第5讲 类的完善

目录CONTENTS

1 this指针

2 静态成员

3 友元

4 类的组合

1 this指针

```
#include <iostream>
using namespace std;
class CScore{
  private:
    int m_nMidtermExam;                    //私有数据成员
    int m_nFinalExam;                      //私有数据成员
  public:
    CScore(int m=0,int f=0);               //声明构造函数CScore()的原型
    void SetScore(int m,int f);
    void ShowScore ();
};
CScore::CScore(int m,int f): m_nMidtermExam(m),m_nFalExam(f)  //定义构造函数
 {  }
void CScore::SetScore(int m,int f)
{          m_nMidtermExam=m;   m_nFinalExam=f;  }
```

```
int main()
{
    CScore  op1,op2;
    op1.SetScore(80,88);
    op2.SetScore(90,92);
    op1.ShowScore ();
    op2.ShowScore ();
    return 0;
}
```

程序运行结果如下:
期中成绩:80
期末成绩:88
总评成绩:85

期中成绩:90
期末成绩:92
总评成绩:91

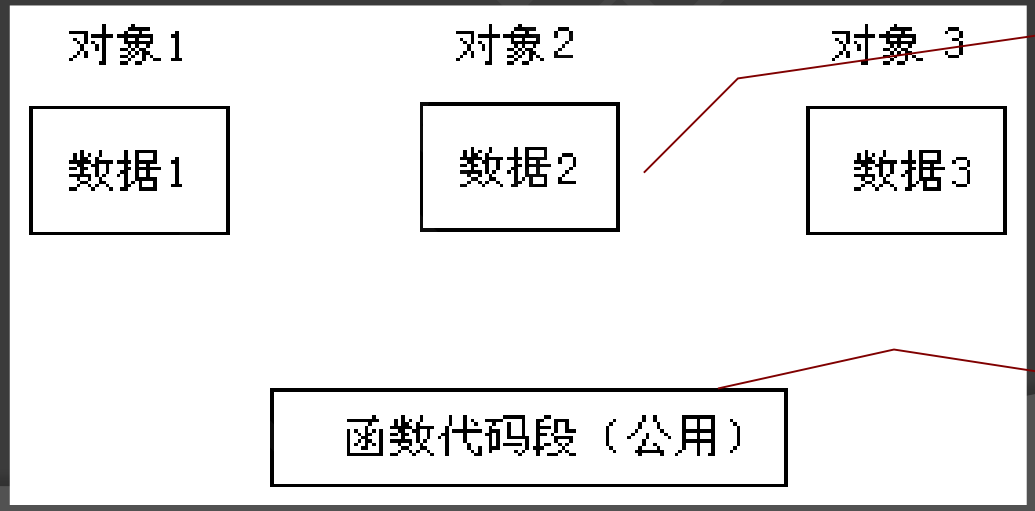
为什么两个对象执行相同的代码会得到不一样的结果？

```
void CScore::SetScore(int m,int f)
{
    m_nMidtermExam=m;  m_nFinalExam=f;
}
Void CScore::ShowScore ()
{
    cout<<"\n期中成绩："<<m_nMidtermExam<<"\n期末成绩：
"<<m_nFinalExam<<"\n总评成绩：
"<<(int)(0.3*m_nMidtermExam+0.7*m_nFinalExam)<<endl;
}
```

成员函数CScore::SetScore()与CScore::ShowScore()的代码中有没有对象的信息呢？

成员的存储方式

```
class CScore
{
……
};
CScore  对象1，对象2, 对象3;
```

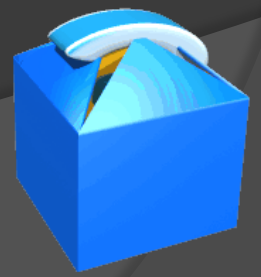


不同对象的数据成员存放在不同的内存地址

所有对象的成员函数对应的是同一个函数代码段

问题

- 一个类中所有的对象调用的成员函数都是同一代码段。
- 那么，成员函数又是怎么识别出当前调用自己的是哪个对象，从而对该对象的数据成员而不是其他对象的数据成员进行处理呢？



this指针

- C++中有一个特殊的指针this，称为自引用指针。这个指针与类密切相关。
- C++为类的每个成员函数都提供了这个隐含的名字为this的指针参数：

  类名* const this

- 指针this指向一个对象，该对象产生成员函数的调用动作。

this指针

- 在成员函数中，隐含了对this指针的使用。例如，成员函数SetScore()的定义被编译系统处理为：

```
void CScore::SetScore(CScore* const this ,int m,int f)
{
    this->m_nMidtermExam=m;
    this->m_nFinalExam=f;
}
```

this指针

- 每个对象占用的存储空间只是该对象的数据部分所占用的存储空间，而不包括函数部分所占用的存储空间。

```
void CScore ::ShowScore (CScore* const this)
{
   cout<<"\n期中成绩："<<this->m_nMidtermExam<<"\n期末成绩："<<this->m_nFinalExam<<"\n总评成绩："<<(int)(0.3*this->m_nMidtermExam + 0.7*this->m_nFinalExam)<<endl;
}
```

this指针在系统中是隐式地存在的，也可以将其显式地表示出来，但只能在非静态成员函数中使用this。

```
int main()
{
    CScore  cs1,cs2;
    cs1.SetScore(&cs1,80,88);
    op2.SetScore(&cs2,90,92);
    cs1.ShowScore (&cs1);
    cs2.ShowScore (&cs2);
    return 0;
}
```

而调用成员函数的语句则被编译系统处理成：

程序运行结果如下:

```
期中成绩:80
期末成绩:88
总评成绩:85

期中成绩:90
期末成绩:92
总评成绩:91
```

```
#include<iostream>
using namespace std;
class A{
      int x;
 public:
      A(int x1) { x=x1;}
      void disp()
     { cout<<"\nthis="<<this <<" when x="<<this->x;}
};
int main()
{     A a(1),b(2),c(3);
      a.disp();
      b.disp();
      c.disp();
      return 0;
}
```

运行结果：

```
this=0x0065FFF4    when x=1
this=0x0065FFF0    when x=2
this=0x0065FFEC    when x=3
```

```
#include <iostream>
using namespace std;
class Sample{
  private:
        int x,y;
  public:
        Sample(int i=0,int j=0){  x=i;  y=j; }
        void copy(Sample& xy);
        void print()
        {  cout<<x<<","<<y<<endl;  }
};
void Sample::copy(Sample& xy)
{         if(this==&xy) return;
          *this=xy;
}
int main()
{
    Sample p1,p2(5,6);      p1.copy(p2);      p1.print();          return 0;
}
```

this指针

- 一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。
- this只能在类的成员函数中使用。因此，获得一个对象后，也不能通过对象使用this指针。
- this在成员函数的开始执行前构造，在成员的执行结束后清除。

再来一个问题

- 关于CScore类，定义一个成员函数，用来比较两个对象的期末成绩高低，并返回较高的对象引用。

```
const CScore & BetterOne(const CScore & other)
{
    if(other.m_nFinalExam > m_nFinalExam)
        return other;
    else
        return *this
}
```

单选题 2分

设置

此题未设置答案，请点击右侧设置按钮

下列没有this指针的函数是

A 静态成员函数

B 构造函数

C 析构函数

D 成员函数

提交

2 静态成员

- 当一个类的多个对象需要共享数据时，可以利用静态数据成员和静态成员函数来实现。

- 例如，学生人数是和每一个学生对象有关的（每次新建一个学生对象，学生人数都应该增1），但它又不能只属于某一个学生对象，这时就可以将学生人数说明成类的静态数据成员，以实现学生对象之间的数据共享。

2.1 静态数据成员

【例】使用静态数据成员计算两个学生期末成绩的总成绩和平均成绩。

```
#include <iostream>
using namespace std;
class CScore{
    private:
            int m_nMidtermExam;      //私有数据成员
            int m_nFinalExam;        //私有数据成员
            static int m_nCount;     //静态数据成员，统计人数
            static float m_fSum;     //静态数据成员，统计总成绩
            static float m_fAve;     //静态数据成员，统计平均分
     public:
            CScore(int m,int f);     //声明有参数的构造函数
            CScore();                //声明无参数的构造函数
            ~CScore();               //声明析构函数
            void ShowScore ();
            void ShowCountSumAve();
};
```

```
CScore::CScore(int m,int f)              //定义有参数的构造函数CScore()
{
        m_nMidtermExam=m;
        m_nFinalExam=f;
        ++m_nCount;                                  //累加学生人数
        m_fSum=m_fSum+m_nFinalExam;          //累加期末成绩
        m_fAve=m_fSum/ m_nCount;              //计算平均成绩
}
void CScore ::ShowScore ()
{   cout<<“\n期中成绩：”<<m_nMidtermExam<<“\n期末成绩："
        <<m_nFinalExam<<"\n总评成绩："
        <<(int)(0.3* m_nMidtermExam +0.7* m_nFinalExam)<<endl;
}
void CScore:: ShowCountSumAve()
{
   cout<<"\n学生人数："<< m_nCount;          //输出静态数据成员
   cout<<"\n期末平均成绩："<<m_fAve;     //输出静态数据成员
   cout<<endl;
}
```

```
int CScore::m_nCount=0;                    //静态数员m_nCount初始化
float CScore::m_fSum=0.0f;                 //静态数员m_fSum初始化
float CScore::m_fAve=0.0f;                 //静态数员m_fAve初始化

int main()
{   CScore sc1(80,88);
                    //定义类Score的对象sc1，调用有参构造函数
     sc1.ShowScore();
                    //调用成员函数showScore()，显示sc1的数据
    sc1. ShowCountSumAve();
    CScore sc2(90,97);
                    //定义类Score的对象sc2，调用有参构造函数
    sc2.ShowScore();
                    //调用成员函数showScore()，显示sc2的数据
    sc2. ShowCountSumAve();
    return 0;
}
```

静态数据成员的定义

- 在一个类中，将一个数据成员说明为static，这种成员称为静态数据成员。
- 与一般的数据成员不同，无论建立多少个类的对象，都只有一个静态数据成员的拷贝。从而实现了同一个类的不同对象之间的数据共享。
- 定义静态数据成员的格式如下:

  static 数据类型 数据成员名;

静态成员的初始化

- 静态数据成员初始化应在类外单独进行，而且应在定义对象之前进行。
- 一般在cpp文件中，主函数main 之前,类声明之后为它提供定义和初始化。

```
class CScore {
  static int m_nCount;
. . .
};
int CScore::m_nCount=0;
int main( )
{. . .
}
```

给静态数据成员m_nCount赋初值

静态成员的初始化

数据类型 类名::静态数据成员名 = 初始值;

例如： int CScore::m_nCount=0;

如果未对静态数据成员赋初值，则编译系统会自动赋初值为0。

例如： int CScore::m_nCount;

等价于

int CScore::m_nCount=0;

静态成员的访问

(1) 静态数据成员属于类(准确地说，是属于类的一个对象集合)，而不像普通数据成员那样属于某一对象，因此可以使用“类名::”访问静态的数据成员。

(2) 用类名访问静态数据成员的格式如下:

类名::静态数据成员名

例如上面例子中：

CScore::m_nCount

静态成员的访问

(3) 公有静态数据成员可以在对象定义之前被访问。

一般格式:　　类名::静态数据成员名

例如：CScore::m_nCount=50;

(4) 对象定义后,公有静态数据成员,也可以通过对象进行访问。

一般格式:　　对象. 静态数据成员名

对象指针->静态数据成员名

例如：　cout<<sc1. m_nCount;

静态成员的访问

(5) 私有静态数据成员不能被外界直接访问，必须通过公有的成员函数间接访问。

(6) C++支持静态数据成员的一个主要原因是可以不必使用全局变量。

```
#include <iostream>
using namespace std;
class myclass{
      public:
            static int i;
};
int myclass::i=0;

int main()
{
      myclass::i=200;
      myclass ob1,*p;
      p=&ob1;
      cout<<"ob1.i:      "<<ob1.i<<endl;
      cout<<"myclass::i: "<<myclass::i<<endl;
      cout<<"p->i:       "<<p->i<<endl;
      return 0;
}
```

静态数据成员初始化,不必在前面加static

公有静态数据成员可以在对象定义之前被访问

公有静态数据成员可通过对象进行访问

程序运行结果如下:

```
ob1.i:      200
myclass::i: 200
p->i:       200
```

2.2 静态成员函数

- 在类定义中，有static说明的成员函数称为静态成员函数。
- 静态成员函数属于整个类，是该类所有对象共享的成员函数，而不属于类中的某个对象。
- 定义静态成员函数的格式如下:

```
static 返回类型 静态成员函数名(参数表）
```

2.2 静态成员函数

- 静态成员函数的主要作用是访问静态数据成员。
- 通过静态成员函数，可以在对象创建之前进行静态数据成员的访问。

公有静态成员函数的调用

调用公有静态成员函数的一般格式有如下几种:

类名::静态成员函数名(实参表)

对象.静态成员函数名(实参表)

对象指针->静态成员函数名(实参表)

需要说明的是：在静态成员函数中只能访问的静态数据成员。

为什么？

```
#include<iostream>
using namespace std;
class CCat{
   double weight;                    //普通数据成员，表示一只小猫的重量
   static double total_weight;       //静态数据成员,用来累计小猫的重量
   static double total_number;       //静态数据成员,用来累计小猫的只数
public:
    CCat(double w)      //构造函数
    { weight=w;
      total_weight+ = w;      //累加小猫的重量
      total_number++; }       //累加小猫的只数
    void display()
    { cout<<“ 这只小猫的重量是 ”  <<weight<<“千克\n";  }
    static void total_disp()
    { cout<<total_number <<" 只的小猫的总重量是 ";
      cout<<total_weight <<“ 千克"<<endl; }
};
```

普通的成员函数显示每只小猫的重量

静态成员函数显示小猫的只数和总重量

```
double CCat::total_weight=0.0;  //静态数据成员初始化
double CCat::total_number=0.0; //静态数据成员初始化
 int main()
 {
     CCat c1(0.5),c2(0.6),c3(0.4);
     c1.display();   //调用普通成员函数,显示第1只小猫的重量
     c2.display();   //调用普通成员函数,显示第2只小猫的重量
     c3.display();   //调用普通成员函数,显示第3只小猫的重量
     CCat::total_disp();       //调用静态成员函数
                                    //显示小猫的只数和总重量
     return 0;
}
```

程序的运行结果如下：

```
这只小猫的重量是 0.5千
这只小猫的重量是 0.6千克
这只小猫的重量是 0.4千克
3只的小猫的总重量是 1.5千克
```

说明

(1) 静态函数成员主要用来访问静态数据成员，当它与静态数据成员一起使用时，达到了对同一个类中对象之间共享数据的目的。

(2)静态成员函数一般为公有的，私有静态成员函数不能被类外部的函数和对象访问。

(3) 使用静态成员函数的一个原因是，可以在建立任何对象之前调用静态成员函数，以处理静态数据成员。

说明

(4) 静态成员函数是类的一部分 。如果要在类外调用公有的静态成员函数，使用如下格式：

类名::静态成员函数名()

说明

(5)若静态成员函数需要访问非静态成员，静态成员函数只能通过对象名(对象指针或引用)访问该对象的非静态成员。

如把display()函数定义为静态成员函数:

```
static void display(CCat& c)
{
    cout <<"这只小猫的重量是 "<<c.weight<< " 千克 \n " ;
}
```

静态成员总结

- 静态数据成员的定义：在定义或说明时加关键字static。
- 静态数据成员的初始化
  - 不能用构造函数初始化。
  - 初始化只能在类体外进行，前面不能加关键字“static”。
- 静态数据成员属于类，不属于对象。静态数据成员的存储空间不在任何一个对象的存储空间内，它是所有对象共有的。

单选题 2分

设置

此题未设置答案，请点击右侧设置按钮

关于静态数据成员**不正确的**说法是

A 静态数据成员虽然能实现同类对象共享数据，但却破坏了类的封装性

B 静态数据成员可以在main函数开始之前进行初始化

C 静态数据成员是所有同类对象共享的数据，它不能具体地属于哪一个对象

D 静态数据成员需在类的定义中利用static关键字进行声明

提交

3 友元

- 类的主要目的之一是实现信息的隐藏与封装，但这样往往会降低对私有数据成员的访问效率。
- 某些特殊的函数或类需要对私有成员进行直接访问，就需要打破类的封装。
- C++提供了友元机制，在最大限度地保证封装性的同时开放某些权限。
- 友元可分为友元函数和友元类。
- 声明友元的关键字是friend。

3.1 友元函数

```
#include <iostream>
using namespace std;
class CScore{
      private:
              int m_nMidtermExam;       //私有数据成员
              int m_nFinalExam;         //私有数据成员
      public:
             CScore(int m,int f);       //声明构造函数CScore()的原型
              void ShowScore ();
};
CScore::CScore(int m,int f)          //定义构造函数CScore()
{     m_nMidtermExam =m;
      m_nFinalExam =f;
}
```

非成员函数

```
int CalcScore (CScore &ob)
{
   return (int)(0.3*ob.m_nMidtermExam+0.7*ob.m_nFinalExam);
}
int main()
{
    CScore as[3]={CScore(80,88), CScore(90,92), CScore(70,80) };
    cout<<"三个期末成绩是："<<endl;
    for(int i=0;i<3;i++)
        cout<<CalcScore (as[i])<<endl;  //调用友元函数
    return 0;
}
```

3.1 友元函数

```
#include <iostream>
using namespace std;
class CScore{
      private:
              int m_nMidtermExam;       //私有数据成员
              int m_nFinalExam;         //私有数据成员
      public:
             CScore(int m,int f);       //声明构造函数CScore()的原型
              void ShowScore ();
             friend int CalcScore (CScore &);
};
CScore::CScore(int m,int f)         //定义构造函数Score()
{     m_nMidtermExam =m;
      m_nFinalExam =f;
}
```

3.1 友元函数

- 友元函数不是类的成员函数，但它可以访问该类所有的成员，包括私有成员、保护成员和公有成员。
- 在类中声明友元函数时，需在函数名前加上关键字friend。
- 友元函数既可以是非成员函数(全局函数)，也可以是另一个类的成员函数。

（1）全局函数声明为友元函数

```
friend int CalcScore (CScore &);

int CalcScore (CScore &ob)
{
    return (int)(0.3*ob.m_nMidtermExam+0.7*ob.m_nFinalExam);
}

cout<<CalcScore(as[i])<<endl;
```

说明

(1) 友元函数的声明可以放在公有部分，也可以放在保护部分和私有部分。

(2) 友元函数不是成员函数。因此，在类的外部定义友元函数时，不必像成员函数那样，在函数名前加上“类名::”。

说明

(3) 因为友元函数不是类的成员，所以它不能直接调用成员变量，它必须通过对象(对象指针或对象引用)作为入口参数，来调用该对象的成员。

(4) 一个函数可以是多个类的友元函数。当一个函数需要访问多个类时，友元函数非常有用。

(2) 成员函数作为友元函数

- 一个类的成员函数也可以作为另一个类的友元，这种成员函数称为友元成员函数。

- 友元成员函数不仅可以访问自己所在类对象中的私有成员和公有成员，还可以访问friend声明语句所在类中的所有成员，这样能使两个类相互合作、协调工作，完成某一任务。

```
#include<iostream.h>
#include<string.h>
（1）class CScore;          //对类CScore的提前引用声明
（2）class CStudent {
   void ShowStudentScore (CScore& sc);
};
（3）class CScore {
   friend void CStudent::ShowStudentScore(CScore& sc);
};
```

特别注意书写顺序：

1、被访问私有数据的类（CScore）声明；

2、作为朋友的类（CStudent）定义，但成员函数（ShowStudentScore）只能给出声明，不能完整定义；

3、被访问类（CScore）的定义，包括友元的声明

4、友元函数（ShowStudentScore）的定义

（4）void CStudent::ShowStudentScore(CScore& sc)

```
{
   cout<<"姓名:"<<name<<endl<<"学号:"<<number<<endl;
   cout<<"期中成绩:"<<sc.m_nMidtermExam<<endl<<"期末成绩:"<<sc.m_nFinalExam<<endl;
}
```

作为CStudent类的成员函数，访问CStudent类对象中的私有数据

作为CScore类的友元函数，访问CScore类对象中的私有数据

```
int main()
{
    CScore score1(90,92);
    CStudent stu1("DianaLin",80204);
    stu1.ShowStudentScore (score1);
    return 0;
}
```

调用CStudent类对象stu1的成员函数和CScore类的友元函数ShowStudentScore，实参是CScore类对象score1。

说明

- 类CStudent的成员函数作为类CScore的友元函数，必须先定义类CStudent。并且在声明友元函数时，要加上成员函数所在类的类名，如：

```
friend void CStudent::ShowStudentScore (CScore& sc);
```

声明友元函数时，要加上成员函数所在类的“类名::”

3.3 友元类

除了函数可以作为另一个类的友元，一个类也可以作为另一个类的友元，称为友元类。

友元类的声明方法是在另一个类声明中加入语句:

friend class 类名;

此语句可以放在公有部分也可以放在私有部分。

3.3 友元类

例如：

```
class  Y {
  …
};
class X {
  …
  friend  class Y;
  …
};
```

类Y中的所有成员函数都成为类X的友元函数
类Y中的所有成员函数都可以访问类X中的公有和私有成员

```
#include<iostream.h>    //例
#include<string.h>
class CScore;                  //对类CScore的提前引用声明
class CStudent{              //声明类CStudent

 void  ShowStudentScore(Score& sc);

};
```

定义函数ShowStudentScore为类CStudent的成员函数

声明类CScore是类CStudent的友元类

```
class CScore{
    friend class CStudent;
 public:
    …
};
void CStudent::ShowStudentScore(CScore& sc)
{
      cout<<"姓名:"<<name<<endl<<"学号:"
            <<number<<endl;
      cout<<"期中成绩:"<<sc.m_nMidtermExam<<endl
            <<"期末成绩:"<<sc.m_nFinalExam<<endl;
}
```

```
int main()
{
    CScore score1(90,92);
    CStudent stu1("DianaLin",80204);
    stu1.ShowStudentScore(score1);
    return 0;
}
```

友元类的声明可以不需要前向声明，因为友元语句已经指出类的存在。

3.3 友元类

友元类也可以是双向的

```
class  Y {
  …
 friend  class X;
};
class X {
  …
  friend  class Y;
  …
};
```

此时，Y中的成员函数只能先给出声明，然后在X的定义之后给出完整部分。

说明

(1) 友元关系是单向的，不具有交换性。

若类X是类Y的友元，类Y是否是X的友元，要看在类中是否有相应的声明。

(2) 友元关系也不具有传递性。

若类X是类Y的友元，类Y是类Z的友元，不一定类X是类Z的友元。

在C++中为什么要引入友元机制呢？

(1)声明了一个类的友元函数,就可以用这个函数直接访问该类的私有数据,从而提高了程序运行的效率。

(2)友元提供了不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。尤其当一个函数需要访问多个类时，友元函数非常有用。

在C++中为什么要引入友元机制呢？

(3)引入友元机制的另一个原因是方便编程，在某些情况下，如运算符被重载时，需要用到友元函数。

友元的利弊

- 友元的作用主要是为了提高效率和方便编辑。但是随着硬件性能的提高，友元的这点作用是微不足道的。

- 相反，友元破坏了类的整体操作性，也破坏了类的封装，所以在使用它时，要权衡利弊。

单选题 2分

设置

此题未设置答案，请点击右侧设置按钮

关于友元函数的说法**正确的**是

A 友元函数可以访问public和protect的成员，但不能访问private的成员

B 使用友元函数的目的是提高访问的灵活性

C 友元函数也是成员函数

D 不属于任何类的普通函数不可以被声明为友元函数

提交

单选题 2分

设置

此题未设置答案，请点击右侧设置按钮

已知类A是类B的友元，类B是类C的友元，则

A 类A一定是类C的友元

B 类C一定是类A的友元

C 类C的成员函数可以访问类B的对象的任何成员

D 类A的成员函数可以访问类B的对象的任何成员

提交

4 类的组合

- 类中的数据成员可以是基本数据类型，也可以是类类型这样自定义的数据类型。

- 在一个类的数据成员中包含了另一个类的对象称为类的组合。这个被包含的对象称为对象成员或子对象。

例如：

```
class A{
  …
};
class B{
    A a;
  public:
    …
};
```

类A的对象a为类B的对象成员

问题：
类B中含有对象成员a后，如何完成对象成员a的初始化工作?
类B的构造函数如何定义？

例如有以下的类X：

```
class  X {
   类名1  对象成员名1;
   类名2  对象成员名2;
          …
   类名n  对象成员名n;
 };
```

类X的构造函数的定义形式为：

```
X::X(参数表0)：对象成员名1(参数表1)，对象成员名2
   (参数表2)，…，对象成员名n(参数表n)
{
  类X的构造函数体
}
```

```
class A {  //.. ..   };
class B {  //.. ..   };
class C {  //.. ..   };
class D {
   A  a;
   B  b;
   C  c;
  public:
   D(参数表0):a(参数表1),b(参数表2),c(参数表3)
   {    // …构造函数体 }
} ;
```

a,b,c为对象成员

类D的构造函数应缀上对象成员 a,b,c 的初始化表

参数表1... 参数表3的数据一般来自参数表0

```
class D {
    A  a;
    B  b;
    C  c;
  public:
    D(参数表0):a(参数表1),b(参数表2),c(参数表3)
    {    // …构造函数体 }
};
    . . .
  D  d1(. . .);
```

(1)定义类D的对象d1时要调用构造函数D::D( )。首先，按各对象成员在类定义中的顺序依次调用它们的构造函数，对这些对象初始化。最后，再执行D::D( )的函数体。

撤销类D的对象d1时,调用析构函数的调用顺序与调用构造函数的顺序相反。

```
#include <iostream>
using namespace std;
class Date{
    private:
            int year;  int month;    int day;
    public:
            Date(int y,int m,int d);
            void ShowDate();
};
Date::Date(int y,int m,int d)
{
    year=y;    month=m;    day=d;
}
void Date::ShowDate()
{
     cout<<year<<"."<<month<<"."<<day<<endl;
}
```

```
class Student{
    private:
        string name;
        Date birthday;
    public:
        Student(string name,int y, int m, int d);
        void Show();
};
Student::Student(string name,int y, int m, int d):birthday(y,m,d)
{
    this->name=name;
}
void Student::Show()                    //定义输出数据函数show()
{
  cout<<"姓名: "<<name<<endl;
  cout<<"出生日期:" ;birthday.ShowDate();
}
```

类Date的对象birthday是类Student的对象成员

定义构造函数Student()，缀上对象成员的初始化列表

定义类Student的对象stu1，调用stu1的构造函数，初始化对象stu1

```
int main()
{
   Student stu1("黎明",1988,7,2);
   stu1.Show();
   return 0;
}
```

调用stu1的Show()函数，显示stu1的数据