类的类被称为基类（**base class**），又称为父类。
  - 通过继承机制，可方便地利用一个已有的类建立新类，重用已有软件中的部分甚至很大的部分。
  - 通过继承，一个新建子类从已有的父类那里获得父类的特性。
  - 派生类继承了基类的所有数据成员和成员函数（不包括基类的构造函数和析构函数），并可以增加自己的新成员，同时也可以调整来自基类的数据成员和成员函数。
  - 基类和派生类是相对而言的。一个基类可以派生出多个派生类，每一个派生类又可以作为基类再派生出新的派生类。一代一代地派生下去，就形成了类层次结构。





# 概述

---

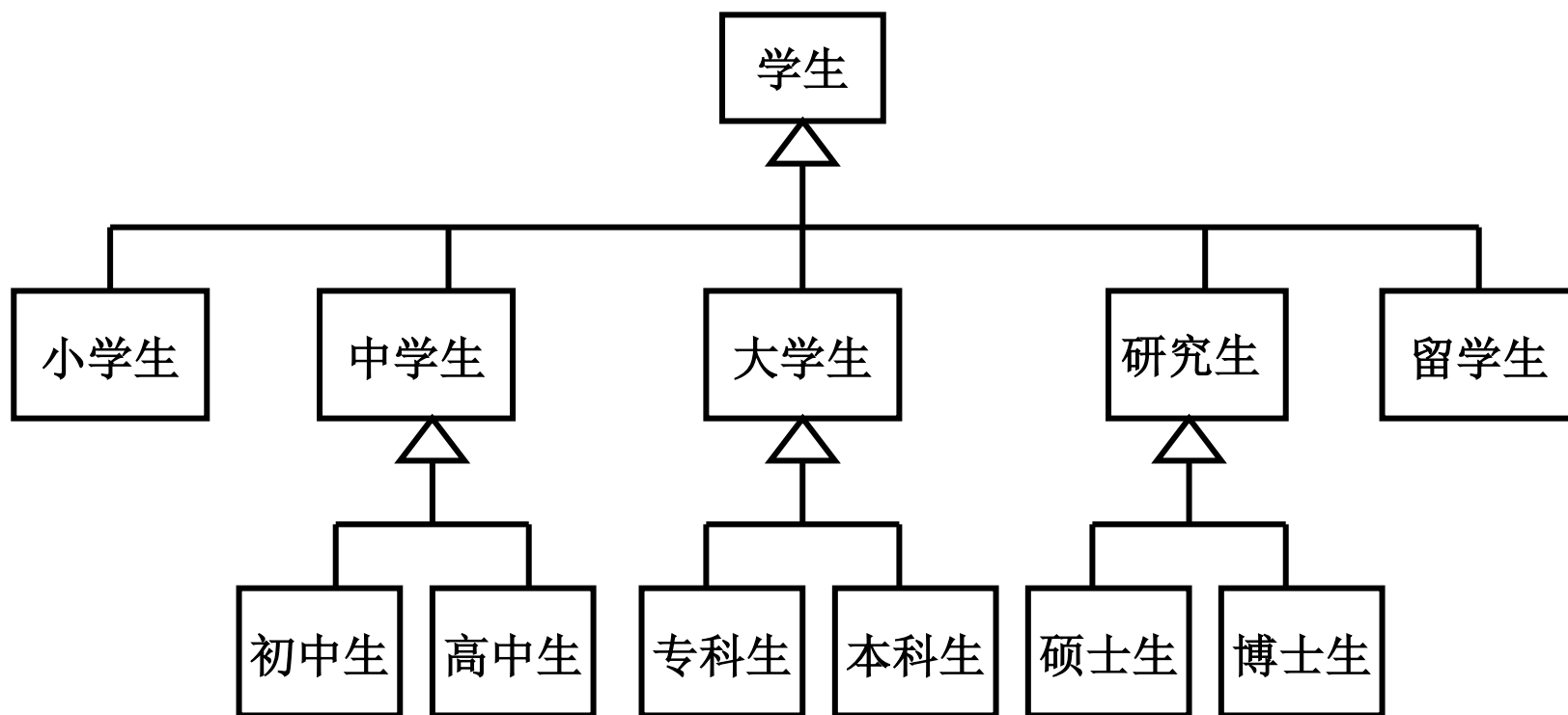
- ◆ 在C++中，一个派生类既可以从一个基类派生，也可以从多个基类派生。
  - 从一个基类派生类的继承被称为单继承。
  - 从多个基类派生类的继承被称为多继承。
  
- ◆ 举例如下：





# 概述

一个派生类只从一个基类派生，这称为单继承(single inheritance)，这种继承关系所形成的**类层次**是一个树形结构，如图。





# 概述

一个派生类不仅可以从一个基类派生，也可以从多个基类派生。  
一个派生类有两个或多个基类的称为多重继承(multiple inheritance)，  
这种继承关系所形成的结构如图所示。

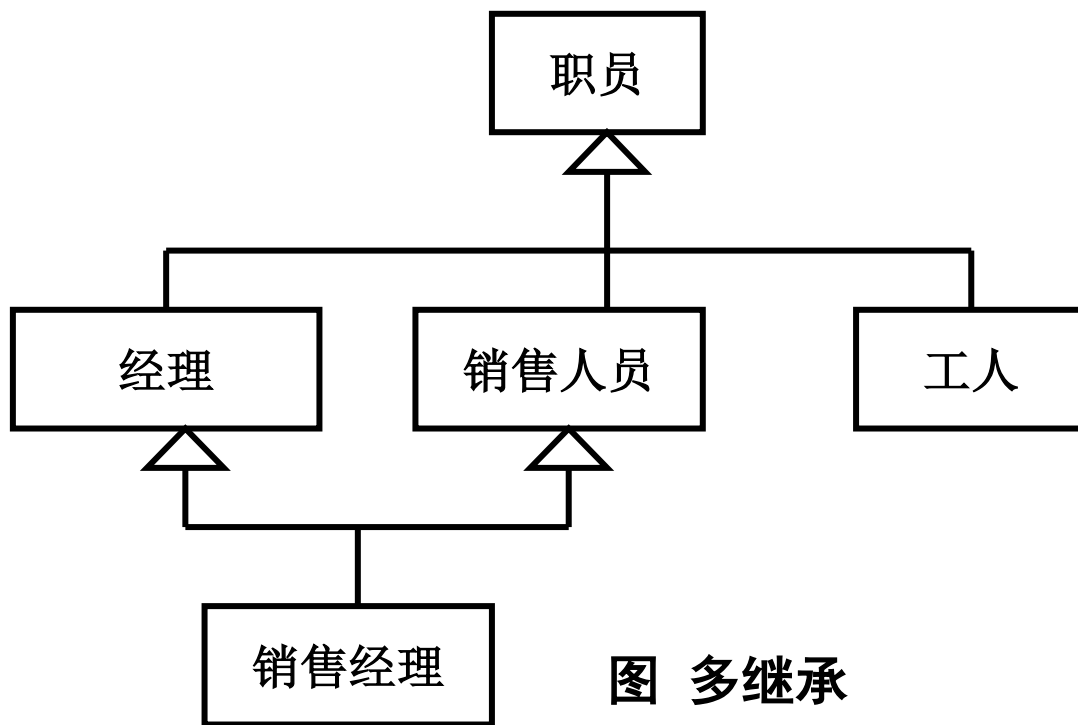


图 多继承





## 4.2 概念和语法

---

◆ 单继承派生类的声明格式如下：

```
class 派生类名: [继承方式] 基类名
{
    派生类新增加的成员
};
```

- 继承方式可是public(公用的)、private(私有的)、protected(受保护的)。此项可选，如不写此项，则默认为private(私有的)。
- 举例：





## 4.2 概念和语法

---

```
#include <iostream>
using namespace std;
class Base    //声明基类
{ public:    //基类公用成员
    void setx(int n) { x=n; }
    int getx() { return x; }
    void showx() { cout<<"Base class: x="<<x<<endl; }
private:    //基类私有成员
    int x;
};
```





## 4.2 概念和语法

```
class Derived: public Base //以public方式声明派生类Derived
{ public:
    void sety(int n){ y=n; }
    int gety(){ return y; }
    void showy()
    { cout<<"Derived class: y="<<y<<endl; }
private:
    int y;
};

void main()
{
    Derived a;
    a.setx(5); a.sety(10);
    a.showx(); a.showy();
}
```







## 4.2 概念和语法

---

### ◆ 派生类的构成

- 派生类中的成员包括从基类继承过来的成员和自己新增加的成员两大部分。
- 从基类继承过来的成员体现了派生类从基类继承而获得的**共性**；
- 而新增加的成员体现了派生类的**个性**，体现了派生类与基类的不同，体现了不同派生类的区别。

下图是上例中基类Base和派生类Derived的成员示意图



## 4.2 概念和语法

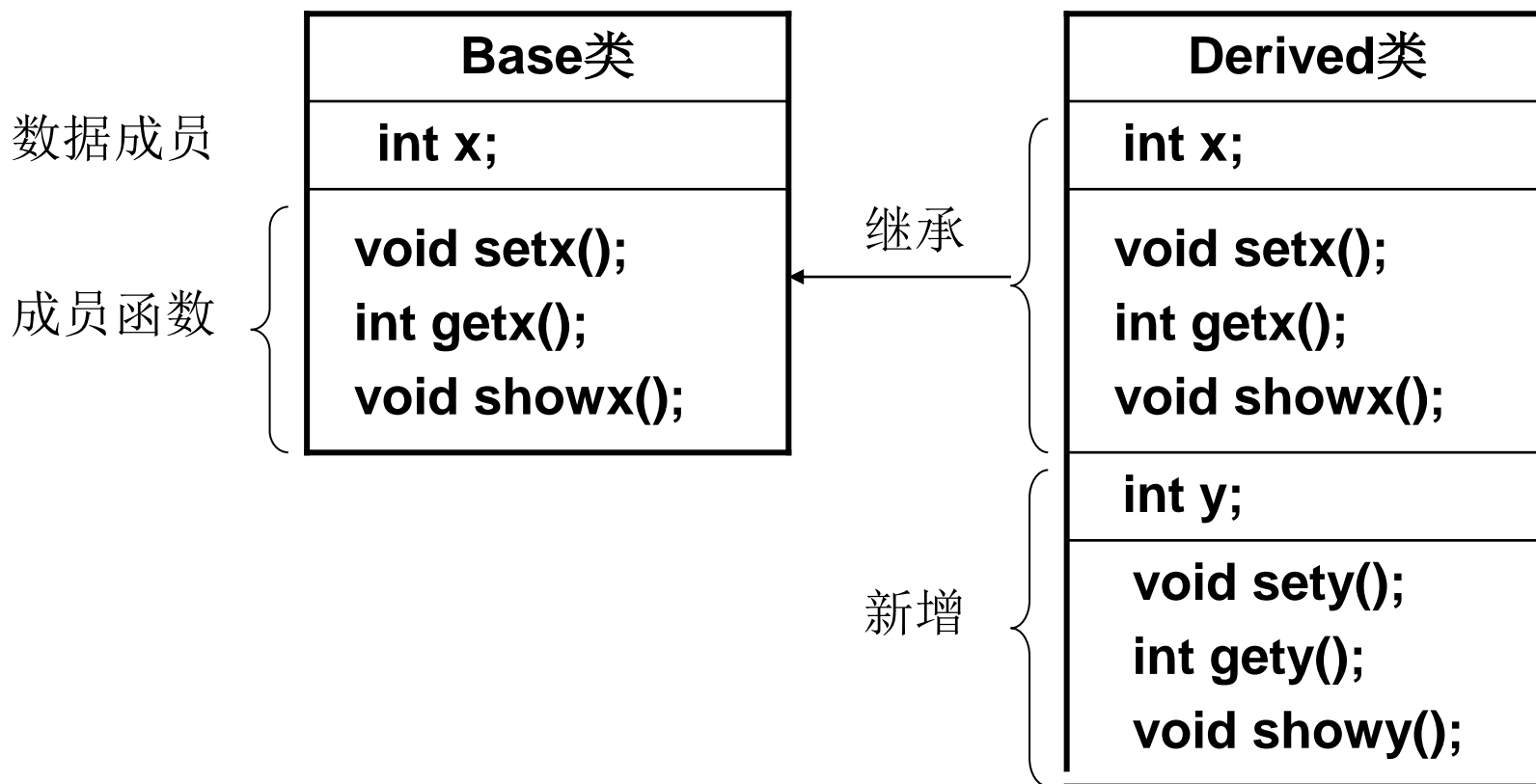


图 基类Base和派生类Derived的成员示意图





## 4.2 概念和语法

### ◆ 其中：

- **public**: 访问控制关键字,指明继承方式是公有继承。
  - 当一个类通过公有继承方式从基类继承时, 基类中的公有成员在派生类中也是公有的。
  - 不指明继承方式关键字**public**时, 编译器会默认继承方式为**private**或**protected**。
- “:” 用于建立基类与派生类的层次结构。
- 基类: **C++**提供的或用户自定义的类。
- 派生类中可以定义数据成员和成员函数, 此外, 还继承基类的所有成员。

### ◆ 再如(书p132-例4-1)





## 4.2 概念和语法

---

```
class Pen{  
public:  
    enum ink{off,on};  
    void set_status(ink);  
    void set_location(int,int);  
private:  
    int x;  
    int y;  
    ink status;  
};
```

```
class Cpen: public Pen{  
public:  
    void set_color(int);  
private:  
    int color;  
};
```





## Pen

- **x,y,status**  
**+void set\_status(ink);**  
**+void set\_location(int,int)**

**CPen**

继承的

**X x,y,status**  
**+void set\_status(ink);**  
**+void set\_location(int,int)**  
**-color**  
**+void set\_color(int)**

CPen p;

p.x=5;

✗

//x是不可访问成员

p.set\_status(Pen::on); ✓

p.color=1;

✗

//color是私有成员

p.set\_color(1);

✓

例 4.1\_133





## 4.2.1 继承机制下的私有成员

- ◆ 基类的所有私有成员仅在基类中可见，而在派生类中是不可见的。
- ◆ 但派生类对象会为基类中的所有私有成员分配空间。

如上例中，**CPen**类从**Pen**类继承了数据成员**x**、**y**和**status**，尽管这些成员在**CPen**类中是不可见的，但无论何时创建**CPen**类的对象时，该对象都将获得相应的存储空间来保存**x**、**y**和**status**等数据成员。
- ◆ 尽管在派生类中不能直接访问基类的私有成员，但可以通过间接的方式（调用从基类继承来的公有成员函数）进行。

如上例中，**x**、**y**和**status**可以通过成员函数**set\_location**和**set\_status**进行访问。

再如:书例4-2



## 继承机制下的私有成员

```
class Point {
public:
    void set_x( int x1 ) { x = x1; }
    void set_y( int y1 ) { y = y1; }
    int get_x() const { return x; }
    int get_y() const { return y; }
private:
    int x;
    int y;
};

class Intense_point : public Point {
public:
    void set_intensity( int i ) { intensity = i; }
    int get_intensity() const { return intensity; }
private:
    int intensity;
};
```





## 继承机制下的私有成员

表4-1 Intense\_point 的成员

成 员	在Intense_point中的状态	来 源
x	不可见	从Point类继承
y	不可见	从Point类继承
set_x	public	从Point类继承
set_y	public	从Point类继承
get_x	public	从Point类继承
get_y	public	从Point类继承
intensity	private	由Intense_point类新增
set_ intensity	public	由Intense_point类新增
get_ intensity	public	由Intense_point类新增







## 4.2.2 改变访问限制

---

- ◆ 使用**using**声明可以改变成员在派生类中的访问限制。例如，基类中的公有成员一般情况下被继承为公有成员，但使用**using**声明可将其改为私有成员（或保护成员）

如：书例4.3



## 改变访问限制

---

```
class BC { // base class  
public:  
    void set_x( float a ) { x = a; }  
private:  
    float x;  
};  
  
class DC : public BC { // derived class  
public:  
    void set_y( float b ) { y = b; }  
private:  
    float y;  
    using BC::set_x;  
};
```





## 改变访问限制

---

这样就无法直接通过**DC**类的任何对象调用**set\_x**:

```
int main() {  
    DC d;  
    d.set_y( 4.31 ); // OK  
    d.set_x( -8.03 ); // ***** ERROR: set_x is private in DC  
    //...  
}
```

思考，能否用：d.BC::set\_x(-8.03); 可以





# 改变访问限制

---

- ◆ 如果基类的某个公有成员函数在继承类中不适合，则可以通过**using**声明将其转变为私有成员函数，从而使它在派生类中隐藏起来

例如，“有序表类”派生自“无序表类”。

基类“无序表类”中的某些成员函数可能并不适合“有序表类”，如某个成员函数“插入”，其在基类中的算法为：在一个无序表中的任意位置插入一个元素。这个算法显然不符合有序表的处理要求。这样的成员函数就应通过**using**声明将其在派生类中隐藏起来。





## 4.2.3 名字隐藏

- ◆ 如果派生类添加了一个数据成员，而该成员与基类中的某个数据成员**同名**，新的数据成员就隐藏了继承来的同名成员
- ◆ 如果派生类添加了与基类的某个成员函数同名的函数，则该函数就隐藏了基类中的同名函数。





## 名字隐藏

### 例4-4

```
class BC { // base class
public:
    void h( float ); // BC::h
};

class DC : public BC { // derived class
public:
    void h( char [ ] ); // ***** DANGER: hides BC::h

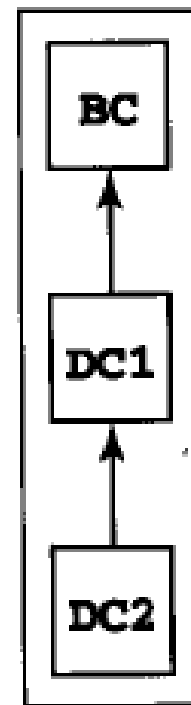
int main() {
    DC d1;
    d1.h( "Boffo!" ); // DC::h, not BC::h
    d1.h( 707.7 ); // ***** ERROR: DC::h hides BC::h
    d1.BC::h( 707.7 ); // OK: invokes BC::h
    //...
}
```



## 4.2.4 间接继承

- ◆ 数据成员和成员函数可以沿着继承链路来继承，继承链路包括了基类及所有派生类。
- ◆ 如右例，DC2 除了继承DC1的数据成员和成员函数，还将经由DC1继承BC的数据成员和成员函数
- ◆ 例：

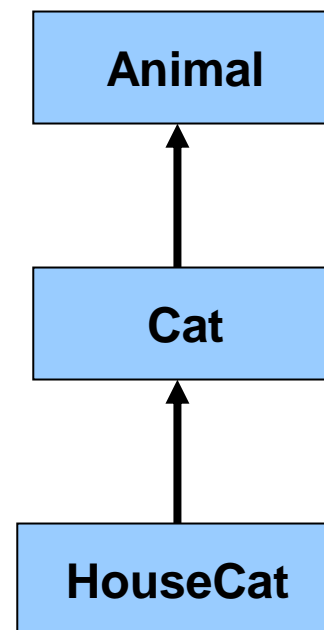
```
class Animal {  
public:  
    string species;  
    float lifeExpectancy;  
    bool  warmBlooded_P;  
};
```





## 间接继承

```
class Cat : public Animal {  
public:  
    string range[ 100 ];  
    float favoritePrey[ 100 ][ 100 ];  
};  
  
class HouseCat : public Cat {  
public:  
    string toys[ 10000 ];  
    string catPsychiatrist;  
    string catDentist;  
    string catDoctor;  
    string apparentOwner;  
};
```







# 课堂练习

- ◆ 下面对派生类的描述中，错误的是（D）。
  - A. 一个派生类可以作为另外一个派生类的基类
  - B. 派生类至少有一个基类
  - C. 派生类的成员除了它自己的成员外，还包含了它的基类的成员
  - D. 派生类中继承的基类成员的访问权限到派生类中保持不变
- ◆ 派生类的对象对它的基类成员中，可以访问的是（A）。
  - A. 公有继承的公有成员。
  - B. 公有继承的私有成员。
  - C. 公有继承的保护成员。
  - D. 私有继承的公有成员。
- ◆ 对基类和派生类的关系描述中，错误的是（B）。
  - A. 派生类是基类的具体化。
  - B. 派生类是基类的子集。
  - C. 派生类是基类定义的延续。
  - D. 派生类是基类的组合。





# 课堂练习

---

- ◆ 若类A与类B的定义如下：

```
class A {  
    int i,j;  
    public: void get(); //..... };  
class B:A {  
    int k;  
    public: void make() { k=i*j; } //.... };
```

则上述定义中，非法的是（ D ）。

A. void get();    B. int k;    C. void make()    D. k=i\*j;





# 课堂练习

---

- ◆ 下列对派生类的描述中，错误的是（D）。
- A、一个派生类可以作为另一个派生类的基类
  - B、派生类至少有一个基类
  - C、派生类的缺省继承方式是private
  - D、派生类只继承了基类的公有成员和保护成员





## 4.3 示例程序：影片跟踪管理

---

- ◆ 构造一个类层次结构，用来对影片的制作进行跟踪管理，这些影片包括一些特殊的影片，如外国影片、导演的影片剪辑。

所有的影片都具有标题、导演、时间和等级（0星到4星）等共同属性；

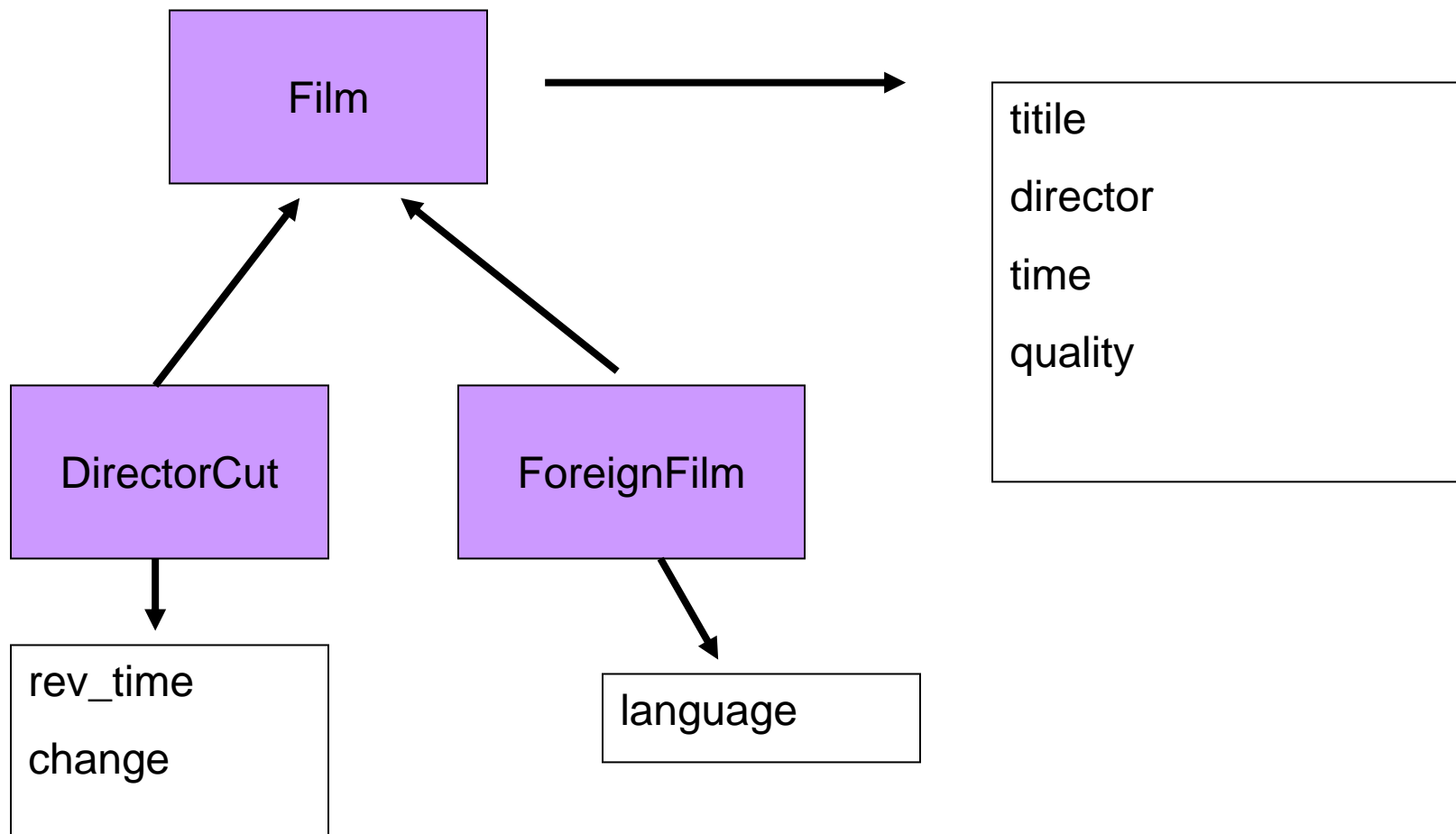
导演的影片剪辑 还有用来存储影片的修订时间、影片变更内容等信息；

外国影片还有存储影片的不同语言版本属性。





## 4.3 示例程序：影片跟踪管理





# 程序设计建议

## ◆ 首先确定类层次，再确定每个类的成员

可以首先设计一个基类**Film**，该类包含所有影片共同的属性及存取这些属性的成员函数，同时还为它设计一个专门用来输出信息的成员函数。

从基类**Film**派生出一个**DirectorCut**类，并为其添加一些数据成员用来存储影片修订时间、影片变更内容等信息，再添加一些成员函数以对这些新增数据成员进行访问，同时为**DirectorCut**类也设计一个用来输出信息的成员函数。

从基类**Film**派生出一个**ForeignFilm**类，并为其添加数据成员来存储影片的不同语言版本，再添加一些成员函数以对这些新增数据成员进行访问，同时为**ForeignFilm**类设计一个用来输出信息的成员函数。

## ◆ 设计代码时，自上而下。先基类，后派生类。





# 解决方案

---

## ◆ Film:

- 四个数据成员
  - title: string, 影片标题
  - Director: string, 导演姓名
  - time: int, 影片播放时间（精确到分钟）
  - quality: int, 影片等级： 0星（最差）到4星（最好）
- 六个数据成员函数(设置上述数据成员，考虑c风格字符串) + output（输出上述数据成员）





```
class Film {  
public:  
    void store_title( const string& t ) { title = t; }  
    void store_title( const char* t ) { title = t; }  
    void store_director( const string& d ) { director = d; }  
    void store_director( const char* d ) { director = d; }  
    void store_time( int t ) { time = t; }  
    void store_quality( int q ) { quality = q; }  
    void output() const;  
private:  
    string title;  
    string director;  
    int time;        // in minutes  
    int quality;     // 0 (bad) to 4 (tops)  
};
```







# 解决方案

---

```
void Film::output() const
{
    cout << "Title: " << title << '\n';
    cout << "Director: " << director << '\n';
    cout << "Time: " << time << " mins" << '\n';
    cout << "Quality: ";
    for ( int i = 0; i < quality; i++ )
        cout << '*';

    cout << '\n';
}
```





# 解决方案

---

- ◆ 类**DirectorCut**: 从类**Film**派生而来, 继承了**Film**的数据成员和成员函数。
  - 添加两个私有数据成员
    - **rev\_time**: 用来表示影片修订时间
    - **Changes**: 影片变更内容
  - 添加一些公有成员函数来存取这两个数据成员
  - 设计自己的**output**函数, 隐藏**Film**中的同名函数。





```
class DirectorCut : public Film {  
public:
```

---

```
    void store_rev_time( int t) { rev_time = t; }  
    void store_changes( const string& s) { changes = s; }  
    void store_changes( const char* s) { changes = s; }  
    void output() const;
```

```
private:
```

```
    int rev_time;  
    string changes;
```

```
};
```

```
void DirectorCut::output() const
```

```
{ Film::output();
```

```
    cout << "Revised time: " << rev_time << " mins\n";
```

```
    cout << "Changes: " << changes << '\n';
```

```
}
```

---





# 解决方案

---

- ◆ **ForeignFilm**类也从类**Film**派生而来，也继承了**Film**的所有数据成员和成员函数。
  - 添加一个私有数据成员用来表示影片语种
  - 同时添加公有成员函数来存取这个数据成员。
  - 定义一个**output**函数以输出信息。





# 解决方案

---

```
class ForeignFilm : public Film
{ public:
    void store_language( const string& l ) { language = l; }
    void store_language( const char* l ) { language = l; }
    void output() const;
private:
    string language;
};

void ForeignFilm::output() const
{ Film::output();
  cout << "Language: " << language << '\n';
}
```

---





## 解决方案

---

◆ 最后，调用的主程序如下：

```
int main()
{   Film f;
    f.store_title("Rear Window"); f.store_director("Alfred Hitchcock");
    f.store_time(112);              f.store_quality(4);
    cout<< "Film--\n";    f.output();    cout<<endl;
    DirectorCut d;
    d.store_title("Jail Bait");      d.store_director("Ed Wood");
    d.store_time(70);    d.store_quality(2); d.store_rev_time(72);
    d.store_changes("Extra footage not in originao included");
    cout<<"DirectorCut--\n";  d.output();  cout<<endl;
    .....
    return 0 ;
}
```

---





## 4.4 保护成员

---

- ◆ 除了私有和公有成员， **C++**还提供了保护成员。
- ◆ 在没有继承的情况下，保护成员和私有成员类似，只在该类中可见。
- ◆ 在公有继承方式下，保护成员和私有成员具有不同性质：
  - 基类的保护成员在派生类中是可见的。
  - 而基类的私有成员在派生类中是不可见的。





# 保护成员

## 例4-6

```
class BC { // base class
public:
    void set_x( int a ) { x = a; }
protected:
    int get_x() const { return x; }
private:
    int x;
```

```
class DC : public BC {
public:
    void add2() { int c = get_x(); set_x( c + 2 ); }
};
```

下列错误:

```
class DC : public BC {
    //...
    // ***** ERROR: x is accessible only within BC
    void add3() { x += 3; }
    //...
};
```

表4-3 DC类的成员

成 员	在DC类中的状态	来 源
set_x	public	从类BC继承
get_x	protected	从类BC继承
x	不能访问	从类BC继承
add2	public	由类DC新增





## 保护成员

---

```
int main() {  
    DC d;  
    d.set_x( 3 ); // OK -- set_x is public in DC  
  
    // ***** ERROR: get_x is protected in DC  
    cout << d.get_x() << '\n';  
  
    d.x = 77; // ***** ERROR: x is private in BC  
    d.add2(); // OK -- add2 is public in DC  
    //...  
}
```





# 保护成员

---

- ◆ 派生类可对从基类继承来的保护成员进行访问，也就是说保护成员在派生类中是可见的
- ◆ 但派生类不能访问一个基类对象的保护成员，因为基类对象属于基类，不属于派生类。

如下例





## 保护成员

### 例4-7

```
class BC { // base class
protected:
    int get_w() const;
    //...
};
class DC : public BC { // derived class
public:
    // get_w belongs to DC because it is inherited from BC
    void get_val() const { return get_w(); }

    // ***** ERROR: b.get_w not visible in DC since b.get_w is
    // a member of BC, not DC
    void base_w( const BC& b ) const
        { cout << b.get_w() << '\n'; }
};
```





## 采用public派生

---

- ◆ 基类的私有成员，在派生类中不可见，基类的私有成员只能被基类的其他成员函数访问(除**friend**函数)。
- ◆ 基类的保护成员，在派生类可见，基类的保护成员除了能被基类的其他成员函数访问外，还能被类层次结构中的所有成员函数访问。
- ◆ 基类的公有成员，在派生类可见，可被任何函数访问。
- ◆ 例4-8





## 采用public派生

---

```
class BC {  
public:  
    void init_x() { x = 0; }  
protected:  
    int get_x() const { return x; }  
private:  
    int x;  
};  
  
class DC : public BC {  
public:  
    void g() { init_x(); cout << get_x() << '\n'; }  
};
```



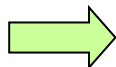


## 保护成员

- ◆ 一般来说，应避免将数据成员设计为保护类型
- ◆ 而是采用私有数据成员与相应保护型访问函数结合的模式，便于实现数据隐藏（复杂数据成员例外）。

例4-9

```
class BC{
protected:
    int y;
};
class DC:public BC{
public:
    void g(int a )
    { y=a;  cout<< y ;}
};
```



```
class BC{
protected:
    int get_y( ) const { return y;}
    void set_y( int a) { y=a;}
private: int y;
};
class DC:public BC{
public:
    void g(int a){ set_y(a); cout<<get_y( );}
```





# 课堂练习

- ◆ 关于公有继承，下列说法错误的是（c）。
  - A. 基类的公有成员和保护成员被继承后作为派生类的公有成员和保护成员。
  - B. 派生类的其他成员可以直接访问基类的公有成员和保护成员。
  - C. 派生类的对象可以访问基类的私有成员。
  - D. 派生类成员和对象都无法访问基类的私有成员。
- ◆ 下列说法中错误的是（B）。
  - A. 公有继承时基类中的public成员在派生类中仍是public的；
  - B. 公有继承时基类中的private成员在派生类中仍是<sup>不可访问</sup>private的；
  - C. 私有继承时基类中的public成员在派生类中是private的；
  - D. 保护继承时基类中的public成员在派生类中是protected的；



## 课堂练习

---

- ◆ 在公有继承中，基类中的公有成员作为派生类的（A）。
  - A. 公有成员    B. 私有成员
  - C. 保护成员    D. 私有成员函数
- ◆ 基类中的（C）不允许外界访问，但允许派生类的成员访问，这样既有一定的隐藏能力，又提供了开放的接口。
  - A. 公有成员    B. 私有成员    C. 保护成员    D. 私有成员函数





## 4.5 继承机制下的构造函数与析构函数

### ◆ 4.5.1 继承机制下的构造函数

当创建一个派生类对象时，基类的缺省构造函数被自动调用，用来对派生类对象中的基类部分进行初始化，并完成其他一些相关事务。

例4-10:

```
class BC{ //Base class
    public:
        BC() { x=y= -1; }
    protected:
        int get_x() const { return x; }
        int get_y() const { return y; }
    private:
        int x; int y; };
```





## 4.5.1 继承机制下的构造函数

---

```
class DC: public BC{
    public:
        void write() const {
            cout << get_x() * get_y() << endl;
        }
};

int main()
{
    DC d1; //d1.BC invoked
    d1.write(); // 1
}
```





# 继承机制下的构造函数

- ◆ 派生类是不能继承基类的构造函数和析构函数的。
- ◆ 当基类构造函数的功能对派生类而言不够用的时候，派生类必须定义自己的构造函数。
- ◆ 如果派生类定义了自己的构造函数，则由该构造函数负责对象中“派生类添加部分”的初始化工作。
- ◆ 在设计派生类的构造函数时，不仅要考虑派生类新增数据成员的初始化，还应当考虑对其从基类继承过来的数据成员的初始化。采取的方法是在执行派生类的构造函数时，调用基类的构造函数（前提是基类拥有构造函数）。





# 继承机制下的构造函数

---

```
class Base                                //声明基类Base
{ public:
    Base(int m,int n ) { x=m; y=n; } //基类构造函数
protected:                             //保护部分
    int x;  int y;  };
class Derived: public Base //声明公有派生类Derived
{ public:                             //派生类的共用部分
    Derived (int m,int n,int k ):Base(m,n) { z=k; }
    void show()
    { cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl; }
private:                               //派生类的私有部分
    int z;
};
```

---





# 继承机制下的构造函数

---

```
int main()
{
    Derived obj (12,34,56);
    obj.show(); //输出结果
    return 0;
}
```

## ◆ 输出结果

x=12

y=34

z=56

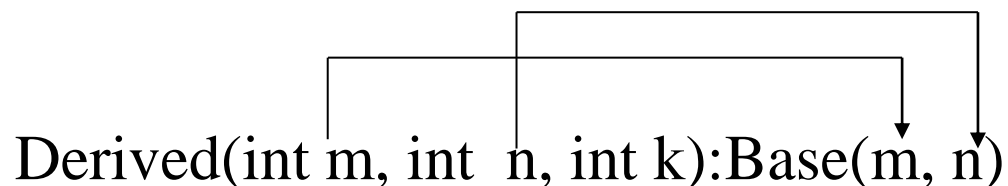




# 继承机制下的构造函数

## ◆ 派生类的构造函数

上例中：派生类**Derived**的构造函数有3个形参，前2个作为调用基类构造函数的实参，第3个为对派生类新增数据成员z初始化所需要的参数。（次序无所谓）



## ◆ 定义简单派生类构造函数的一般形式为：

<派生类构造函数名>(<总参数列表>): <基类构造函数名>(<参数表>)  
{ <派生类新增数据成员初始化> };

## ◆ 再如：（书上例4-11）



```
class Animal {
public:
    Animal() { species = "Animal"; }
    Animal( const char* s ) { species = s; }
private:
    string species;
};

class Primate: public Animal {
public:
    Primate() : Animal( "Primate" ) { }
    Primate( int n ) : Animal( "Primate" )
        { heart_cham = n; }
private:
    int heart_cham;
};

Animal slug;                // Animal()
Animal tweety( "canary" );  // Animal( const char* )
Primate godzilla;           // Primate()
Primate human( 4 );         // Primate( int )
```



# 继承机制下的构造函数

- ◆ 在建立一个对象时，执行构造函数的顺序是：
  - 1) 最先调用基类的构造函数，对基类数据成员初始化；  
对基类的构造函数的调用顺序取决于这些基类在被继承时的说明顺序，与它们在初始化列表中给出的顺序无关。
  - 2) 再调用数据成员是类对象的构造函数，其调用次序按在类中定义的先后次序；
  - 3) 最后执行派生类构造函数的函数体，对派生类新增数据成员初始化。

先父母，后客人，最后自己







# 继承机制下的构造函数

- ◆ 多米诺骨牌效应：在一个层次很深的类层次结构中，创建一个派生类对象将导致派生链中的所有类的构造函数被逐一调用。

即在派生的层次结构中，每一层派生类的构造函数只负责调用其上一层（即它的直接基类）的构造函数。

如上例（见下页）



# 继承机制下的构造函数

```
class Base                                //声明基类Base
{ public:    Base(int m,int n ) { x=m; y=n; } //基类构造函数
  protected: int x;  int y;          //保护部分};
class Derived: public Base //声明公有派生类Derived
{ public:                          //派生类的共用部分
    Derived (int m,int n,int k ):Base(m,n) { z=k; }
    void show()
    { cout<<"x="<<x<<endl<<"y="<<y<<endl<<"z="<<z<<endl; }
  private: int z;                  //派生类的私有部分
};
```

由派生类Derived再派生出派生类IndirectDerived，则派生类IndirectDerived的定义如下：






# 继承机制下的构造函数

---

```
class IndirectDerived: public Derived
{public:
    IndirectDerived (int m,int n,int k,int t):Derived(m,n,k)
    { w=t; }
    void show()
    {   Derived::show();
        cout<<"w="<<w<<endl;
    }
private:
    int w;
};
```

◆ 再如书上例4-12





```
class Animal {
public:
    Animal() { species = "Animal"; }
    Animal( const char* s ) { species = s; }
private:
    string species;
};

class Primate: public Animal {
public:
    Primate() : Animal( "Primate" ) { }
    Primate( int n ) : Animal( "Primate" )
        { heart_cham = n; }
private:
    int heart_cham;
};

class Human : public Primate {
public:
    Human() : Primate() { }
    Human( int c ) : Primate( c ) { }
};
```

```
Human jill();    // Human()
Human fred( 4 ); // Human( int )
```



# 继承机制下的构造函数

---

**Animal::Animal( ... )** body executes first



**Primate::Primate( ... )** body executes second



**Human::Human( ... )** body executes third



## 4.5.2 派生类构造函数的规则

- ◆ 如果基类拥有构造函数但没有默认构造函数，那么派生类的构造函数必须显式地调用基类的某个构造函数。

例4-13

```
class BC{
    public:    BC(int a) {cout<<"BC::(int) executes...\n"; x=a; }
    private: int x;  };

class DC:public BC{
    public:
        DC() {cout<<"DC::() executes...\n";} //Error !
    private:
        int y;  };
```





## 4.5.2 派生类构造函数的规则

---

- ◆ 一般来说，最好为基类提供一个默认构造函数，不但可以避免出现上述问题，而且并不妨碍派生类构造函数去调用基类的非默认构造函数。
- ◆ 假设基类拥有默认构造函数，而其派生类也定义了一些构造函数，不过派生类的任何构造函数都没有显式地调用基类的某个构造函数。在这种情况下，当创建一个派生类对象时，基类的默认构造函数将被自动地调用。





## 4.5.2 派生类构造函数的规则

---

例4-14

```
class BC{
    public:
        BC(){cout<<"BC::( ) executes...\n";}
        BC(int a){cout<<"BC::(int) executes...\n";    x=a; }
    private: int x; };
class DC:public BC{
    public:
        DC() {cout<<"DC::() executes...\n";}
    private: int y; };
int main()
{ DC d; }
```

Output:  
BC::() executes  
DC::() executes







## 4.5.2 派生类构造函数的规则

- ◆ 以“DC类从BC类派生”为例，总结如下： (例)
  - 若DC有构造函数而BC没有，当创建DC类的对象时，DC的相应构造函数被自动调用。（系统自动给BC创建默认构造函数）

```
class BC1 //系统自动给BC 创建默认构造函数
{ //... };
class DC1:public BC1
{ public: DC1() { cout<<"DC1:DC1() executes...\n"; }
};
```

-----

```
DC1 d1;
```

(continue)





## 4.5.2 派生类构造函数的规则

- ◆ 以“DC类从BC类派生”为例，总结如下：
  - 若DC没构造函数而BC有，则BC必须拥有默认构造函数。只有这样，当创建DC类的对象时，才能自动执行BC的默认构造函数。

```
class BC2          // BC必须拥有默认构造函数
{ public: BC2() {cout<<"BC2:BC2() executes...\n"; }
};
class DC2:public BC2
{ //...  };
```

-----  
DC2 d2;

(continue)





## 4.5.2 派生类构造函数的规则

- 若DC有构造函数，且BC有默认构造函数，则创建DC类的对象时，BC的默认构造函数会自动执行，除非当前被调用的派生类构造函数在其初始化段中显式地调用了BC的非默认构造函数。

```
class BC3
{ public: BC3() {cout<<"BC3:BC3() executes...\n"; }
      BC3(int pintT) {cout<<"BC3:BC3(int) executes...\n"; } };
class DC3:public BC3
{ public: DC3() {cout<<"DC3:DC3() executes...\n"; } };
class DC31:public BC3 //显式地调用了BC的非默认构造函数
{ public: DC31(int pintT):BC3(pintT) {cout<<"DC3:DC3(int)
      executes...\n"; }    };
```

-----

```
DC3    d3; // BC的默认构造函数会自动执行
```

```
DC31   d31(5);
```





## 4.5.2 派生类构造函数的规则

- ◆ 以“DC类从BC类派生”为例，总结如下：
  - 若DC和BC都有构造函数，但BC没有默认构造函数，则DC的每个构造函数必须在其初始化段中显式地调用BC的某构造函数。只有这样，当创建DC的对象时，BC的构造函数才能获得执行机会。

```
class BC4
{ public: BC4(int pintT) {cout<<"BC4:BC4(int) executes...\n"; }
};

class DC4:public BC4
{ public: DC4(int pintT):BC4(pintT) {cout<<"DC4:DC4(int)
    executes...\n"; }
};

-----
DC4  d4(5);
```

(continue)





## 4.5.2 派生类构造函数的规则

---

- ◆ 以“DC类从BC类派生”为例，总结如下：
  - 若DC和BC都没有构造函数，系统自动给DC、BC创建默认构造函数。

```
class BC5
{ //...
};
class DC5:public BC5
{ //...
};
```

-----  
DC5 d5;





## 4.5.2 派生类构造函数的规则

---

### ◆ 再次强调:

在创建派生类对象时，必须显式地或隐式地执行其基类的某个构造函数，因为有时候，派生类的构造函数可能会依赖基类的构造函数来完成一些必要的操作。例如，依赖基类构造函数来完成部分数据成员的初始化。

### 例4-15

```
class Team {  
public:  
    Team(int len=100) { names=new string[maxno=len]; }  
protected:  
    string *names;  
    int maxno;  
};
```





## 4.5.2 派生类构造函数的规则

---

```
class BaseballTeam:public Team {  
    public:  
        BaseballTeam(const string s[],int si):Team(si)  
    {  
        for(int i=0;i<si;i++)  
            names[i]=s[i];  
    }  
    //...  
};
```





## 4.5.3 继承机制下的析构函数

- ◆ 和构造函数一样，基类的析构函数派生类也不能继承。
- ◆ 在声明派生类时，可以根据需要定义自己的析构函数，用来对派生类中新增加的成员进行清理工作。
- ◆ 在执行派生类的析构函数时，系统会自动调用基类的析构函数，对基类进行清理。
- ◆ 派生类析构函数的执行顺序与构造函数正好相反。

这是因为析构函数通常用来释放由构造函数分配的内存资源。这种次序，可以确保最近分配的内存资源最先被释放。







## 4.5.3 继承机制下的析构函数

```
class Base
{public:
    Base(){cout<<"Base Constructor"<<endl;}
    ~Base(){cout<<"Base Destructor"<<endl;}
};

class Derived:public Base
{public:
    Derived () {cout<<"Derived Constructor"<<endl;}
    ~Derived () {cout<<"Derived Destructor"<<endl;}
};

int main()
{   Derived obj;
    return 0; }
```

程序运行结果如下：  
Base Constructor  
Derived Constructor  
Derived Destructor  
Base Destructor



## 4.5.3 继承机制下的析构函数

---

- ◆ 由于每个类至多只有一个析构函数，因此对析构函数的调用不会产生二义性，这样在析构函数中不必显式地调用其他析构函数，这一点和构造函数的调用规则是不同的。





## 练习

- ◆ 派生类的构造函数的成员初始化列表中，不能包含（**C**）。  
A、基类的构造函数      B、派生类中子对象的初始化  
C、基类中子对象的初始化      D、派生类中一般数据成员的初始化
- ◆ 程序片段如下：

```
class A { public: A( ) {cout<<"A";} };  
class B { public:B( ) {cout<<"B";} };  
class C: public A {  
    B b;  
    public: C( ) {cout<<"C";} };  
int main( ) { C obj; return 0;}
```

执行后的输出结果是（**D**）。

- A. CBA      B. BAC      C. AC      D. ABC





# 练习

◆ 有如下类定义：

```
class XA{      int x;  
    public:  XA(int n) {x=n;} };  
class XB: public XA{ int y;  
    public XB(int a,int b); };
```

在构造函数XB的下列定义中，正确的是（B）。

- A. XB::XB (int a, int b) : x(a), y(b){ }
- B. XB::XB (int a, int b) : XA(a), y(b) { }
- C. XB::XB (int a, int b) : x(a), XB(b){ }
- D. XB::XB (int a, int b) : XA(a), XB(b){ }





# 练习

分析以下程序的执行结果

```
class base
{
    public: base() {cout<< "cb"<<endl;}
           ~base() {cout<< "db"<<endl; } };

class subs:public base
{
    public: subs() {cout<< "cs"<<endl;}
           ~subs() {cout<< "ds"<<endl;} };

void main()
{
    subs s;
}
```

cb  
cs  
ds  
db



## 练习

class base //分析以下程序的执行结果:

```
{    int n;
    public: base(int a) { cout<<"cb"<<endl; n=a;
                        cout<<"n="<<n<<endl; }
    ~base(){cout<<"db"<<endl;}
};
class subs:public base
{
    base bobj;
    int m;
    public: subs(int a,int b,int c):base(a),bobj(c)
    {    cout<<"cs"<<endl; m=b; cout<<"m="<<m<<endl;    }
    ~subs(){cout<<"ds"<<endl;}    };
void main()
{    subs s(1,2,3);    }
```

cb  
n=1  
cb  
n=3  
cs  
m=2  
ds  
db  
db





## 4.6 示例程序：设计序列的类层次结构 (例 sequence)

- ◆ 设计一继承层次结构，用来处理(建立、插入、删除)文件中的字符串序列 (**sequence**) 和有序字符串序列 (**Sorted sequence**)。

说明：

- 所谓**sequence** (序列) 是一个列表，有着第一个、第二个元素等，如序列： Abby George Ben

其中Abby是第一个元素，George是第二个元素，Ben 是第三个元素。

右是另一个序列： George Ben Abby

- **Sorted sequence** (有序序列) 是序列的一种，其中的元素是按照某种顺序排列的。如序列： Abby Ben George 是一个有序序列，因为其中的元素是以首字母为序来排列的。

Abby George Ben 不是一个有序序列



## 4.6 示例程序：设计序列的类层次结构

- ◆ 为实现 **Sequence** 类的建表（数据源自文件）、插入、删除功能：
  - **Sequence** 类应拥有如下数据成员：
    - 用来存储多个字符串的成员。 **s[ ]**
    - 用来存储文件名的成员。 **filename** (字符串存储的文件)
    - 用来存储最后一个字符串的索引值的成员。 **last**
    - 用来处理输入输出文件的成员。 **Ifstream/ofstream in/out**
  - **Sequence**类的公有成员函数可完成如下操作：
    - 添加字符串到指定的位置。 **adds(int, string&)**
    - 删除指定位置的字符串。 **del(int)**
    - 输出序列。 **output()**







## 4.6 示例程序：设计序列的类层次结构

### ◆ Sequence类的默认构造函数完成如下操作：

文字串列表s的初始化：设定最后一个字符串的索引（last）值为-1，表明序列中没有任何字符串。

### ◆ 只有惟一参数（**const char\***类型 代表文件名）的构造函数完成如下操作：（从文件读入文字串到s中）

- 设定最后一个字符串的索引值为-1，表明序列中没任何字符串。
- 将传递进来的代表文件名的字符串拷贝到相应的数据成员（**filename**）中。
- 尝试打开该文件用于输入，如果文件不能打开，构造函数直接返回。
- 从文件中读入序列，直到文件结尾或是内存不够。
- 关闭文件。





## 4.6 示例程序：设计序列的类层次结构

---

- ◆ 析构函数完成如下操作：（写字符串到文件）
  - 如果文件名是一个null字符串，则返回。
  - 打开该文件用于输出。
  - 将序列写入到文件中。
  - 关闭文件。





## 4.6 设计序列的类层次结构

### ◆ 基类Sequence

- 数据成员
  - 最后一个字符串的位置索引 `int last`
  - 字符串数组 `string s[MaxStr]`
  - 文件名 `string filename`
  - 输入/输出文件 `in/out`
- 函数成员
  - 默认无参构造函数 `Sequence()`
  - 有参构造函数 `Sequence(const char*)`
  - 添加 `addS(int,string)`
  - 删除 `del(int )`
  - 输出 `output()`
  - 析构函数 `~Sequence()`



class Sequence {

public:

Sequence() : last( -1 ) { }

---

Sequence( const char\* );

bool addS( int, const string& );

bool del( int );

void output() const;

~Sequence();

protected:

enum { MaxStr = 50 };

string s[ MaxStr ];

int last;

private:

string filename;

ifstream in;

ofstream out;

};





```
Sequence::Sequence( const char* fname ) {
```

```
    last = -1;
```

---

```
    filename = fname;
```

```
    in.open( fname );    if ( !in )    return;
```

```
    while ( last < MaxStr - 1 && getline( in, s[ last + 1 ] ))    last++;
```

```
    in.close();
```

```
}
```

```
Sequence::~~Sequence() {
```

```
    if ( filename == "" )    return;
```


```
    out.open( filename.c_str() );
```

```
    for ( int i = 0; i <= last; i++ )    out << s[ i ] << '\n';
```

```
    out.close();
```

```
}
```





```
bool Sequence::addS( int pos, const string& entry ) {  
    if ( last == MaxStr - 1 || pos < 0 || pos > last + 1 ) return false;  
    for ( int i = last; i >= pos; i-- )    s[ i + 1 ] = s[ i ];  
    s[ pos ] = entry;    last++;  
    return true;  
}
```

---

```
bool Sequence::del( int pos ) {  
    if ( pos < 0 || pos > last )    return false;  
    for ( int i = pos; i < last; i++ )    s[ i ] = s[ i + 1 ];  
    last--;  
    return true;  
}
```

```
void Sequence::output() const {  
    for ( int i = 0; i <= last; i++ ) cout << i << " " << s[ i ] << '\n';  
}
```

---



## 主程序（Sequence类 测试程序）

```
int main()
{
    string inbuff,iwhere; int wh;
    Sequence items("test.dat");
    while(true)
    {
        cout<<"\nSequence output:\n"; items.output();
        cout<<endl<<"1--add"<<endl;
        cout<<"2--del"<<endl;
        cout<<"3--quie"<<endl;
        getline(cin,inbuff);
        if(inbuff=="1")
        {
            cout<<"item to add:"; getline(cin,inbuff);
            cout<<"add where?";getline(cin,iwhere);
            wh=atoi(iwhere.c_str());
```





## 主程序（Sequence类 测试程序）

---

```
        if( items.addS(wh,inbuff))
            cout<<"item added"<<endl;
        else  cout<<"item not added"<<endl; }
else if(inbuff=="2")
{   cout<<endl<<"where to delete:";
    getline(cin,iwhere);
    wh=atoi(iwhere.c_str());
    if(items.del(wh))
        cout<<"item deleted"<<endl;
    else  cout<<"item not deleted\n"; }
else if(inbuff=="3") break; }
return 0;
}
```

---








## 4.6 示例程序：设计序列的类层次结构

### ◆ 派生类SortedSeq

SortedSeq可从Sequence类派生出：

- SortedSeq类无需添加新的数据成员，所有数据成员均由Sequence类继承而来。
- SortedSeq类需定义一个用于插入元素的成员函数，该函数只有一个参数：将要被插入的元素。至于元素插入的位置，不需要作为该函数的参数，因为SortedSeq中的元素是有序的，插入位置可通过计算获得。
- 新添加函数成员
  - 默认无参构造函数SortedSeq()
  - 有参构造函数SortedSeq(const char\*)
  - 添加addSS(const string&)
  - 排序sort()



```
class SortedSeq : public Sequence {
private:
    using Sequence::addS;
protected:
    void sort();
public:
    SortedSeq() { }
    SortedSeq(const char* fname) : Sequence( fname ) { sort(); }
    bool addSS( const string& );
};

bool SortedSeq::addSS( const string& entry ) {
    int i;
    for ( i = 0; i <= last; i++ ) //设为升序
        if ( entry <= s[ i ] )      break;
    return addS( i, entry );
}
```

---



```
void SortedSeq::sort() {  
    string temp;  
    int i, j;  
    for ( i = 0; i <= last - 1; i++ ) {  
        temp = s[ i + 1 ];  
        for ( j = i; j >= 0; j-- )  
            if ( temp < s[ j ] ) s[ j + 1 ] = s[ j ]; else break;  
        s[ j + 1 ] = temp;  
    }  
}
```





## 4.6 示例程序：设计序列的类层次结构

- ◆ **SortedSeq**类的**sort**函数用来对序列进行排序，由于该函数仅为内部使用，因此设计为保护类型。
  - ◆ **SortedSeq**类的默认构造函数调用**Sequence**类的默认构造函数。只有惟一参数的构造函数调用**Sequence**类的只有惟一参数的构造函数，并对输入的序列进行排序。
  - ◆ **SortedSeq**类没有析构函数，它继承**Sequence**类的析构函数。
- ◆ 主程序如下：





## 主程序（ SortedSeq类 测试程序）

```
int main()
{
    string inbuff,iwhere; int wh;
    SortedSeq items2("test.dat");
    while(true)
    {
        cout<<"\nSequence output:\n"; items2.output();
        cout<<endl<<"1--add"<<endl;
        cout<<"2--del"<<endl;
        cout<<"3--quie"<<endl;
        getline(cin,inbuff);
        if(inbuff=="1")
        {
            cout<<"item to add:"; getline(cin,inbuff);
```





## 主程序（SortedSeq类测试程序）

---

```
        if(items2.addSS(inbuff))
            cout<<"item added"<<endl;
        else
            cout<<"item not added"<<endl; }
else if(inbuff=="2")
{
    cout<<endl<<"where to delete:";
    getline(cin,iwhere);
    wh=atoi(iwhere.c_str());
    if(items2.del(wh))
        cout<<"item deleted"<<endl;
    else cout<<"item not deleted\n"; }
else if(inbuff=="3")    break; }

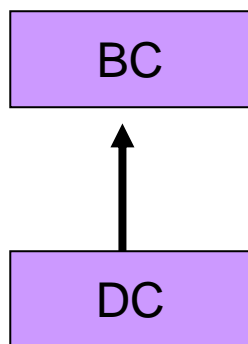
return 0;
}
```

---

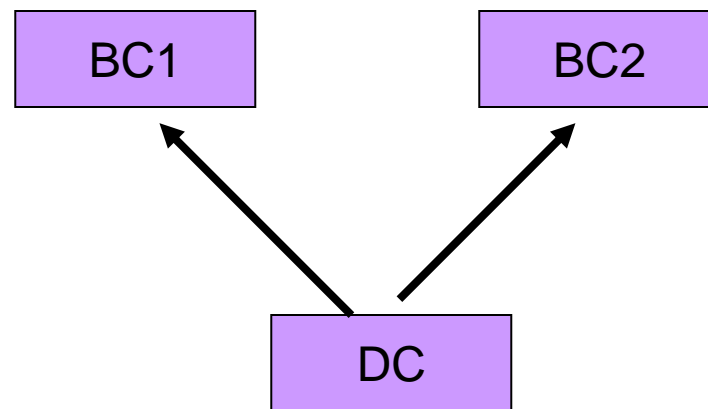


## 4.7 多继承

- ◆ 单继承基类和派生类组成树结构，而多继承基类和派生类组成有向图结构。
- ◆ 多继承的派生类可以同时具有多个基类，它同时继承了这些基类的所有成员。



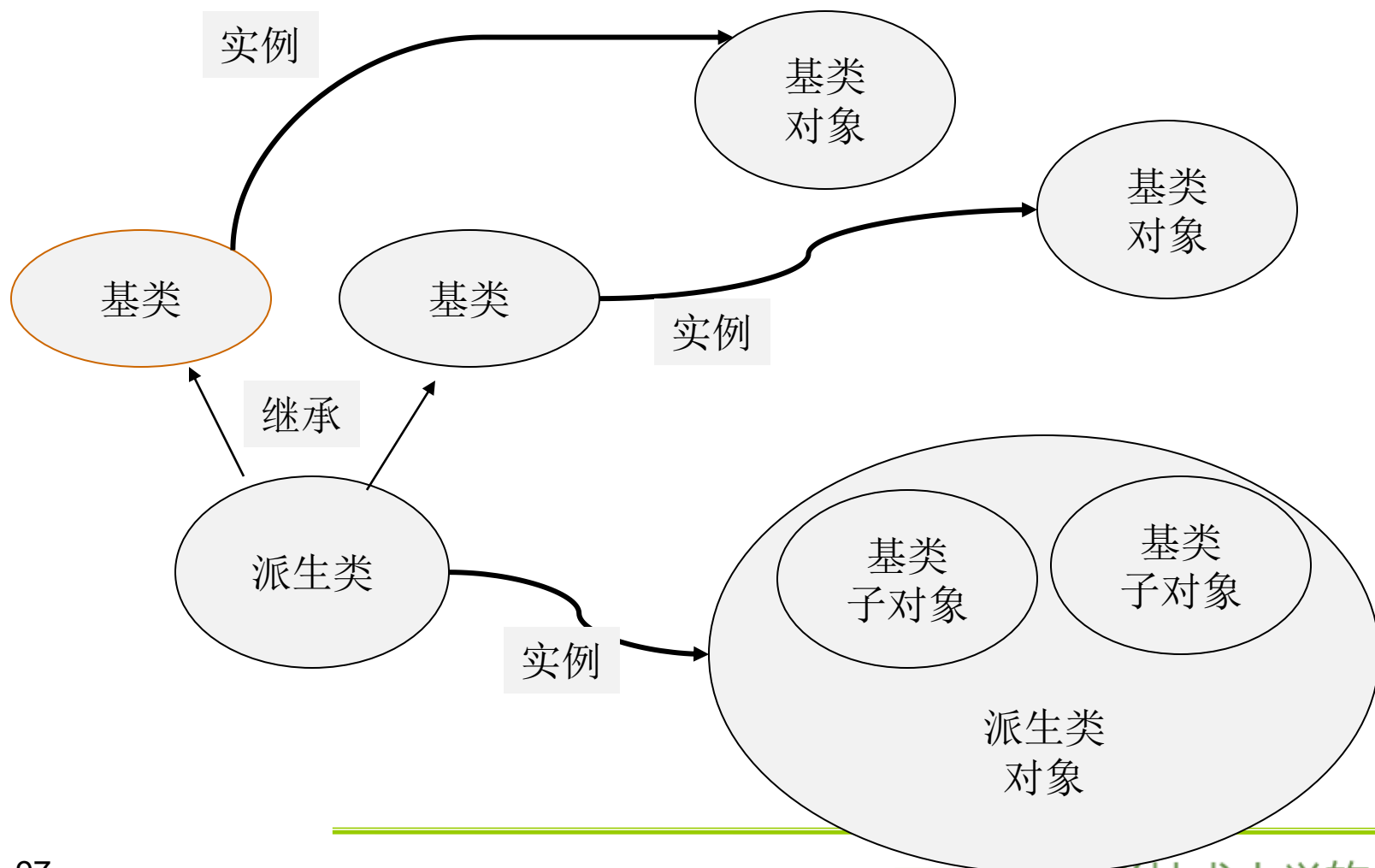
单继承



多继承



# 基类与派生类对象示意图







# 构造函数回顾

- ◆ 在派生类的对象中，由基类中声明的数据成员和函数成员所构成的封装体称为**基类子对象**。
- ◆ 基类子对象由基类中声明的构造函数进行初始化。
- ◆ 构造函数不能被继承，所以，一个派生类的构造函数必须通过调用基类的某个构造函数来初始化基类子对象。
- ◆ 派生类的构造函数只负责初始化在派生类中声明的数据成员。
- ◆ 数据成员的初始化顺序取决于它们在类中被声明的顺序，而与它们在成员初始化列表中出现的顺序无关。



## 例：观察派生类中成员初始化顺序

```
class B { public: B(int pintT) { cout<<"test"<<pintT<<"\t"; } };
class C : public B {
    public: C( int i, int j ,int k ) : B( i ), member(j)
        { cout<<k<<endl; }
    private: B member; };
int main() { C c(1,2,3); return 0; }
-----
test1          test2    3
```

### ◆ 定义派生类的一般格式：

```
class 派生类名：访问控制(继承方式) 基类名1，访问控制 基类名2
    ， ...访问控制 基类名n
{ };
```





## 4.7.1 继承和访问规则

---

- ◆ 派生类构造函数执行顺序是先执行所有基类的构造函数，再执行派生类本身构造函数。
- ◆ 在多继承情况下，基类构造函数的执行顺序按它们在被继承时所声明的顺序（从左到右）依次调用，与它们在初始化列表中的顺序无关。



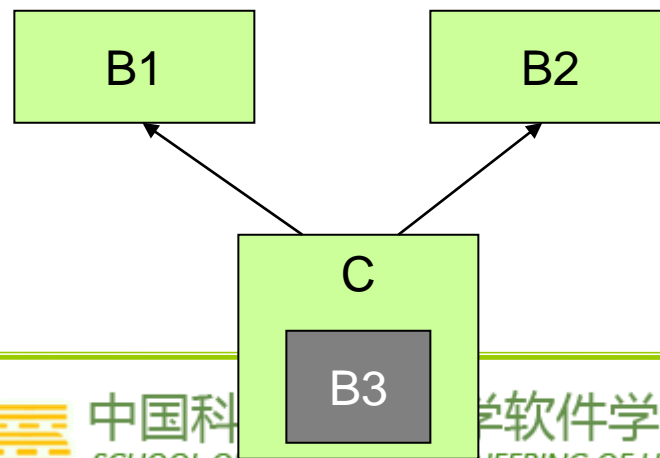


## 例：多继承构造函数调用

```
class B1{ public: B1(int i) {cout<<"constructing B1"<<endl; } };
class B2{ public: B2(int j) {cout<<"constructing B2"<<endl; } };
class B3{ public: B3(int k) {cout<<"constructing B3"<<endl;} };
class C: public B2, public B1 {
    public: C(int a, int b, int c):B1(a), B2(b), member(c)
        { cout<<"constructing C"<<endl; }
    private: B3 member; };
int main() { C obj(1,2,3);    return 0; }
```

结果：

```
constructing B2
constructing B1
constructing B3
constructing C
```





# 多继承构造函数调用顺序

---

- ◆ 基类的构造函数被先调用（按声明时的顺序），数据成员所在类的构造函数次之，最后执行派生类的构造函数。





# 多继承机制下的命名冲突

## ◆ 命名冲突

由于多继承，可能造成对基类中某个成员的访问出现了不唯一的情况，称为对基类成员访问的二义性问题。

## ◆ 表现形式有两种：

- 派生类和某个基类之间发生命名冲突。(前述的 名字隐藏)
- 基类与基类之间发生命名冲突。

同名成员的二义性

## ◆ 解决命名冲突问题是程序员的责任。





# 多继承机制下的命名冲突

## ◆ 基类与基类之间发生命名冲突。

同名成员的二义性。在多重继承中，如果不同基类中有同名的函数，则在派生类中就有同名的成员，这种成员会造成二义性。

例：

```
class A {  
    public:  
        void f ( );  
};  
class B {  
    public:  
        void f ( );  
        void g ( );  
};
```





# 多继承机制下的命名冲突

---

```
class C : public A, public B {  
    public:  
        void g ( ); //派生类和某个基类之间发生命名冲突  
        void h ( );  
};  
int main( )  
{  
    .....  
    C obj;  
    obj.g(); //ok, C中的g ( ) , 非B中的g ( )  
    obj.f ( ); //无法确定访问A中或是B中的f ( )  
    .....  
}
```







## (续)

解决方法：使用基类名

```
int main( )
{   C obj;
    obj.g( );
    obj.A::f( ); //A中的f();
    obj.B::f( ); //B中的f();
    return 0;
}
```

C类的成员访问 f() 时也必须避免这种二义。

以上这种用基类名来控制成员访问的规则称为支配原则。

如：  
obj.g(); //隐含用C的g()  
obj.B::g(); //用B的g()

以上两个语句无二义。





# 多继承机制下的命名冲突

- ◆ 如果同一个成员名在两个具有继承关系的类中进行了定义，那么，在派生类中所定义的成员名具有支配地位。在出现二义性时，如果存在具有支配地位的成员名，那么编译器将使用这一成员，而不是给出错误信息。

```
class A { public: void a( ){} };  
class B : public A { public: void a( ){} };  
class C: public A,public B { };  
//.....  
C c1;  
c1.a() //不具有二义性，访问B中的a
```





## 4.7.1 继承和访问规则

### ◆ 例4-19 p163

```
class BC1 {  
    private:  
        int x;  
    public:  
        void set_x(int a) { x=a; }  
};  
class BC2 {  
    private:  
        int x;  
    public:  
        void set_x(int a) { x=a; }  
};
```

```
class DC:public BC1,public BC2  
{ private:  
    int x;  
    public:  
        void set_x(int a) {x=a;}  
};  
int main()  
{ DC d1;  
    d1.set_x(999);  
    d1.BC1::set_x(111);  
    d1.BC2::set_x(222);  
}
```





**BC1**

**- int x  
+ set\_x**

**BC2**

**- int x  
+set\_x**

d1

BC1::x

111

BC2::x

222

DC::x

999

+

**DC**

+

**BC1**

**X int x  
+ set\_x**

**- int x  
+ set\_x**

**X int x  
+ set\_x**

**BC2**





## 4.7.2 虚基类

- ◆ 多继承层次结构可能非常复杂，由于二义的原因，一个类不能从同一类直接继承多次。

如：

```
class derived:public base,public base
{
//...
}
```

是错的。

如必须这样，可使用中间类：即派生类从同一个间接基类继承了多次。



## 虚基类

例 4.20 p.164

```
class BC{  
    public:  
        int x;  
};  
class DC1:public BC { };  
class DC2:public BC { };  
class Z:public DC1,public DC2  
{ }
```

```
int main()  
{ Z z;  
    z.DC1::x=1;  
    z.DC2::x=2;  
    cout<<z.DC1::x<<endl;  
    cout<<z.DC2::x<<endl;  
}
```

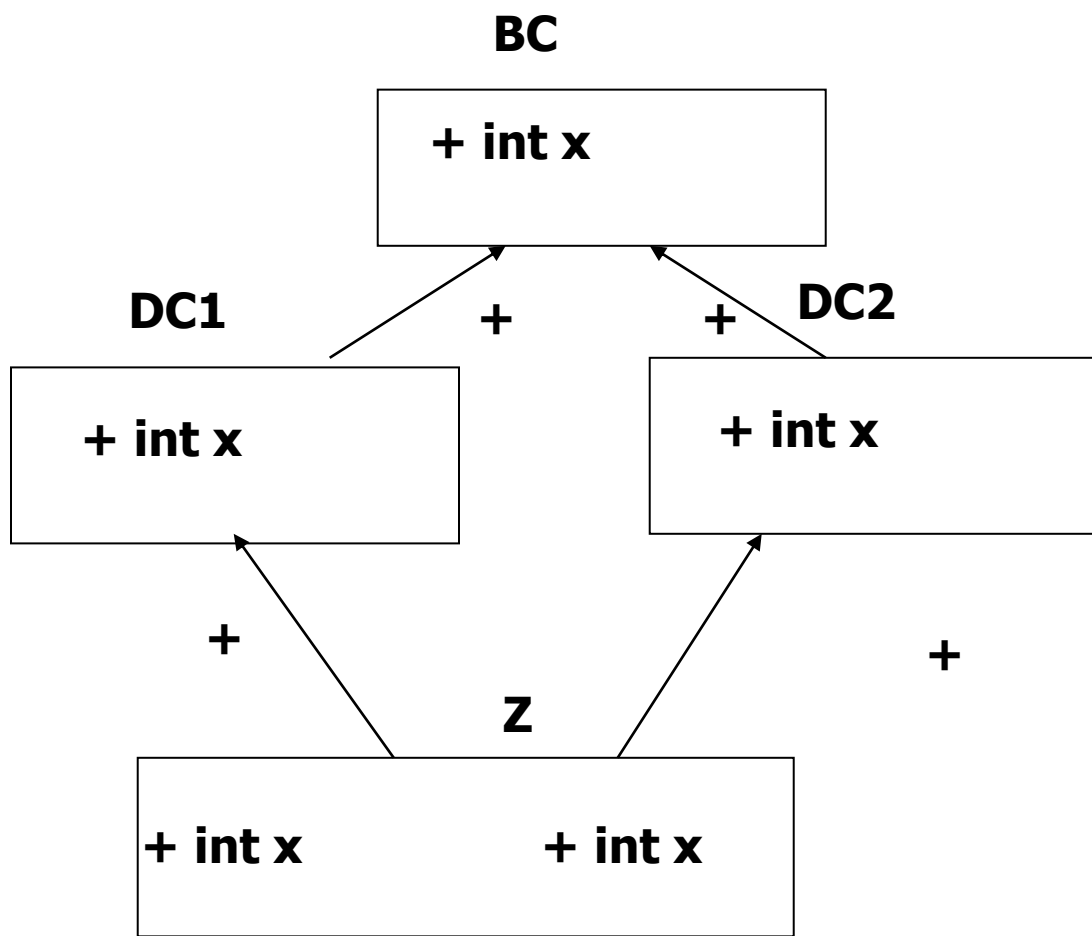
输出： 1  
          2

构造函数顺序： BC, DC1, BC, DC2, z





# 虚基类



Z z;

z.x=1;

z.BC::x=1;

z.DC1::x=1;

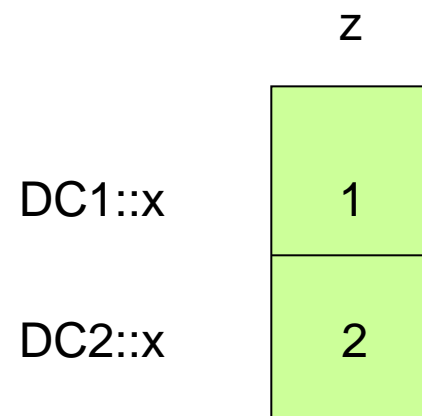
z.DC2::x=2;

×

×

}

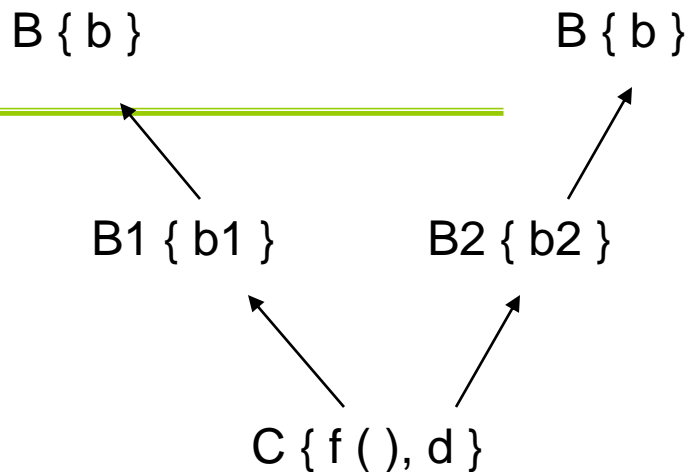
二  
义  
性





(再如)

```
class B {  
    public:  
        int b; };  
class B1 : public B {  
    private:  
        int b1; };  
class B2 : public B {  
    private:  
        int b2; };  
class C : public B1, public B2 {  
    public:  
        int f ( );  
    private:  
        int d; };
```



下面对b的访问是错的

C c;

c.b

c.B::b

下面对b的访问是对的

C c;

c.B1::b

c.B2::b

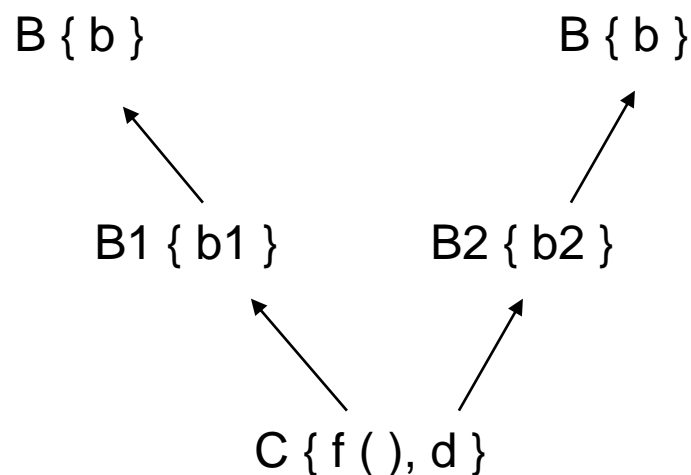






# 虚基类

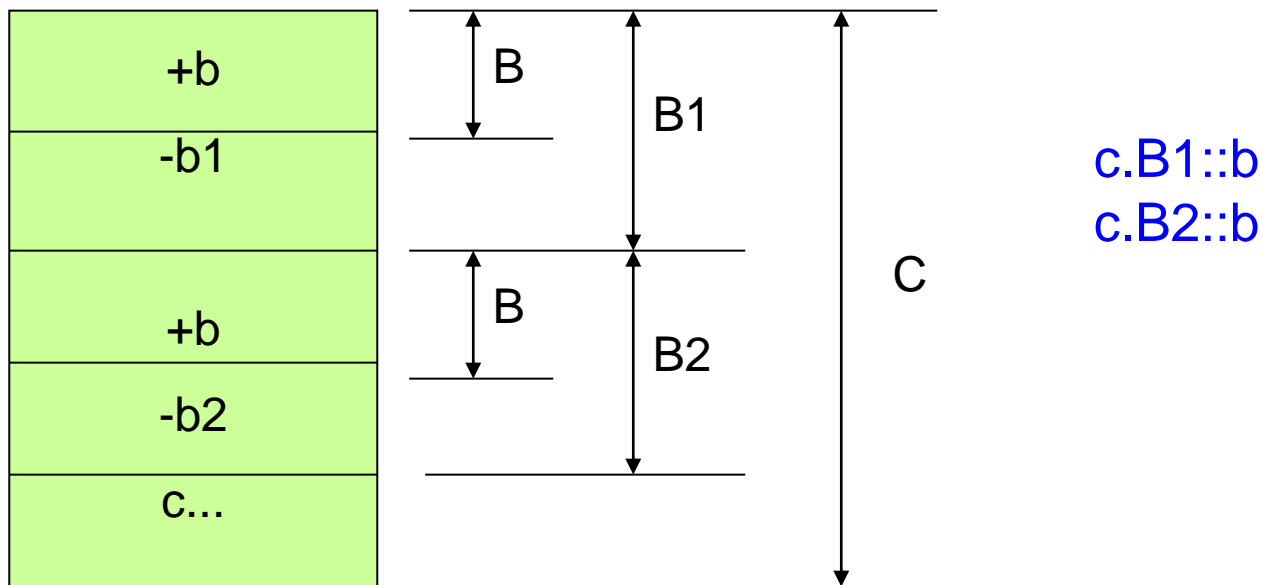
- ◆ 从右图中可以看出：**C**类的对象包含基类**B**的两个基类子对象：一个是由**B1**继承路径产生的，一个是由**B2**继承路径产生的。





## 派生类C的对象的存储结构如下：

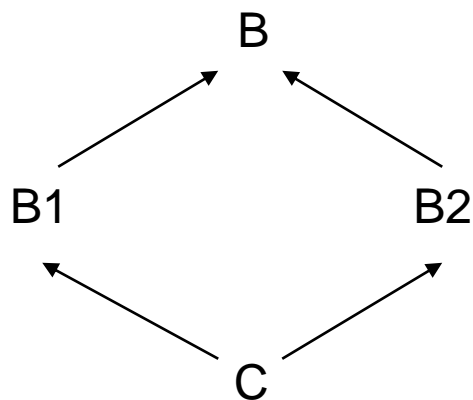
- ◆ 基类子对象B在派生类C中存储2份，访问这2个子对象的成员时，需要给出访问路径，这样才能避免二义性问题。





# 虚基类

- ◆ 在多条继承路径上有一个公共的基类时，如果希望只存储一个公共基类，可利用虚基类机制。



```
class B { public: int b; };  
class B1:virtual public B { private: int b1; };  
class B2:virtual public B { private: int b2; };  
class C:public B1, public B2 {private: int c; };
```



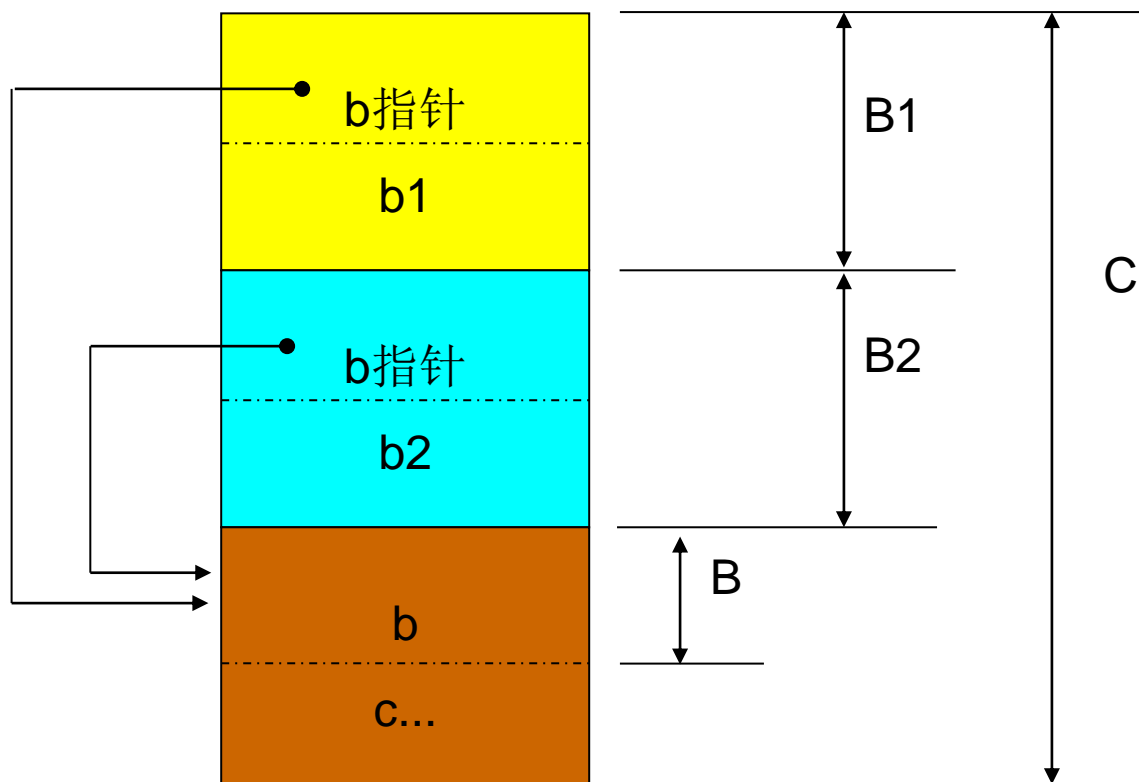
## 虚基类

- ◆ 上例中，**B1**类和**B2**类在从**B**类派生时，使用关键字**virtual**指明将**B**类作为它们的虚基类。这样，在建立**C**类的一个对象时，这些同名的虚基类在该对象中只产生一个虚基类子对象。





# 带有虚基类的派生类C的对象的存储结构示意图



```
class B { public: int b; };  
class B1:virtual public B { private: int b1; };  
class B2:virtual public B { private: int b2; };  
class118 C:public B1, public B2 {private: int c; };
```





# 虚基类

```
class BC{ public: int x; };
class DC1:virtual public BC{ };
class DC2:virtual public BC{ };
class Z:public DC1,public DC2{ };
int main()
{ Z z;
  z.DC1::x=1;
  z.DC2::x=2;
  cout<<z.DC1::x<<endl;
  cout<<z.DC2::x<<endl;
}
```

输出:

2  
2

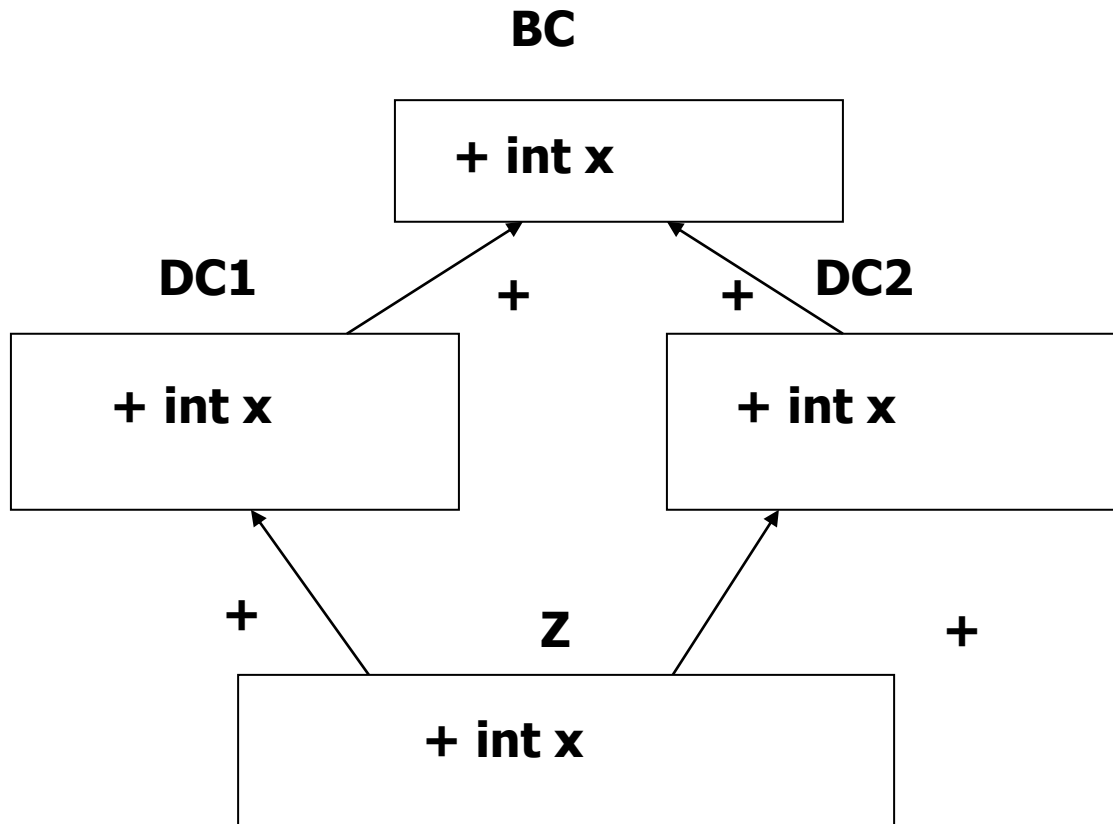
构造函数顺序: BC, DC1, DC2, z





# 虚基类

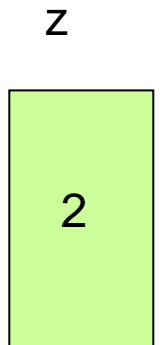
```
class BC{ public: int x; };  
class DC1:virtual public BC{ };  
class DC2:virtual public BC{ };  
class Z:public DC1,public DC2{ };
```



```
Z z;  
z.x=3;  
z.BC::x=4;  
z.DC1::x=1;  
z.DC2::x=2;
```

一  
回  
事

DC1::x  
DC2::x



◆ 理解下列程序输出

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

```
class B
```

```
{ public: B():b(5) {};  
  int b; };
```

```
class B1:virtual public B
```

```
{ public:void t() {cout<<b<<endl; } };
```

```
class B2:virtual public B
```

```
{ public:void t() {cout<<b<<endl; } };
```

```
class C:public B1, public B2
```

```
{ public:void t() {cout<<b<<'\t'<<B1::b<<'\t'<<B2::b<<endl;} };
```

```
int main()
```

```
{ B1 b1; b1.t();    B2 b2; b2.b=20; b2.t();  b1.t();
```

```
  C c;    c.b=10;  c.t();  c.B1::b=1; c.t();
```

```
  return 0; }
```

执行结果：（虚基类理解.cpp）

**5**

**20**

---

**5**

**10      10      10**

**1      1      1**





```
#include <iostream>
using namespace std;
class B
```

```
{ public: B() :b(5) {};  
  int b; };
```

```
class B1 :virtual public B
```

```
{ public: void t() { b = 15; cout << b << endl; }  
  int b; };
```

```
class B2 :virtual public B
```

```
{ public: void t() { cout << b << endl; } };
```

```
class C :public B1, public B2
```

```
{ public: void t() { cout << b << '\t' << B1::b << '\t' << B2::b << endl; }  
};
```

这个b指的是B1::b

```
int main()
```

```
{ B1 b1; b1.t();          B2 b2; b2.b = 20; b2.t();          b1.t();
```

```
  C c;  c.b = 10;  c.t(); c.B1::b = 1;    c.t();
```

```
  cout<<"*"<<c.B1::B::b;  return 0; }
```

执行结果：（虚基类理解.cpp）

15

20

15

10      10      5

1      1      5

\*5





# 虚基类的特性

- ◆ 虚基类既可以减少公共基类的存储个数，也解决多份存储带来的二义性问题。
- ◆ **virtual**关键字只对紧随其后的基类名起作用。一个类可以在一个类格中既被用作虚基类，也可以被用作非虚基类。

```
class B { };
```

```
class B1 :virtual public B { }
```

```
class B2 :virtual public B { }
```

```
class B3 :public B { }
```

//下句，语法不错，但除了增加复杂性，无意义。

```
class C :public B1, public B2, public B { }
```

下面语法错误：

```
class B1 :virtual public B, public B { } //Error
```



# 虚基类

- ◆ **C++**规定，在一个成员初始化列表中出现对虚基类和非虚基类构造函数的调用，则虚基类的构造函数先于非虚基类的构造函数的执行。

```
class B { public: B() { cout << "B,"; } };
class B1 :virtual public B { public:B1() { cout << "B1,"; } };
class B2 : virtual public B { public:B2() { cout << "B2,"; }};
class B3 { public:B3(int x) { cout << x<<","; } };
class C :public B3,public B1, public B2 {
    public: C():B3(5) { cout << "C"; } };
int main()
{ C c; return 0; }
```

运行结果： B,5,B1,B2,C





# 虚基类

- ◆ 虚基类的构造函数须由最派生类来调用，而对于其他的派生类则跳过对虚基类构造函数的调用。详细说明如下：

- 最派生类

由于继承结构的层次可能很深，规定类层次中，将在建立对象时所指定的类称为最派生类。

为初始化基类子对象，派生类的构造函数要调用基类的构造函数。对于虚基类来说，其构造函数只能被调用一次。

- C++规定，虚基类子对象由最派生类的构造函数通过调用虚基类的构造函数进行初始化。

(continue)





# 虚基类

- 在类层次中，从叶子到根的路径上，所有结点都可能被指定为最派生类，因此，**C++**规定，在一个派生类中只要含有虚基类（无论是直接或是间接的），则在其构造函数的初始化列表中都应列出对该虚基类构造函数的调用；如果未列出，则表示是使用该虚基类的缺省构造函数进行初始化。
- 在创建对象时，只有最派生类对虚基类构造函数的调用会被真正执行，而其他基类的虚基类构造函数调用则会被忽略。这样便保证了对虚基类的对象只初始化一次。

如下例：





## 例：虚基类构造函数的调用。

```
class A
{
    int a;
public:
    A(int i) { a=i; cout<<"constructing  A.\n"; }
    ~A() { cout<< "destroying  A.\n"; }
    void  print() { cout<<a; }
};

class B1:virtual public A
{
    int b1;
public:
    B1(int i,int j):A(i) { b1=j; cout<<"constructing  B1.\n";}
    ~B1() { cout<< "destroying B1.\n"; }
    void  print(){ cout<<b1; }
};
```



## 例：虚基类构造函数的调用。

---

```
class B2:virtual public A
{ public:
    B2(int i, int j):A(i) { b2=j; cout<<"constructing  B2.\n"; }
    ~B2(){ cout<<"destroying  B2.\n"; }
    void  print(){ cout<<b2; }
private:
    int  b2;
};
```





## 例：虚基类构造函数的调用。

---

```
class C:public B1,public B2
{   public:
    C(int i,int j,int k,int m,int n):B1(i,j),B2(k,m),A(i)
        { c=n; cout<<"constructing  C.\n"; }
    ~C(){cout<<"destroyiny  C\n";}
    void  print()    {
        A::print();   //引用正确，不会产生二义性
        B1::print();   B2::print();
        cout<<c<<endl;
    }
private :
    int  c;
};
```

---







## 例：虚基类构造函数的调用。

---

```
void main()
{   C c2(1,2,3,4,5);   c2.print(); }
```

程序运行的结果为：

```
constructing A.
constructing B1.
constructing B2.
constructing C.
1245
destroyiny C
destroying B2.
destroying B1.
destroying A.
```





# 练习

---

```
class A //分析以下程序的执行结果
{
    public:    int n;    };
class B:public A{};
class C:public A{};
class D:public B,public C
{
    int getn() { return B::n; }    };
void main()
{
    D d;
    d.B::n=10;    d.C::n=20;
    cout<<d.B::n<<","<<d.C::n<<endl;
}
```

输出：10 , 20





# 练习

◆ 多继承的构造顺序可分为如下4步。

- ① 所有非虚基类的构造函数按照它们被继承的顺序构造；
- ② 所有虚基类的构造函数按照它们被继承的顺序构造；
- ③ 所有子对象的构造函数按照它们声明的顺序构造；
- ④ 派生类自己的构造函数体；

这4个步骤的正确顺序是（ C ）。

A、④③①②

B、②④③①

C、②①③④

D、③④①②

◆ 设置虚基类的目的是（ B ）。

A、简化程序

B、消除二义性

C、提高运行效率

D、减少目标代码





# 练习

◆ 关于多继承二义性的描述中错误的是（ C ）。

A、一个派生类的两个基类中都有某个同名成员，在派生类中对这个成员的访问可能出现二义性

B、解决二义性的最常用的方法是对成员名的限定法

C、基类和派生类中同时出现的同名函数，也存在二义性问题  
派生类的函数会对基类的函数实现名字隐藏

D、一个派生类是从两个基类派生出来的，而这两个基类又有一个共同的基类；对该基类成员进行访问时，可能出现二义性

◆ 下列虚基类的声明中，正确的是（ D ）。

A、class virtual B: public A      B、virtual class B: public A

C、class B: public A virtual      D、class B: virtual class A





## 练习

- ◆ 带有虚基类的多层派生类构造函数的成员初始化列表中都要列出虚基类的构造函数，这样将对虚基类的子对象初始化（ D ）。
  - A、与虚基类下面的派生类个数有关
  - B、多次
  - C、两次
  - D、一次
- ◆ 下列关于虚基类的描述中，错误的是（ B ）。
  - A、虚基类子对象的初始化由最派生类完成
  - B、虚基类子对象的初始化次数与虚基类下面的派生类个数~~有关~~
  - C、设置虚基类的目的是消除二义性
  - D、带有虚基类的多层派生类构造函数的成员初始化列表中都要列出对虚基类构造函数的调用





## 练习

- ◆ 继承具有（**B**），即当基类本身也是某一个类派生类时，底层的派生类也会自动继承间接基类的成员。
  - A. 规律性
  - B. 传递性
  - C. 重复性
  - D. 多样性
- ◆ 多继承派生类构造函数构造对象时，（**B**）被最先调用。
  - A. 派生类自己的构造函数
  - B. 虚基类的构造函数
  - C. 非虚基类的构造函数
  - D. 派生类中子对象类的构造函数
- ◆ 下列虚基类的声明中，正确的是（**D**）。
  - A. `class virtual B:public A`
  - B. `virtual class B:public A`
  - C. `class B:public A virtual`
  - D. `class B: virtual public A`





## 4.8 派生类的继承方式

- ◆ 不同的继承方式决定了基类成员在派生类中的访问属性。
- ◆ 派生类对基类的继承方式有3种：
  - 公用继承(public inheritance)

基类的公用成员和保护成员在派生类中保持原有访问属性，其私有成员仍为基类私有。
  - 私有继承(private inheritance)

基类的公用成员和保护成员在派生类中成了私有成员。其私有成员仍为基类私有。
  - 受保护的继承(protected inheritance)

基类的公用成员和保护成员在派生类中成了保护成员，其私有成员仍为基类私有。

保护成员的意思是：不能被外界访问，但可被派生类的成员访问。





## 4.8.1 私有继承

- ◆ 私有继承（**private inheritance**）的性质如下：
  - 基类中所有公有成员在派生类中是私有的。
  - 基类中所有保护成员在派生类中是私有的。
  - 基类中所有私有成员仅在基类中可见。
- ◆ 在声明一个派生类时将基类的继承方式指定为**private**的。
- ◆ 用私有继承方式建立的派生类称为私有派生类(**private derived class**)，其基类称为私有基类(**private base class**)。







## 4.8.1 私有继承

---

```
class Base    //声明基类
{ public:     //基类公用成员
    void setx(int n) { x=n; }
    int getx() { return x; }
    void showx() { cout<<"x="<<x<<endl; }
private:     //基类私有成员
    int x;
};
```





## 4.8.1 私有继承

```
class Derived: private Base //以private方式声明派生类Derived
{ public:
    void sety(int n) { y=n; }
    int gety() { return y; }
    void showy() { cout<<"y="<<y<<endl; }
    void set(int m,int n)
    {   setx(m); //在Derived中x不可访问
        y=n;   }
    void show()
    {   cout<<"x="<<getx()<<endl; cout<<"y="<<y<<endl; }
private:
    int y;
};
```





## 4.8.1 私有继承

---

```
int main()
{   Derived obj;
    //obj.setx(10); //不可访问, 私有
    //obj.showx(); //不可访问, 私有
    obj.sety(20);
    obj.showy();
    obj.set(30,40);
    obj.show();
    return 0;
}
```

程序运行结果如下:

y=20

x=30

y=40





## 4.8.2 保护继承

- ◆ 保护继承（**protected inheritance**）的性质如下：
  - 基类中所有公有成员在派生类中是保护成员。
  - 基类中所有保护成员在派生类中是保护成员。
  - 基类中所有私有成员仅在基类中可见。
- ◆ 由**protected**声明
- ◆ 从类的用户角度来看，保护成员等价于私有成员。但有一点与私有成员不同，保护成员可以被派生类的成员函数访问。





## 4.8.2 保护继承

---

```
#include <iostream>
class Base    //声明基类
{ public:     //基类公用成员
    void setx(int n){ x=n; }
    void showx() { cout<<"x="<<x<<endl; }
protected:  //基类保护成员
    int getx() { return x; }
private:    //基类私有成员
    int x;
};
```





## 4.8.2 保护继承

//以protected方式声明派生类Derived

class Derived: **protected** Base

{ public:

void sety(int n){ y=n; }

int gety(){ return y; }

void showy(){ cout<<"y="<<y<<endl; }

void set(int m,int n) {

setx(m); //在Derived中x不可访问

y=n; }

void show()

{ cout<<"x="<<getx()<<endl; cout<<"y="<<y<<endl; }

private:

int y;

};





## 4.8.2 保护继承

---

```
int main()
{   Derived obj;
    //obj.setx(10); //不可访问, 保护
    //obj.showx(); //不可访问, 保护
    obj.sety(20);
    obj.showy();
    obj.set(30,40);
    obj.show();
    return 0;
}
```

程序运行结果如下:

y=20

x=30

y=40

◆ 下面是继承机制的再举例





# 继承机制举例

- ◆ 一个类Location，被作为基类 （继承机制）

```
class Location {  
    public:  
        void InitL( int xx, int yy );  
        void Move( int xOff, int yOff );  
        int GetX( ) { return X; }  
        int GetY( ) { return Y; }  
    private:  
        int X, Y; };  
void Location::InitL( int xx ,int yy ) {  X = xx;  Y = yy; }  
void Location::Move( int xOff, int yOff ) {  
    X += xOff;          Y += yOff;    }
```







# 继承机制举例

---

- ◆ 使用**public**继承，定义如下：

```
class Rectangle : public Location    {  
    public:  
        void InitR( int x, int y, int w, int h );  
        int GetH( ) { return H; }  
        int GetW( ) { return W; }  
    private:  
        int W, H;  
};  
void Rectangle::InitR( int x, int y, int w, int h ) {  
    InitL( x, y );      W = w; H = h;  
}
```





## 继承机制举例

- ◆ Rectangle继承了Location的行为:

```
int main( )
{   Rectangle rect;
    rect.InitR( 2, 3, 20, 10 ); // 间接调用基类的方法
    rect.Move( 3, 2 ); // 调用基类中声明的方法
    cout << rect.GetX() << ',' << rect.GetY() << ','
        << rect.GetW() << ',' << rect.GetH() << endl;
    return 0;
}
```

- ◆ 运行结果:

5 5 20 10





# 继承机制举例

---

- ◆ 使用**private**继承，定义如下：

将上例的继承结构改为**private**，即

```
class Rectangle : private Location {  
    .....  
}
```

编译时将出错，派生类的对象不能访问基类中的方法。

要使**Rectangle**类有用，即能对**Rectangle**类的对象进行必要的操作，可以对**Rectangle**类进行如下修改：





# 继承机制举例

---

```
class Rectangle : private Location {  
    public:  
        void InitR( int x, int y, int w, int h );  
        void Move( int xOff, int yOff );  
        int GetX( ) { return Location::GetX(); }  
        int GetY( ) { return Location::GetY(); }  
        int GetH( ) { return H; }  
        int GetW( ) { return W; }  
    private:  
        int W, H;  
};
```





## 继承机制举例

---

```
void Rectangle::InitR( int x, int y, int w, int h ) {  
    InitL( x, y );  
    W = w;  
    H = h;  
}  
  
void Rectangle:: Move( int xOff, int yOff ) {  
    Location::Move(xOff, yOff );  
}
```

◆ 注意：下面的写法将会导致死循环

```
void Rectangle:: Move( int xOff, int yOff )  
{ Move(xOff, yOff ) }
```





# 总结

- ◆ 在派生类中，成员有**3**种不同的访问属性：
  - 公用的：派生类内和派生类外都可以访问。
  - 受保护的：派生类内可以访问，派生类外不能访问，其下一层的派生类可以访问。
  - 私有的：派生类内可以访问，派生类外不能访问。

基类中的成员	在 <b>公用派生类</b> 中的访问属性	在 <b>私有派生类</b> 中的访问属性	在 <b>保护派生类</b> 中的访问属性
私有成员	<b>不可访问</b>	<b>不可访问</b>	<b>不可访问</b>
公用成员	<b>公用</b>	<b>私有</b>	<b>保护</b>
保护成员	<b>保护</b>	<b>私有</b>	<b>保护</b>



## 练习

◆ 有如下程序：

```
class Base{
    private: void fun1( ) const {cout<<"fun1";}
    protected: void fun2( ) const {cout<<"fun2";}
    public: void fun3( ) const {cout<<"fun3";} };
class Derived : protected Base{
    public: void fun4( ) const {cout<<"fun4";} };
int main(){
    Derived obj; obj.fun1( ); //①
    obj.fun2( ); //②
    obj.fun3( ); //③
    obj.fun4( ); //④ }
```

其中没有语法错误的语句是（ ）。





# 练习

- ◆ 设有基类定义：

```
class Cbase {  
    private: int a;  
    protected: int b;  
    public: int c; };
```

派生类采用何种继承方式可以使成员变量**b**成为自己的私有成员(A)

- A. 私有继承
- B. 保护继承
- C. 公有继承
- D. 私有、保护、公有均可

- ◆ 派生类的对象对它的哪一类基类成员是可以访问的？ (A)

- A. 公有继承的基类的公有成员
- B. 公有继承的基类的保护成员
- C. 公有继承的基类的私有成员
- D. 保护继承的基类的公有成员







# 练习

◆ 关于保护继承，下列说法错误的是（C）。

A. 保护继承的特点是基类的所有公用成员和保护成员都成为派生类的保护成员。

B. 派生类对象不能访问基类中的任何成员。

C. 派生类的对象可以访问基类的公有成员。

D. 保护继承的派生类和私有继承的派生类，对基类成员访问属性是相同的。

◆ 关于私有继承，下列说法错误的是（B）。

A. 基类的公有成员和保护成员被继承后作为派生类的私有成员，派生类的其他成员可以直接访问他们。

B. 基类的公有成员和保护成员被继承后作为派生类的私有成员，派生类的其他成员不能直接访问他们。

C. 基类的私有成员，派生类的成员和派生类的对象都无法访问

D. 派生类的对象无法访问基类的所有成员。





# 练习

◆ 下面叙述错误的是（A）。

- A. 基类的protected成员在派生类中仍然是protected
- B. 基类的protected成员在public派生类中仍然是protected的
- C. 基类的protected成员在private派生类中是private的
- D. 基类的protected成员不能被派生类的对象访问

◆ 派生类的对象对它的基类成员中（A）是可以访问的。

- A. 公有继承的公有成员
- B. 公有继承的私有成员
- C. 公有继承的保护成员
- D. 私有继承的公有成员

下列对派生类的描述中，错误的是（D）。

- A. 一个派生类可以作为另一个派生类的基类
- B. 派生类至少有一个基类
- C. 派生类的缺省继承方式是private
- D. 派生类只含有基类的公有成员和保护成员





# 练习

- ◆ 在公有继承中，基类中的保护成员作为派生类的（C）。
  - A. 公有成员    B. 私有成员
  - C. 保护成员    D. 私有成员函数
- ◆ 基类（A）在派生类中的性质和继承的性质一样。
  - A. 公有成员    B. 私有成员    C. 保护成员    D. 私有成员函数
- ◆ 派生类的对象对它的基类成员中的（A）是可以访问的。
  - A. 公有继承的公有成员    B. 公有继承的私有成员
  - C. 公有继承的保护成员    D. 私有继承的私有成员





# 补充：子类型化和类型适应

---

## ◆ 子类型化

- 类型化的概念涉及到行为共享，它与继承有着密切的关系
- 有一个特定的类型**S**，当且仅当它至少提供了类型**T**行为，则称类型**S**是类型**T**的子类型
- 子类型是类型之间一般和特殊的关系

## ◆ 例：





## 补充：子类型化和类型适应

---

```
class A
{   public:
    void Print( ) const { cout << "A::print( )called.\n";}
};

class B : public A
{   public:
    void f( ) { }
};
```

- 类 B 公有继承了类 A，类 B 是类 A 的一个子类型
- 类 B 是类 A 的子类型，类 B 具备类 A 中的操作
- 类 A 中的操作可以被用于操作类 B 的对象





## 补充：子类型化和类型适应

---

```
void f1(const A & r)
{  r.Print( ); }
void main( )
{  B b;
    f1(b);
}
```

执行该程序将会输出如下结果：

A::Print( ) called.

类B的对象b交给了处理类A的对象的函数f1( )进行处理。  
对类A的对象操作的函数，可以对类A的子类的对象进行操作。  
子类型关系是不可逆的,子类型关系是不对称的。





# 补充：子类型化和类型适应

## ◆ 类型适应

- 类型适应是指两种类型之间的关系
- 派生类的对象可以用于基类对象所能使用的场合，我们说派生类适应于基类
- 派生类对象的指针和引用也适应于基类对象的指针和引用
- 子类型化与类型适应是一致的。A类型是B类型的子类型，那么A类型必将适应于B类型。
- 子类型的重要性就在于减轻程序人员编写程序代码的负担
- 一个函数可以用于某类型的对象，则它也可用于该类型的各个子类型的对象
- 不必为处理这些子类型的对象去重载该函数





## 补充：子类型化和类型适应

---

### ◆ 例

```
class A
{ public:
    A() { a = 0; }
    A(int i) { a = i; }
    void print() { cout << a << endl; }
    int geta() { return a; }
private:
    int a;
};
```







## 补充：子类型化和类型适应

---

```
class B : public A
{ public;
    B() { b = 0; }
    B(int i, int j) : A(i), b(j) { }
    void print() { A::print( ); cout << b << endl; }
private:
    int b;
};

void fun(A& d)
{ cout << d.geta( ) * 10 << endl;
}
```





## 补充：子类型化和类型适应

---

```
void main( )
{  B bb(9,5);  //bb.a=9, bb.b=5
   A aa(5);    //aa.a=5
   aa = bb;    //aa.a=9
   aa.print( ); // Output: 9
   A *pa = new A(8);  //pa->a=8
   B *pb = new B(1,2); //pb->a=1, pb->b=2
   pa = pb;          //pa->a=1
   pa->print( );  //Output: 1 没有多态
   fun(bb);        //Output: 90
}
```





## 补充：子类型化和类型适应

---

◆ Ans:

9

1

90

解释:

aa = bb; pa = pb;          合法

bb = aa; pb = pa;          非法





# 补充：子类型化和类型适应

## 赋值兼容规则

赋值兼容规则是指：在公有派生的情况下，一个派生类的对象可用于基类对象适用的地方。赋值兼容规则有三种情况（假定类 **derived** 由类 **base** 派生）：

1) 派生类的对象可以赋值给基类的对象。

```
derived d; base b;  
b=d;
```

2) 派生类的对象可以初始化基类的引用。

```
derived d;  
base& br=d;
```

3) 派生类的对象的地址可以赋给指向基类的指针。

```
derived d;  
base *pb=&d;
```





## 练习

- ◆ 下面（**B**）的叙述不符合赋值兼容规则。
  - A、派生类的对象可以赋值给基类的对象
  - B、基类的对象可以赋值给派生类的对象
  - C、派生类的对象可以初始化基类的引用
  - D、派生类的对象的地址可以赋值给指向基类的指针
- ◆ 下面叙述错误的是（**C**）。
  - A、派生类可以使用**private**派生
  - B、对基类成员的访问必须是无二义性的
  - C、基类成员的访问能力在派生类中维持不变
  - D、赋值兼容规则也适用于多继承的组合





# 练习

- ◆ 下列关于子类型的描述中错误的是（ A ）。
- A、子类型关系是可逆的
  - B、公有派生类的对象可以初始化基类的引用
  - C、只有在公有继承下，派生类是基类的子类型
  - D、子类型关系是可传递的





## 4.9 常见编程错误

---

1. 实现公有继承，忘掉public
2. 类外不能访问类的保护成员，友元函数除外
3. 类外不能访问类的私有成员，友元函数除外
4. 如基类有构造函数，但没有默认构造函数，则派生类构造函数必须显式地在其初始化阶段中调用
5. 调用隐藏的基类成员

