

# 2023

# 面向对象程序设计

## 第四讲：类与对象的其他特性

李际军 lijijun@cs.zju.edu.cn



# 学习目标



0

掌握类的静态成员（静态数据成员和静态成员函数）的定义和使用方法

1  
02

掌握友元函数、友元类的作用、定义和使用方法

03

了解类的作用域，理解对象的类型和生存期

04

掌握各种常量的特点、定义和使用方法

# 目录

---



**4.1**

类的静态成员

**4.2**

友元

**4.3**

类的作用域和对象的生存期

**4.4**

常量类型



# 4.1

## 类的静态成员



静态成员是为了解决同一个类的不同对象之间数据成员和成员函数的共享问题。

类的成员分为：

- ◆ 静态成员：类属性，存储在静态区
- ◆ 非静态成员：对象属性，存储在动态栈区



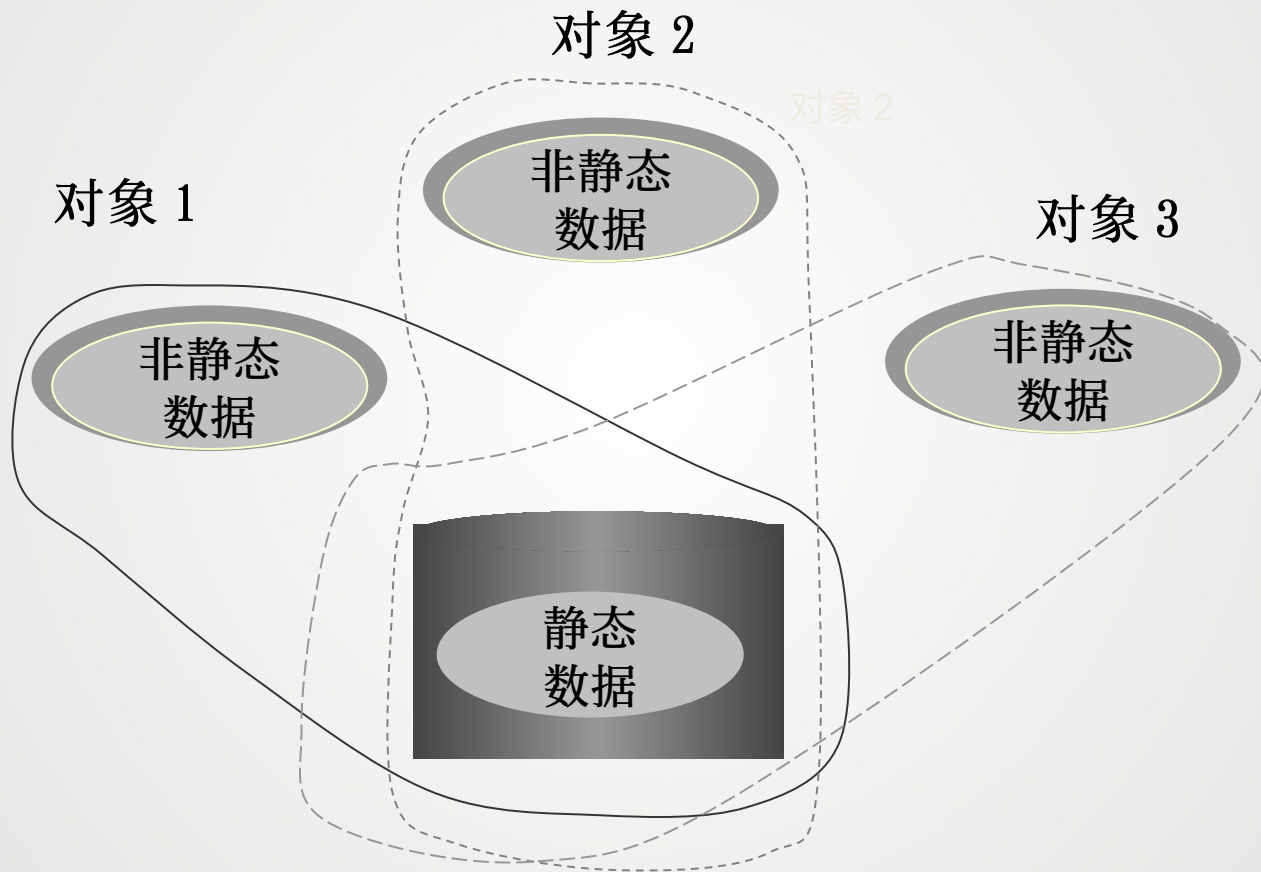
```
class Student{
private:
    string name;
    string class_id;
    int total_student_in-calss;
public:
    Student(...){...}
    .....
};
int main()
{
    Student st1(“ 张三”, “软件 2016”, 31 ) ;
    Student st2(“ 李四”, “软件 2016”, 32 ) ;
}
```

将对象共有属性用普通数据成员表示，每个对象都保存共有数据的一个副本，容易出现不一致问题



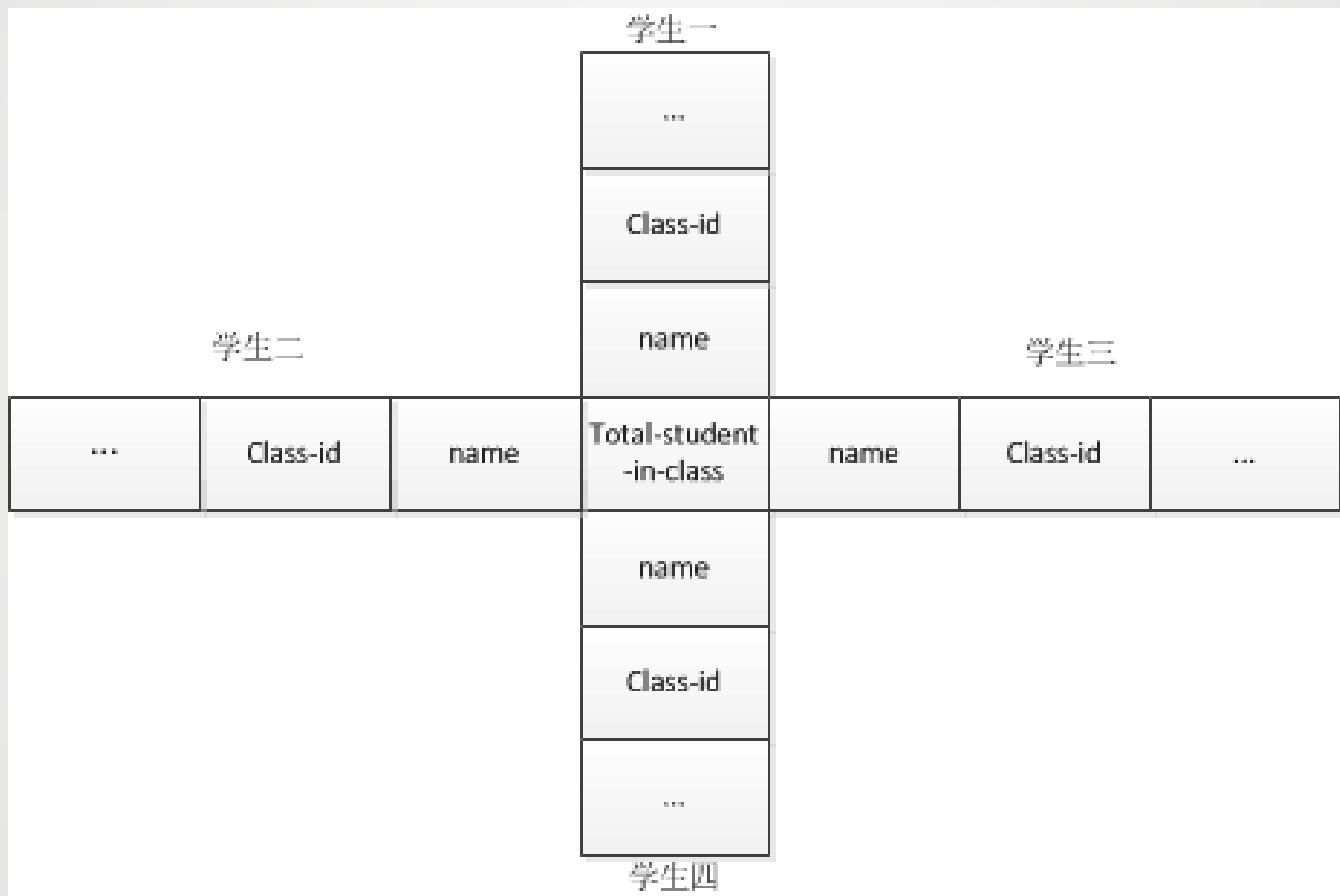
```
int total_student_in-calss=0;
class Student{
private:
    string name;
    string class_id;
public:
    Student(string name1 ,   string class_id1)
    {   name= name1 ;
        class_id= class_id1
        total_student_in-calss++
    }
    .....
};
```

类属性用全局变量描述，也会带来增加耦合度，降低信息隐藏和数据封装性，命名冲突等问题











有些情况下，可能希望有某一个或几个数据成员为同一个类的所有对象共有，也就是实现数据共享。若是采用类的普通数据成员的定义，这一目的是无法达到的。

这个问题可以通过定义一个或几个全局变量来解决，但如果在一个程序文件中有多个函数，那么在任何一个函数中都可以改变全局变量的值，这样全局变量的安全性就得不到保证，会破坏了类的封装性，也做不到信息隐藏。因此在实际程序编写中，很少使用全局变量。



C++ 通过静态数据成员来解决这个问题。

静态数据成员是类的所有对象共享的数据成员，而不是某个对象的数据成员。

使用静态数据成员的好处在于不但实现了数据共享，而且可以节省所使用的内存空间。系统给静态数据成员单独分配了一块存储区域，不论定义了多少个类的对象，静态数据成员的值对每个对象都是一样。



静态数据成员是一种特殊的数据成员类型，它的定义以关键字 `static` 开头。

静态数据成员定义的格式为：

`static`    数据类型    静态数据成员名；



【例 4-1】定义一个学生类 Student，其中包含的数据成员为：学生姓名，学号，成绩，以及学生总人数。程序代码如下：

```
class Student{  
Private:  
    char stu_name[10];  
    int stu_no;  
    float score;  
    static int total;    // 静态数据成员的定义  
public:  
    Student(char *name, int no, float sco);  
    void Print( );  
};
```



说明：

（1）静态数据成员和普通数据成员一样遵从  
`public`、`protected`、`private` 访问规则；

（2）静态数据成员属于本类的所有对象共享，不属于特定的类对象，在没有产生类对象时其作用域就可见，即在没有产生类的实例时，就可以操作它。

静态数据成员不能在类的构造函数中初始化。

静态数据成员也不可在类的体内进行赋初值，因为若在一个对象里给它赋初值。

静态数据成员的初始化工作只能在类外，并且在对象生成之前进行。



静态数据成员的初始化与一般数据成员初始化不同，其格式为：

**数据类型 类名 :: 静态数据成员 = 初始化值 ;**

说明：

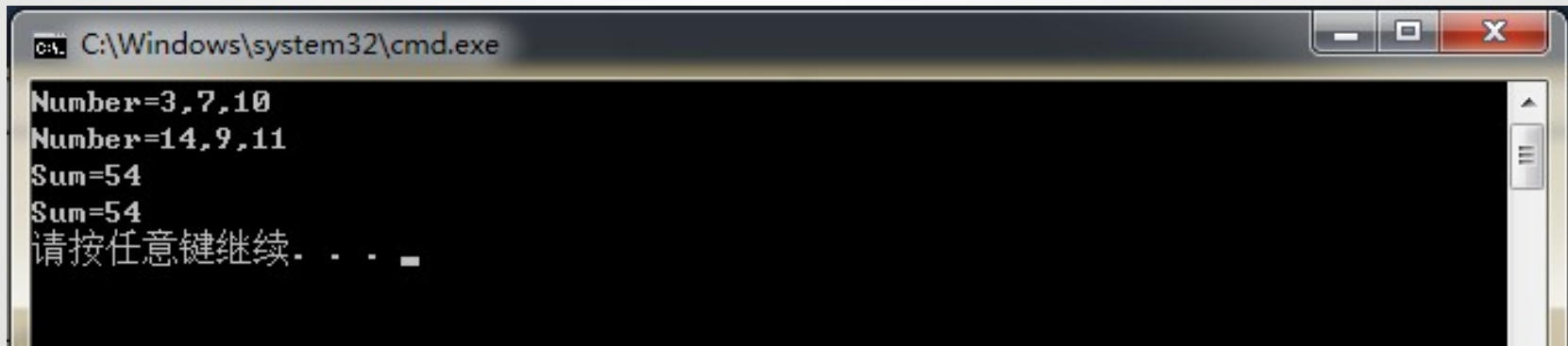
- (1) 静态数据成员初始化在类体外进行，而且前面不加 `static`，以免与一般静态变量或对象相混淆。
- (2) 初始化时不加该成员的访问权限控制符 `private`，`public` 等。
- (3) 初始化时使用作用域运算符来标明它所属类，因此，静态数据成员是类的成员，而不是对象的成员。

【例 4-2】类的静态数据成员初始化举例。

```
#include <stdafx.h>
#include<iostream>
#include<math.h>
using namespace std;
class Myclass
{
private:
    int A, B, C;
    static int Sum;
public:
    Myclass(int a, int b, int c);
    void GetNumber();
    void GetSum();
};
```

```
int MyClass::Sum = 0;           // 静态数据成员的初始化
MyClass::MyClass(int a, int b, int c)
{
    A = a;
    B = b;
    C = c;
    Sum += A+B+C;
}
void MyClass::GetNumber()
{
    cout<<"Number="<<A<<","<<B<<","<<C<<endl;
}
void MyClass::GetSum()
{
    cout<<"Sum="<<Sum<<endl;
}
```

```
int main()
{
    MyClass M(3, 7, 10), N(14, 9, 11);
    M.GetNumber();
    N.GetNumber();
    M.GetSum();
    N.GetSum();
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The window has standard Windows window controls (minimize, maximize, close). The command prompt displays the following output:

```
Number=3,7,10
Number=14,9,11
Sum=54
Sum=54
请按任意键继续. . .
```

静态数据成员在类外需要通过类名对它进行访问。静态数据成员的访问形式为：

**类名 :: 静态数据成员 ; (公有)**

也可以通过对象名访问，对象名访问形式为：

**对象名 . 静态数据成员 ; (公有)**

【例 4-3】类的静态数据成员使用举例。

```
#include <stdafx.h>
```

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
class Date
```

```
{
```

```
private:
```

```
    int month;
```

```
    int day;
```

```
    int year;
```

public:

**static int n;**

Date(int m,int d,int y)

*// 带参数的构造函数*

{

month=m;

day=d;

year=y;

**n++;**

}

Date(const Date& d)

*// 拷贝构造函数*

{

month=d.month;

day=d.day;

year=d.year;

**n++;**

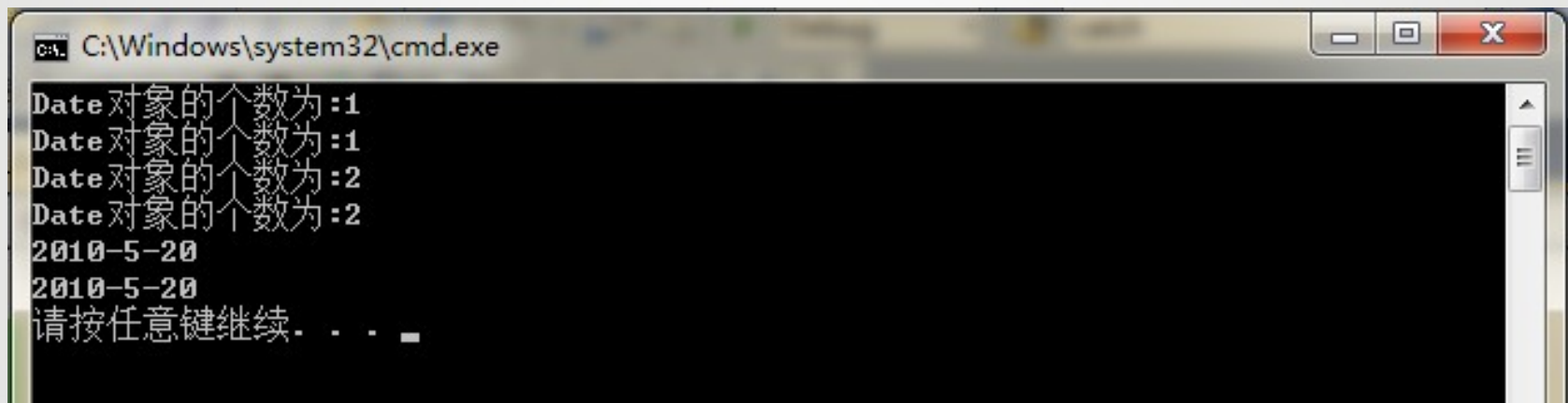
}



```
    ~Date( )                // 析构函数
    {
        n--;
    }
    void display( )
    {
        cout<<year<<"-"<<month<<"-"<<day<<endl;
    }
};
int Date::n=0;
```



```
int main( )
{
    Date date1(5, 20, 2010);
    cout<<"Date 对象的个数为 : "<< Date::n <<endl;
    cout<<"Date 对象的个数为 : "<<date1.n <<endl;
    Date date2=date1;
    cout<<"Date 对象的个数为 : "<< Date::n <<endl;
    cout<<"Date 对象的个数为 : "<<date2.n <<endl;
    date1.display( );
    date2.display( );
    return 0;
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has standard Windows window controls (minimize, maximize, close) in the top right corner. The command prompt shows the following output:

```
Date对象的个数为:1
Date对象的个数为:1
Date对象的个数为:2
Date对象的个数为:2
2010-5-20
2010-5-20
请按任意键继续. . .
```

1

静态数据成员主要用于：

- (1) 保存对象的个数。在构造函数中对该静态成员加 1，在析构函数里对该静态成员减 1。比如某个类的所有类对象共享一块动态分配的内存。
- (2) 表示对象共有的数据，如最大值、最小值等
- (3) 作为一个标记，标记一些动作是否发生，比如：文件的打开状态，打印机的使用状态，等等。
- (4) 存储链表的第一个或者最后一个成员的内存地址。链表的表头等



静态成员函数的声明格式为：

**static 返回类型 静态成员函数名 ( 参数表 );**

同普通成员函数一样，静态成员函数可以在类内定义，也可以在类外定义。在类外定义时，和普通成员函数的定义格式相同，而不要使用 `static` 前缀。

定义静态成员函数的格式：

**函数类型 类名 :: 静态成员函数名 ( 参数表 )**



静态成员函数是类的一部分，而不是对象的一部分。如果要在类外调用公用的静态成员函数，要使用类名和域运算符” :: ”，其格式为：

**类名：： 静态成员函数名（实参表）；**

也允许通过对象名来调用静态成员函数，格式为：

**对象名．静态成员函数名（实参表）；**



1

静态成员函数不属于某一对象，它与任何对象都无关，因此它没有 `this` 指针，不能访问类的默认非静态成员（包括非静态数据成员和非静态成员函数），只能访问本类中的静态成员（包括静态数据成员和静态成员函数）。

静态成员函数为操作静态成员而设置。

2



【例 4-4】静态成员函数访问本类非静态成员应用举例。

```
class Point
{
public:
    Point(int a, int b)
    {
        x=a;
        y=b;
    }
    static void f1(Point m);
private:
    int x;
    static int y;
};
```



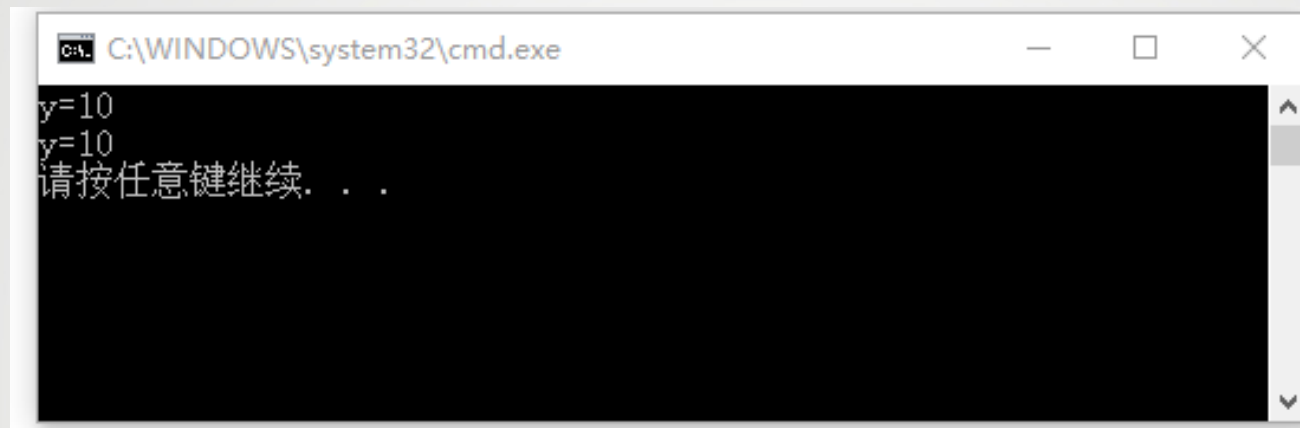
```
void Point::f1(Point m)
{
    cout<<"y="<<y<<endl;
}
int Point::y=0;           // 静态数据成员初始化
void main()
{
    Point P1(5,5),p2(10,10);
    Point::f1(P1); // 静态成员函数调用时不用对象名
    Point::f1(p2);
}
```



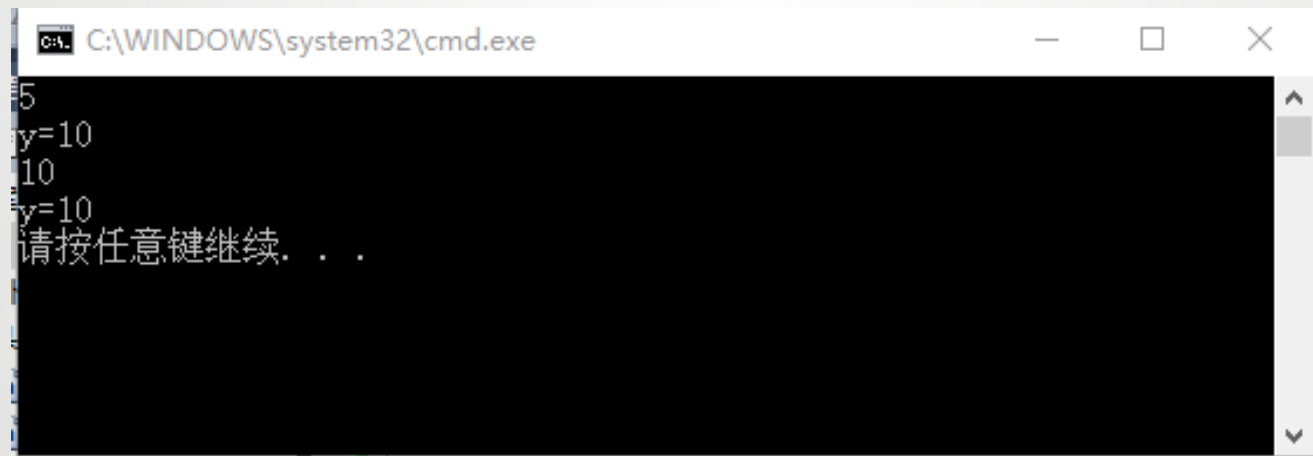


## 4.1.5 静态成员函数

33



```
C:\WINDOWS\system32\cmd.exe
y=10
y=10
请按任意键继续. . .
```



```
C:\WINDOWS\system32\cmd.exe
5
y=10
10
y=10
请按任意键继续. . .
```



### 【例 4-5】静态成员函数应用举例。

```
class Student                                // 定义 Student 类
{
    int num;
    int age;
    float score;
    static float sum;                        // 静态数据成员
    static int count;                       // 静态数据成员
public:
    Student(int n, int a, float s):num(n), age(a), score(s)
{ } // 定义构造函数
    void total( );
    static float average( );               // 声明静态成员函数
};
```



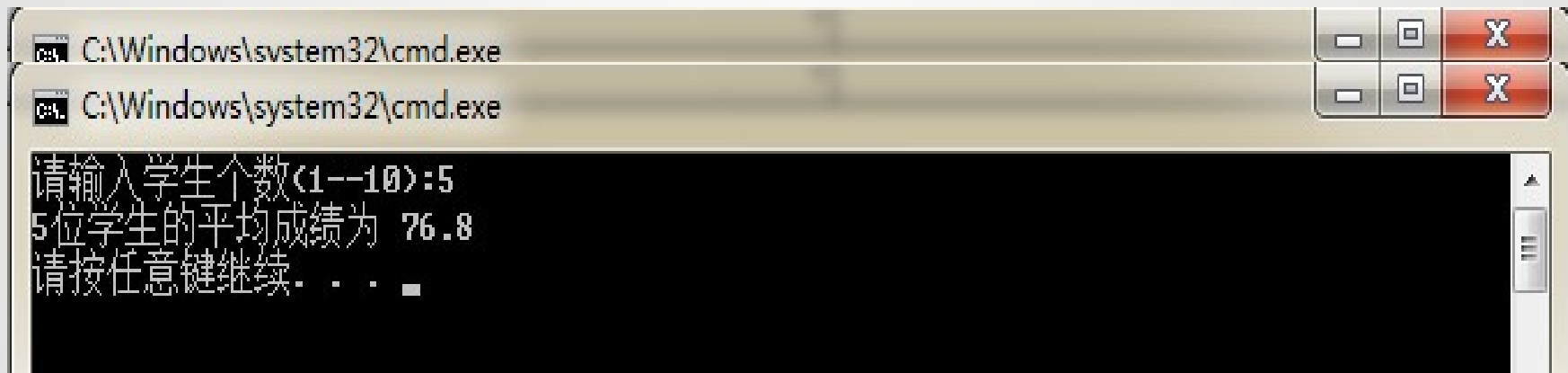
```
void Student::total()    // 定义非静态成员函数
{
    sum+=score;          // 计算总分
    count++;             // 累统计总人数
}

float Student::average( ) // 定义静态成员函数
{
    return(sum/count);
}
```

```
float Student::sum=0;    // 对静态数据成员初始化
int Student::count=0;    // 对静态数据成员初始化
```



```
int main( )
{
    Student stu[10]={           // 定义对象数组并初始化
        Student(10010,18,93),
        Student(10020,19,68),
        Student(10030,19,79),
        Student(10040,19,82),
        Student(10050,17,62),
        Student(10060,19,86),
        Student(10070,20,72),
        Student(10080,19,87),
        Student(10090,19,65),
        Student(10100,20,98)
    };
    int n;
    cout<<" 请输入学生个数 (1--10):";
    cin>>n;           // 输入需要求前面多少名学生的平均成绩
    for(int i=0;i<n;i++)
        stu[i].total( );
    cout<< n<<" 位学生的平均成绩为 " << Student::average( ) << endl;
    // 调用静态成员函数
    return 0;
}
```



静态成员函数间接访问非静态数据成员实例。

```
#include<iostream>
using namespace std;
class Myclass
{
public:
    Myclass(int =10);
    static int Getn(Myclass a);    // 静态成员函数
private:
    int m;                        // 非静态数据成员
    static int n;                  // 静态数据成员
};
Myclass:: Myclass(int mm):m(mm)
{ }
```



```
int Myclass::Getn(Myclass a)
{
    cout<<a.m<<endl;    // 通过对象间接访问非静态数据成员
    return n;
}
int Myclass::n=100;
int main()
{
    Myclass app1;
    cout<<Myclass::Getn(app1)<<endl; // 通过参数传递对象名
    return 0;
}
```

运行结果：

10

100

■ 在使用静态成员时，需要注意：

- （1）静态成员受访问权限的控制。
- （2）静态成员的访问方式有两种：通过类名或通过对象名。
- （3）静态数据成员的初始化必须在类外，不能通过构造函数进行初始化。
- （4）静态成员函数的作用就是为了访问私有的静态数据成员，尽量不去访问非静态数据成员。
- （5）静态成员函数没有 this 指针，因为它不属于任何一个对象。



# 4.2

## 友元



将数据与处理数据的函数封装在一起，构成类，既实现了数据的共享又实现数据的隐藏，无疑是面向对象程序设计的一大优点，但是封装并不是绝对的。

**静态成员定义**提供了同类不同对象数据的共享，属于**累内数据共享**。

C++ 为了进一步提高数据共享，通过**友元机制**实现**类外数据共享**

- 类具有封装和信息隐蔽的特性，只有类的成员函数才能访问类的私有成员，其他函数无权访问。为提高运行效率，有时确实需要非成员函数能够访问类的私有成员。
- 解决访问类的私有成员的方法：
  - （1）定义公有的访问私有数据成员的成员函数，通过类外调用公有成员函数达到目的；
  - （2）使用友元机制。
- 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，但破坏了类的封装性和数据的隐蔽性。

友元不是该类的成员函数，但是可以访问该类的私有成员。

友元的作用在于提高程序的运行效率，但是，它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。

对于一个类而言，它的友元是一种定义在该类外部的或者普通函数或者另一个类的成员函数或者另一个类，但需要在该类体内进行说明。

当友元是一个函数时，称该函数为友元函数；当友元是一个类时，称该类为友元类。



友元函数不是当前类中的成员函数，它既可以是一个不属于任何类的一般函数，也可以是另外一个类的成员函数。

将一个函数声明为一个类的友元函数后，它不但可以通过对象名访问类的公有成员，而且可以**通过对象名访问类的私有成员和保护成员**。

## 1. 非成员函数（普通函数）作为友元函数

声明非成员函数作为友元函数的语句格式为：

**friend 返回值类型 函数名（参数表）；**

■ 说明：

- （1）友元函数为非成员函数，一般在类中进行声明，在类外进行定义；
- （2）友元函数的声明可以放在类声明中的任何位置，即不受访问权限的控制；
- （3）友元函数可以通过对象名访问类的所有成员，包括私有成员。

## 【例 4-6】非成员函数作为友元函数应用举例。

```
class Date
{
    int month;
    int day;
    int year;
public:
    Date(int y,int m,int d);
    Date(Date &d);
    void display();
    friend void modifyDate(Date& date,int year,int month,int day);    // 声明类 Date 的友元函数
};
```

```
void modifyDate(Date& date,int year,int month,int day)
```

```
// 友元函数定义
```

```
{
```

```
    date.year=year;
```

```
    date.month=month;
```

```
    date.day=day;
```

```
}
```





```
Date::Date(int y,int m,int d)
{
    month=m;
    day=d;
    year=y;
}
Date::Date(Date &d)
{
    year=d.year;
    month=d.month;
    day=d.day;
}
void Date::display()
{
    cout<<year<<"-"<<month<<"-"<<day<<endl;
}
```

```
int main( )
{
    Date date1(2012,12,21);
    Date date2=date1;
    date1.display( );
    date2.display( );
    modifyDate(date1,2010,12,21);
    modifyDate(date2,2011,12,21);
    date1.display( );
    date2.display( );
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
2012-12-21
2012-12-21
2010-12-21
2011-12-21
请按任意键继续. . .
```

**【例 4-7】** 定义一个点类 Point，并求出两点间的距离。

```
#include<iostream>
#include<cmath>
using namespace std;
class Point
{
public:
    Point(int =0,int =0);
    ~Point(){}
    void Show();
    friend double Distance(Point p1,Point p2); // 声明为友元
private:
    int x,y;
};
```

```
Point::Point(int x1,int y1):x(x1),y(y1){ }
void Point::Show()
{   cout<<"( "<<x<<" , "<<y<<" )"<<endl; }
double Distance(Point p1,Point p2) // 求距离
{
    return sqrt((p1.x-p2.x)*(p1.x-p2.x)+
                (p1.y-p2.y)*(p1.y-p2.y));
}
int main()
{   Point p1(3,4),p2;
    p1.Show();   p2.Show();
    cout<<"Distance:"<<Distance(p1,p2)<<endl;
    return 0;
}
```

运行结果：

( 3 , 4 )

( 0 , 0 )

Distance:5

## 2. 类的成员函数作为友元函数

一个类的成员函数作为另一个类的友元函数的语句格式为：

**friend 返回值类型 类名 :: 函数名（参数表）；**

- 如果友元函数是一个类的成员函数，则在定义友元函数时要加上其所在类的类名。
- 访问友元函数时，在友元函数的前面加上自己的对象名即可。
- 如果同一函数需要访问不同类的对象，那么最适用的方法是使它成为这些不同类的友元，关键字 **friend** 在函数定义中不能重复。



**【例 4-8】** 类的成员函数作为另一个类的友元函数应用举例。

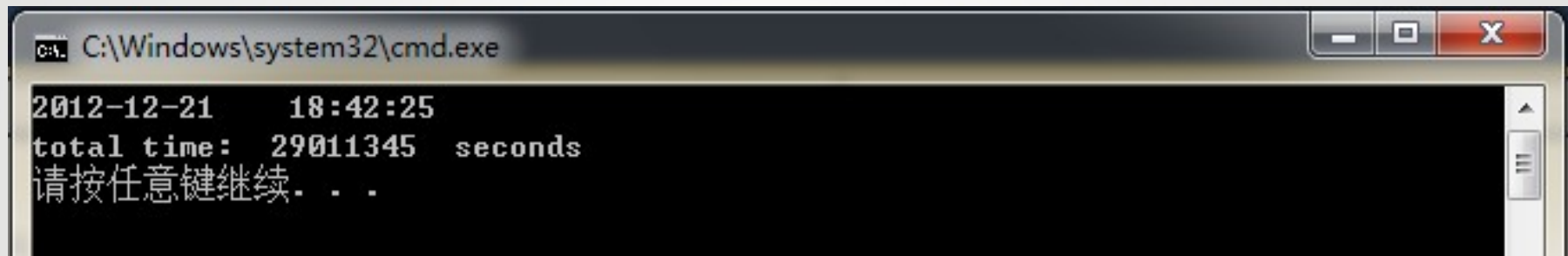
```
#include <stdafx.h>
#include <iostream>
using namespace std;
class Time;    // 前向引用声明
class Date
{
    int year;
    int month;
    int day;
public:
    Date(int y,int m,int d);
    void Calcutetime(Time t);
};
```



```
class Time
{
    int hour;
    int minute;
    int second;
public:
    Time(int h,int m,int s);
    friend void Date::Calcutetime(Time t); // 友元函数
};
int main()
{
    Date date(2012,12,21);
    Time time(18,42,25);
    date.Calcutetime(time);
    return 0;
}
```



```
Date::Date(int y,int m,int d):year(y),month(m),day(d){
Time::Time(int h,int m,int s):hour(h),minute(m),second(s){
void Date::Calcutetime (Time t)
{
int mon[12]={31,28,31,30,31,30,31,31,30,31,30,31};
int i,days=0,totaltime;
for(i=1;i<month;i++)
    days=days+mon[i-1];
if((year%4==0 && year %100!=0 ||year %400==0)&&month
>=3)
    days=days+1;
days+=day-1;
totaltime=((days*24+t.hour)*60+t.minute)*60+t.second ;
cout<<year <<'- '<<month <<'- '<<day <<"  ";
cout<<t.hour <<':'<<t.minute <<':'<<t.second <<endl;
cout<<"total time: " <<totaltime<<" seconds"<<endl;
}
```



```
C:\Windows\system32\cmd.exe  
2012-12-21    18:42:25  
total time: 29011345  seconds  
请按任意键继续. . .
```



【例 4-9】定义一个学生类 Student 和一个教师类 Teacher。在教师类中定义一个能修改学生成绩的成员函数。

分析如下：

- (1) 定义学生类 Student，并定义对象在主函数中进行测试。
- (2) 定义教师类 Teacher，并在主函数中定义对象进行测试。
- (3) 在教师类中添加修改学生成绩的成员函数，并进行测试。



（1）定义学生类 **Student**，并定义对象在主函数中进行测试。

```
#include<iostream>
#include<string>
#include<iomanip>
using namespace std;
class Student{
public:
    Student(string = "", string = "", double = 0);
    ~Student(){}
    void Show();
private:
    string num;    string name;    double score;
};
Student::Student(string n1, string n2, double s):num(n1),name(n2),score(s) { }
void Student::Show()
{    cout<<setw(8)<<"num"<<setw(8)<<"name"<<setw(8)<<"score"<<endl;
    cout<<setw(8)<<num<<setw(8)<<name<<setw(8)<<score<<endl;
}
int main()
{    Student stu("x001"," 王强 ",88);    stu.Show();    return 0; }
```



（2）定义教师类 Teacher，并在主函数中定义对象进行测试。

```
#include<iostream>
#include<string>
#include<iomanip>
using namespace std;
class Teacher{
public:
    Teacher(string="",string="");
    ~Teacher(){}
    void Show_Teacher();
private:
    string num;  string name;
};
Teacher::Teacher(string n1,string n2):num(n1),name(n2) { }
void Teacher::Show_Teacher()
{   cout<<setw(8)<<"num"<<setw(8)<<"name"<<endl;
    cout<<setw(8)<<num<<setw(8)<<name<<endl;
}
int main()
{   Teacher t("t001"," 杨桃 ");   t.Show_Teacher();   return 0;   }
```

(3) 在教师类中添加修改学生成绩的成员函数，并进行测试。

```
#include<iostream>
#include<string>
#include<iomanip>
using namespace std;
class Student; // 类的提前声明
class Teacher{
public:
    void SetScore(Student&,double); // 修改指定学生成绩
    .....
};
class Student .....
    friend void Teacher::SetScore(Student &stu,double s); // 声明为友元函数
};
//Teacher 类和 Student 类的成员函数的定义
..... // 相同部分省略
```



(3) 在教师类中添加修改学生成绩的成员函数，并进行测试。

```
void Teacher::SetScore(Student &stu,double s)// 修改指定学生成绩
```

```
{
```

```
    stu.score=s;
```

```
}
```

```
int main()
```

```
{
```

```
    Teacher t("t001"," 杨桃 ");
```

```
    Student stu("x001"," 王强 ",88);
```

```
    cout<<" 修改之前: "<<endl;
```

```
    stu.Show_Student();
```

```
    t.SetScore(stu,99);
```

```
    cout<<" 修改之后: "<<endl;
```

```
    stu.Show_Student();
```

```
    return 0;
```

```
}
```

运行结果：

修改之前：

num	name	score
x001	王强	88

修改之后：

num	name	score
x001	王强	99



关于友元函数的几点说明：

- （1）由于友元函数不是类的成员函数，所以对友元函数指定访问权限无效，因此可以把友元函数的说明放在 `private`，`public`，`protected` 的任意段中。
- （2）使用友元函数可以提高程序的执行效率。
- （3）友元函数要慎用，因为它可以在类外通过对象直接访问类的私有或保护成员，破坏了类的信息隐蔽性。



如果希望 A 类中的所有成员函数都能够访问 B 类中所有私有和保护成员，可以将 A 类中的每个成员函数声明为 B 类的友元函数，但这样做显得比较繁琐。为此，C++ 提供了友元类，也就是一个类可以声明为另一个类的友元类。

若 A 类声明为 B 类的友元类，那么，A 类中的每一个成员函数都可以访问 B 类中的任何类型的成员。

声明友元类的语句格式为：

```
friend class 类名 ;
```

### ■说明:

- (1) 友元类的声明同样可以在类声明中的任何位置;
- (2) 友元类的所有成员函数将都成为友元函数。

【例 4-10】友元类应用举例。

```
class DateFriend;           // 前向引用声明
class Date
{
private:
    int month;
    int day;
    int year;
public:
    Date(int m,int d,int y) ;
    friend class DateFriend; // 定义友元类
};
```

```
Date::Date(int m,int d,int y)
```

```
{  
    month=m;  
    day=d;  
    year=y;  
}
```

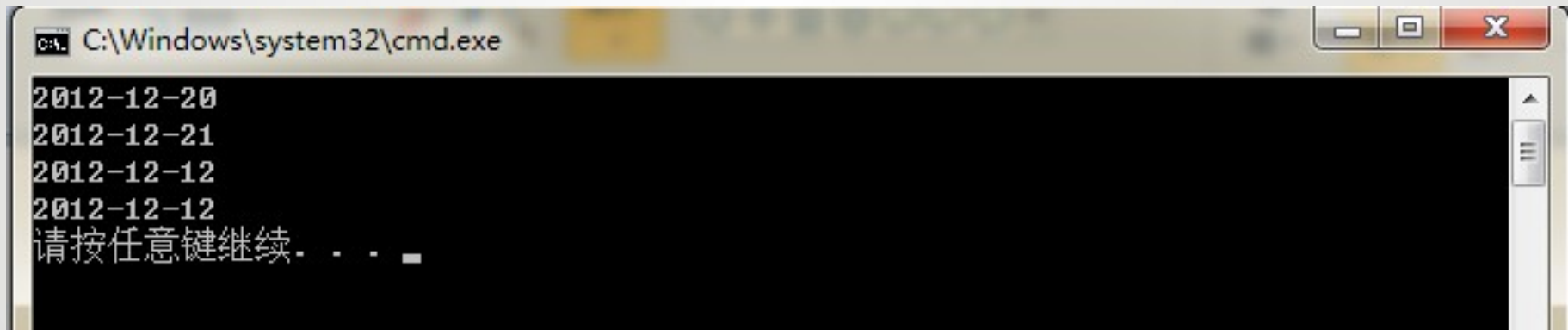
```
void DateFriend::modifyDate(Date& date,int month,int day,int  
year)
```

```
{  
    date.month=month;  
    date.day=day;  
    date.year=year;  
}
```

```
class DateFriend{
public:
    void modifyDate(Date& date,int month,int day,int year);
    void display(const Date& date);
};

int main( ){
    Date date1(12,20,2012);
    Date date2(12,21,2012);
    DateFriend dateFriend;
    dateFriend.display(date1);
    dateFriend.display(date2);
    dateFriend.modifyDate(date1,12,12,2012);
    dateFriend.modifyDate(date2,12,12,2012);
    dateFriend.display(date1);
    dateFriend.display(date2);
    return 0;
}
```

```
void DateFriend::display(const Date& date)
{
    cout<<date.year<<"-"<<date.month<<"-"<<date.day<<endl;
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The command prompt has a black background with white text. It displays four lines of dates: '2012-12-20', '2012-12-21', '2012-12-12', and '2012-12-12'. Below the dates, it shows the text '请按任意键继续. . . \_' (Press any key to continue. . . \_).



【例 4-11】将例 4-10 通过友元类实现。

```
#include<iostream>
#include<string>
#include<iomanip>
using namespace std;
class Student;           // 类的提前声明
class Teacher{
public:
    Teacher(string = "", string = "");
    ~Teacher(){}
    void Show_Teacher();
    void SetScore(Student &, double); // 修改指定学生成绩
private:
    string num;
    string name;
};
```



```
class Student{
public:
    Student(string="",string="",double=0);
    ~Student(){}
    void Show_Student();
    friend class Teacher;           // 声明类 Teacher 为友元类
private:
    string num;
    string name;
    double score;
};
//Teacher 类和 Student 类的成员函数的定义
..... // 相同部分省略
```



```
void Teacher::SetScore(Student &stu,double s) // 修改指定学生成绩
{
    stu.score=s;
}
int main(){
    Teacher t("t001"," 杨桃 ");
    Student stu("x001"," 王强 ",88);
    cout<<" 修改之前: "<<endl;
    stu.Show_Student();
    t.SetScore(stu,99);
    cout<<" 修改之后: "<<endl;
    stu.Show_Student();
    return 0;
}
```

运行结果:

修改之前:

num	name	score
x001	王强	88

修改之后:

num	name	score
x001	王强	99

■ 需要注意：

- （1）友元关系具有单向性。如果声明类 A 是类 B 的友元，则类 A 的所有成员函数都将变成友元函数，都可以访问类 B 的所有成员，但类 B 的成员函数却不能访问类 A 的私有和保护成员。
- （2）友元关系不具有传递性。如果类 A 是类 B 的友元，类 B 是类 C 的友元，类 C 和类 A 之间如果没有声明，就没有任何友元关系，不能进行数据共享。
- （3）友元的提出方便了程序的编写，但是却破坏了数据的封装和隐蔽，应该尽量减少友元的使用。

# 4.3



## 类的作用域和对象的生存期



作用域是指一个标识符的有效范围。

C++ 中标识符的作用域有函数作用域、块作用域、类作用域和文件作用域。



类的作用域是指在类的定义中由一对花括号所括起来的部分，包括数据成员和成员函数。

在类的作用域中，类中的成员函数可以不受限制的访问本类的成员（数据成员和成员函数）。

在类的作用域之外，类的成员通过对象的句柄引用，句柄可以是对象名、对象引用或对象指针。

在类的成员函数中定义的标识符有函数作用域。如果类的成员函数中定义了与类作用域内变量同名的另一个变量，那么在函数作用域内，函数作用域内的变量将隐藏类作用域内的变量。要在函数中访问这种被隐藏类作用域变量，就需要在其前面添加类名和作用域运算符（`::`）。

圆点成员选择运算符（`.`）与对象名或对象引用结合使用，即可访问对象成员。箭头成员选择运算符（`->`）与对象指针结合使用，也可访问对象成员。

### 【例 4-12】类的作用域应用举例。

```
class Count{
public:
    int x;
    void Calcute(int x){
        int y;
        y=x+2;    // 形参  $x$  与类的数据成员  $x$  同名，因此在函数中，类的数据成员被隐藏
        Count::x =y*2;    // 由于类的数据成员  $x$  被隐藏，要访问它需类作用符
    }
    void print() {
        cout<<x<<endl;
    }
};
```



```
int main ()
{
    Count count;           // 定义 Count 类的对象 count
    Count *count_Ptr = &count;
    // 定义 Count 类的指针 coun_Ptr, 并指向 count 对象
    Count &count_Ref = count;
    // 定义 Count 类的引用 coun_Ref, 它是 count 对象的别名
    cout<<" 使用对象名 "<<endl;
    count.x = 1;
    cout<<" 调用 Calcute 函数前 x=";
    count.print();
    count.Calcute (count.x);
    cout<<" 调用 Calcute 函数后 x=";
    count.print();
    cout<<" 使用引用 ";
    count_Ref.x = 2;
```





```
    cout<<" 调用 Calcute 函数前 x=";  
    count_Ref.print();  
    count.Calcute (count_Ref.x);  
    cout<<" 调用 Calcute 函数后 x=";  
    count.print();  
    cout<<" 使用指针 ";  
    count_Ptr->x = 3;  
    cout<<" 调用 Calcute 函数前 x=";  
    count_Ptr-> print();  
    count.Calcute (count_Ptr->x);  
    cout<<" 调用 Calcute 函数后 x=";  
    count.print();  
    return 0;  
}
```

A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The command prompt has a black background with white text. The text displayed is as follows:  
使用对象名  
调用Calcute函数前x=1  
调用Calcute函数后x=6  
使用引用调用Calcute函数前x=2  
调用Calcute函数后x=8  
使用指针调用Calcute函数前x=3  
调用Calcute函数后x=10  
请按任意键继续. . .  
The window includes standard Windows window controls (minimize, maximize, close) in the top right corner and a vertical scrollbar on the right side.



对象的生存期是指对象从被创建开始到被释放为止的时间。

按生存期的不同，对象可分为局部对象、静态对象、全局对象和动态对象四种。



### 1、局部对象

局部对象是指定义在一个程序块或函数体内的对象。当定义对象时，系统自动调用构造函数，该对象被创建，对象的生存期开始。当退出该对象所在的函数体或程序块时，调用析构函数，释放该对象，对象的生存期结束。

### 2、静态对象

静态对象是指以关键字 `static` 标识的对象。当定义对象时，系统自动调用构造函数，该对象被创建，对象的生存期开始。当程序结束时调用析构函数，该对象被释放，对象的生存期结束。因此，静态对象的生存期从定义该对象时开始，到整个程序结束时终止。

### 3、全局对象

全局对象是指定义在函数体外的对象。它的作用域从定义时开始到程序结束时终止。当程序开始时，该对象被创建。当程序结束时调用析构函数，该对象被释放。全局对象的生存期从程序开始运行时开始，到整个程序结束时终止。

### 4、动态对象

动态对象是指以运算符 `new` 创建，以运算符 `delete` 释放的对象。当程序执行运算符 `new` 时创建该动态对象，对象的生存期开始。当执行运算符 `delete` 时释放该动态对象，对象的生存期结束。

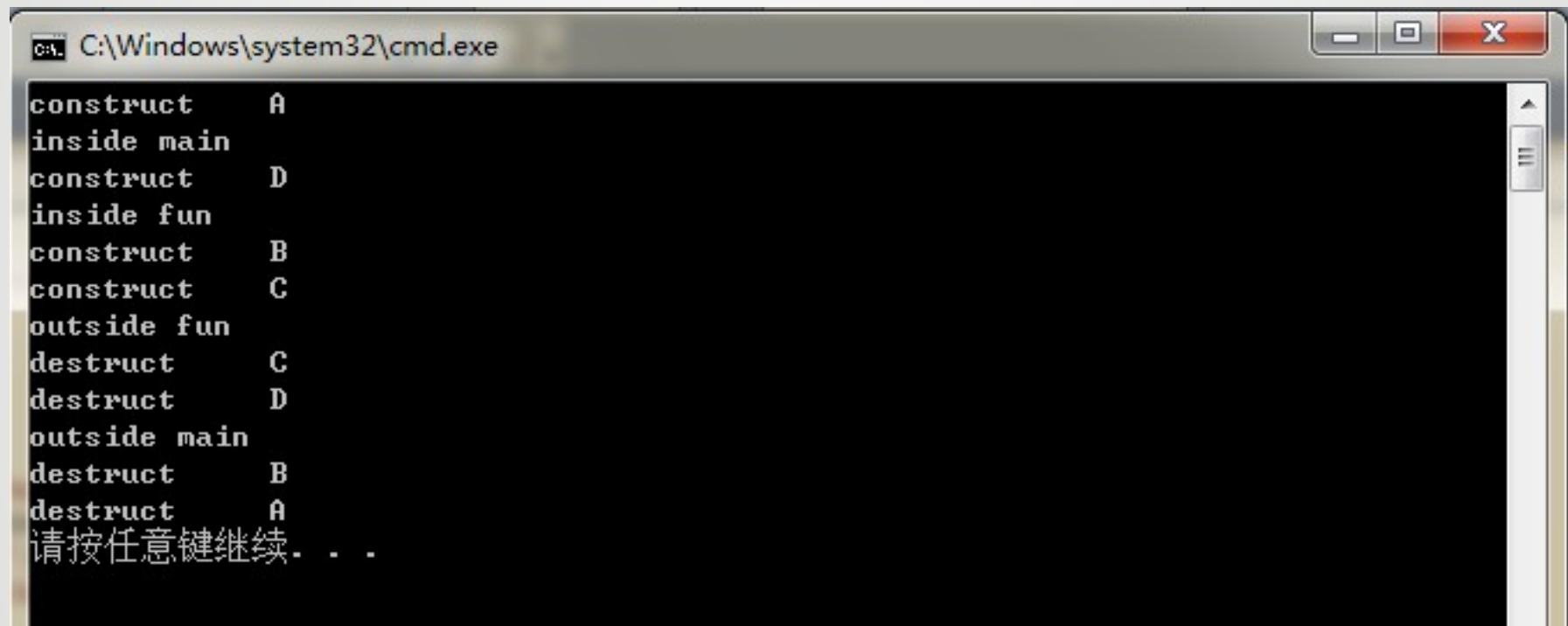


### 【例 4-13】对象生存期应用举例。

```
class ObjectLife
{
    char cha;
public:
    ObjectLife(char c)
    {
        cha=c;
        cout<<"construct" <<cha<<endl;
    }
    ~ObjectLife()
    {
        cout<<"destruct" <<cha<<endl;
    }
};
```



```
void fun()
{
    cout<<"inside fun"<<endl;
    static ObjectLife B('B'); // 定义静态对象
    ObjectLife C('C');        // 定义局部对象
    cout<<"outside fun"<<endl;
}
void main()
{
    cout<<"inside main"<<endl;
    ObjectLife *D= new ObjectLife('D'); // 定义动态对象
    fun();
    delete D;                          // 释放动态对象
    cout<<"outside main"<<endl;
}
```



```
C:\Windows\system32\cmd.exe
construct      A
inside main
construct      D
inside fun
construct      B
construct      C
outside fun
destruct       C
destruct       D
outside main
destruct       B
destruct       A
请按任意键继续. . .
```



# 4.4

## 常量类型



对于既需要共享，又需要防止值被改变的数据，应该声明其为常量。常量在程序运行过程中其值是不可改变的，因而可以有效保护数据。

- 常量的定义使用。定义或说明常量时必须对其进行初始化。类型修饰符 `const`.
- 常量包含简单数据类型常量、对象类型常量（常量对象）、引用类型常量（常量引用）、常量对象成员（包括常量成员函数和常量数据成员）、数组常量（常量数组）和指针常量（常量指针）等。
- 本节介绍常量对象、常量引用、常量对象成员、指向常量的指针和常量指针。



常量对象的特点是它的数据成员的值在对象的整个生存期内都不能被修改。在定义常对象时必须进行初始化。

常量对象的定义格式如下：

`<类名> const <对象名>;`

或者

`const <类名> <对象名>;`



类的常量成员包括常量成员函数和常量数据成员。

## 1. 常量成员函数

常量成员函数的定义要使用 `const` 关键字，其定义格式为：

<返回值类型> 函数名（参数表） `const`;

说明：

（1）`const` 是函数类型的一部分，在实现部分也要带该关键字。

（2）`const` 关键字可用于对重载函数的区分。

（3）常量成员函数不能更新类的数据成员的值，也不能调用该类中没有用 `const` 修饰的成员函数。只能调用常量成员函数。

**【例 4-14】** 常量成员函数、常量对象应用相应情况举例。

```
class Rectangle
```

```
{
```

```
    int w,h;
```

```
public:
```

```
    int getValue1() const;
```

```
    int getValue();
```

```
    void setValue(int a,int b);
```

```
    void setValue(int x,int y) const;
```

*//const 关键字可以用于对重载函数的区分*

```
    Rectangle(int x,int y);
```

```
    Rectangle(){}  
};
```

```
void main()
{
    Rectangle const a(3,4);    // 定义常量对象
    a.setValue(10,20);        // 常量对象可以调用常量成员函数
    Rectangle c(2,6);         // 定义普通对象
    c.setValue (10,20);
    cout<<a.getValue()<<endl; // 错误, 常量对象不能调用非常量成员函数
    cout<<a.getValue1 ()<<endl;
    cout<<c.getValue()<<endl;
    cout<<c.getValue1() <<endl;
}
```

```
int Rectangle::getValue1() const
{
    return w*h;
}
int Rectangle::getValue()
{
    return w+h;
}
void Rectangle::setValue(int a,int b)
{
    w=a;h=b;    // 可以更新数据成员
    getValue1(); // 正确, 非常量成员函数可以调用常量成员函数
}
```



```
void Rectangle::setValue(int a,int b) const
```

```
{
```

```
    w=a;h=b; / 错误, 常量成员函数不能更新任何数据成员
```

```
    getValue(); // 错误, 常量成员函数不能调用非常量成员函数
```

```
    getValue1(); // 正确, 常量成员函数可以调用常量成员函数
```

```
}
```

```
Rectangle::Rectangle(int x,int y)
```

```
{
```

```
    w=x;
```

```
    h=y;
```

```
}
```

错误列表	
<div> <div>✖ 4 个错误</div> <div>⚠ 0 个警告</div> <div>ℹ 0 个消息</div> </div>	
	说明
✖ 1	error C2662: "Rectangle::getValue" : 不能将 "this" 指针从 "const Rectangle" 转换为 "Rectangle &"
✖ 4	error C2662: "Rectangle::getValue" : 不能将 "this" 指针从 "const Rectangle" 转换为 "Rectangle &"
✖ 3	error C3490: 由于正在通过常量对象访问 "h" , 因此无法对其进行修改
✖ 2	error C3490: 由于正在通过常量对象访问 "w" , 因此无法对其进行修改

看出：

- （1）常量对象只能调用类的常量成员函数，不能调用类的非常量成员函数。
- （2）常量成员函数内，不能修改类的数据成员。
- （3）常量成员函数只能调用类的其它常量成员函数，不能调用类的非常量成员函数。
- （4）`const` 关键字可以用于对重载函数的区分。
- （5）非常量成员函数不但可以调用非常量成员函数，也可以调用常量成员函数。
- （6）`const` 是函数类型的一个组成部分，因此在函数的定义部分也要带 `const` 关键字。



**【例 4-15】对【例 4-14】修改后的结果。**

```
#include <stdafx.h>
#include <iostream>
using namespace std;

class Rectangle{
    int w,h;
public:
    int getValue() const;
    int getValue();
    Rectangle(int x,int y);
    Rectangle(){}
};
```



```
int main()
{
    Rectangle const a(3,4);
    Rectangle c(2,6);
    cout<<a.getValue()<<endl;
    cout<<c.getValue()<<endl;
}
int Rectangle::getValue() const
{
    return w*h;
}
```



```
int Rectangle::getValue()
{
    return w+h;
}
Rectangle::Rectangle(int x,int y)
{
    w=x;
    h=y;
}
```

## 2、常量数据成员

类的数据成员也可以是常量。使用 `const` 关键字说明的数据成员为常量数据成员。

若在一个类中定义了常量数据成员，那么任何函数都不能对该数据成员赋值。

构造函数对该数据成员进行初始化，只能通过初始化列表进行。



```
#include <iostream>    // 【例 4-16】
using namespace std;
class A
{
    const int a;        // 常量数据成员
    static const int b; // 静态常量数据成员




public:
    A();
    A(int i);
    void Output();
};



const int A::b = 20; // 静态常量数据成员在类外初始化
```



```
A::A():a(10)
{ //a=10;      // 错误, 常量数据成员不能在函数内赋值
}
A::A(int i):a(i) / 正确, 常量数据成员通过初始化列表初始化
{}
void A::Output ()
{ cout<<a<<":"<<b<<endl;
}
int main()
{
    A a1(10),a2;
    a1.Output ();
    a2.Output();
    return 0;
}
```

错误列表

 2 个错误 |  0 个警告 |  0 个消息

	说明
 2	error C2166: 左值指定 const 对象
 1	error C2758: "A::a" : 必须在构造函数基/成员初始值设定项列表中初始化



**【例 4-17】对【例 4-15】修改后的结果。**

```
#include <stdafx.h>
#include <iostream>
using namespace std;
class A
{
    const int a;
    static const int b;
public:
    A();
    A(int i);
    void Output();
};
const int A::b =20;
```



```
A::A ():a(15)
{
}
A::A(int i):a(i)
{
}
void A::Output ()
{
    cout<<a<<":"<<b<<endl;
}
int main()
{
    A a1(10),a2;
    a1.Output();
    a2.Output();
    return 0;
}
```



在声明引用时用 `const` 修饰，那么被声明的引用就是常量引用。常量引用所引用的对象不能被改变。若用常量引用作函数的形参，那么就不会意外地发生对实参的更改。

常量引用的声明格式如下：

`const` 类型说明符 & 引用名 ;

**【例 4-18】常量引用应用举例。**

```
void Output(const int &i)
```

```
{
```

```
    i++; // 错误, 常量引用作为形参, 其值不能被改变
```

```
    cout<<i<<endl;
```

```
}
```

```
int main()
```

```
{
```

```
    int i=12;
```

```
    Output(i);
```

```
    return 0;
```

```
}
```



## 错误列表



1 个错误



0 个警告



0 个消息

		说明
	1	error C3892: "i" : 不能给常量赋值



1

error C3892: "i" : 不能给常量赋值





`const` 与指针的配合使用有两种方式：

一种是用 `const` 修饰指针指向的变量，即修饰指针所指向的变量的内容，称为指向常量的指针；

另一种是用 `const` 修饰指针，即修饰存储在指针里的地址，称为常量指针。



### 1. 常量指针

常量指针的定义格式如下：

类型名 \* const 指针名 ;

例如：

```
int x=3;
```

```
int * const w=&x;
```

表明 w 为一个指向 int 类型的变量 x 的常量指针。它必须有一个初始值（地址），并且只能指向这个初始变量，不能“被改变”指向其它变量，但变量的值可以被改变。



### 针 说明:

- (1) 如果一个变量已被声明为常变量，只能用指向常变量的指针变量指向它，而不能用一般的（指向非 const 型变量的）指针变量去指向它。
- (2) 指向常变量的指针变量除了可以指向常变量外，还可以指向未被声明为 const 的变量。此时不能通过此指针变量改变该变量的值。如果希望在任何情况下都不能改变 c1 的值，则应把它定义为 const 型。
- (3) 如果函数的形参是指向非 const 型变量的指针，实参只能用指向非 const 变量的指针，而不能用指向 const 变量的指针，这样，在执行函数的过程中可以改变形参指针变量所指向的变量（也就是实参指针所指向的变量）的值。



- (1) 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它，而不能用一般的（指向非 const 型对象的）指针变量去指向它。
- (2) 如果定义了一个指向常对象的指针变量，并使它指向一个非 const 的对象，则其指向的对象是不能通过指针来改变的。如果希望在任何情况下 t1 的值都不能改变，则应把它定义为 const 型。
- (3) 指向常对象的指针最常用于函数的形参，目的是在保护形参指针所指向的对象，使它在函数执行过程中不被修改。
- (4) 如果定义了一个指向常对象的指针变量，是不能通过它改变所指向的对象的值的，但是指针变量本身的值是可以改变的。



例如:

```
double y=4.3;
```

```
double * const m=&y;
```

```
double z=3.4;
```

```
m=&z;      // 错误, 不能改变常量指针指向的变量
```

```
*m=3.4;    // 正确, 可以改变常量指针指向变量的值
```



## 2. 指向常量的指针

指向常量的指针的定义格式如下：

**const 类型名 \* 指针名 ;**

例如：

```
const int *w;
```

表明 `w` 为一个指向 `const int` 类型的指针，它指向一个整型常量，这个常量的值不能被改变，但 `w` 指向的变量可以被改变，即指针所指向的地址可以被改变。



例如：

```
const double *m;
```

```
double y=4.3;
```

```
m=&y;
```

```
*m=3.4;
```

*// 错误，不能改变指向常量的指针指向变量的内容*

```
double z=3.4;
```

```
m=&z; // 正确，可以改变指向常量的指针指向的变量
```



### 【例 4-19】常量指针作函数参数应用举例。

```
class aa
{public:
    int s[6];
    aa(){};
    aa(aa &p)    // 拷贝构造函数
    { cout<<"copy construct..."<<endl;    }
    void input(const int *p,int n);
};

void aa::input(const int *p,int n)
{   for(int i=0;i<n;i++)
    s[i]=*(p+i);}

void print(const aa *sa)
{   for (int i=0; i<6 ;i++)
    cout<<(*sa).s[i]<<endl; }
```





#### 4.4.4 常量指针与指向常量的指针

121

```
Int main()
{
    int array[6];
    aa wa;
    cout<<" 请输入数组元素的内容: "<<endl;
    for (int i=0; i<6;i++)
        cin>>array[i];
    wa.input (array,6);
    cout<<" 请输出数组元素的内容: "<<endl;
    print (&wa);
}
```



#### 4.4.4 常量指针与指向常量的指针

122

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The text shows a sequence of inputs and outputs for an array. First, it prompts "请输入数组元素的内容:" (Please enter the content of the array elements:). Then, it lists the inputs: 23, 24, 12, 45, 34, 67. Next, it prompts "请输出数组元素的内容:" (Please output the content of the array elements:). Finally, it lists the outputs: 23, 24, 12, 45, 34, 67. At the bottom, it says "请按任意键继续. . . ." (Press any key to continue. . . .).

```
C:\Windows\system32\cmd.exe
请输入数组元素的内容:
23
24
12
45
34
67
请输出数组元素的内容:
23
24
12
45
34
67
请按任意键继续. . . .
```

# 4.5



## 类成员与指针

- 通过指向成员的指针只能访问公有成员
- 作用：通过指针访问类对象的内部
- 声明指向成员的指针方法
- 声明指向公有数据成员的指针
- 类型说明符 类名 ::\* 指针名
- 声明指向公有函数成员的指针
- 类型说明符 ( 类名 ::\* 指针名 )( 参数表 );



## 1 指向类的非静态成员的指针（续）

125

- 指向数据成员的指针的使用
- 说明指针应该指向哪个成员

**指针名 = & 类名 :: 数据成员名;**

- 通过对象名（或对象指针）与成员指针结合来访问数据成员

**对象名 . \* 类成员指针名**

**或:**

**对象指针名 -> \* 类成员指针名**



- 指向函数成员的指针的使用
- 初始化

**指针名 = 类名 :: 函数成员名;**

- 通过对象名（或对象指针）与成员指针结合来访问函数成员

**( 对象名 .\* 类成员指针名 )( 参数表 )**

**或:**

**( 对象指针名 ->\* 类成员指针名 )( 参数表 )**



# 1 指向类的非静态成员的指针

- 程序简单(续)例

```
#include <iostream.h>
class Point
{ int x,y;
public:
    Point(int a, int b)
        { x=a; y=b; }
    int GetX()
        { return x; }
    int GetY()
        { return y; }
}; // 待续...
```

```
// 续前页
void main()
{   Point A(4,5);           // 声明对象 A
    Point *p1=&A;           // 声明对象指针并初始化
    // 声明成员函数指针并初始化
    int (Point::*p_GetX)()=Point::GetX;
    // ( 1 ) 使用成员函数指针访问成员函数
    cout<<(A.*p_GetX())<<endl;
    // ( 2 ) 使用对象指针访问成员函数
    cout<<(p1->GetX())<<endl;
    // ( 3 ) 使用对象名访问成员函数
    cout<<A.GetX()<<endl;
}
```



- 静态成员在类中地位特殊
- 对类的静态成员的访问不依赖于对象
- 可以用普通的指针来指向和访问静态成员，即如 C 语言中所学方法一样使用指针

- 程序简单示例

```
#include <iostream.h>
class Point
{ int x,y;
public:
    static int count;
    Point(int a, int b)
    { x=a; y=b; count++; }
    ~Point() { count--; }
    int GetX() { return x; }
    static int GetC() { return count; }
}; // 待续...
```



## 2 指向类的静态成员的指针 (续)

131

```
// 续前页
int Point::count=0; // 静态数据成员定义性说明
void main()
{ // 声明一个 int 型指针，指向类的静态成员
  int *pcnt=&Point::count;
  Point A(4,5);      // 声明对象 A
  // 直接通过指针访问静态数据成员
  cout<<" Object id="<<*pcnt<<endl;
  // 指向函数的指针，指向类的静态成员函数
  int(*gc)()=Point::GetC;
  cout<<" Object id="<<gc()<<endl;
}
```



## 3 浅拷贝与深拷贝

132

- 对象间赋值 (=) 是一个拷贝过程
- 浅拷贝
  - 实现对象间数据元素的一一对应复制。
- 深拷贝
  - 当被复制的对象数据成员是指针类型时，不是复制该指针成员本身，而是将指针所指的对象进行复制。
- 系统提供的拷贝（如默认拷贝构造函数等）只能实现浅拷贝，深拷贝必须自定义



### 3 浅拷贝与深拷贝 (续)

133

- 浅拷贝示例

```
#include <iostream.h>
class Student
{
    int age;
    char *name;    // 注意这里, 有指针
public:
    Student(int a, char* n);
    ~Student();
    void Print();
};                // 待续...
```

```
Student::Student(int a, char* n)
{
    age=a;
    name=new char[strlen(n)+1];
    strcpy(name,n);
}
Student::~~Student()
{
    delete []name;
}
// 待续...
```



### 3 浅拷贝与深拷贝 (续)

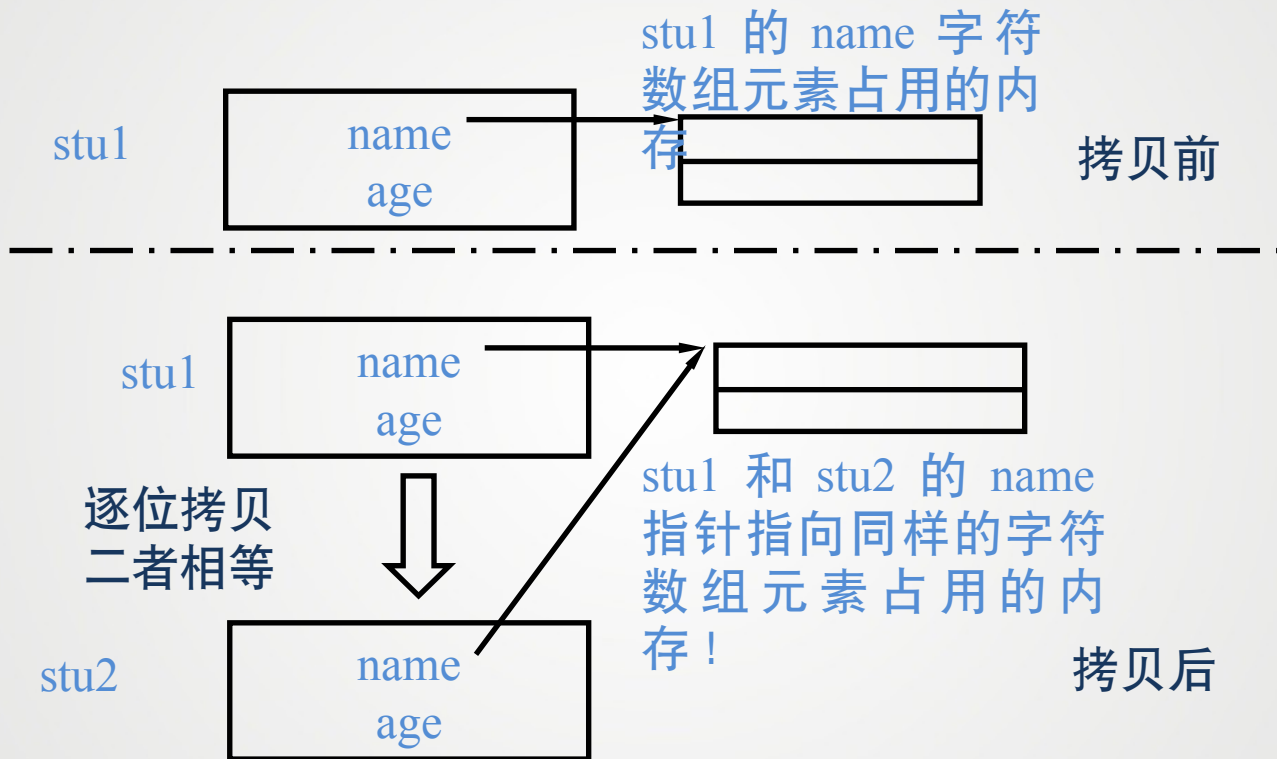
135

```
void Student::Print()
{
    cout<<"Age:"<<age<<endl;
    cout<<"Name:"<<name<<endl;
}
void main()
{
    Student stu1(20,"liu");
    Student stu2(stu);
}
// 该程序能正确运行吗?
```

(续)

- 当类中有指针数据成员时，并且为其开辟了空间，浅拷贝会出现问题！
- 程序使用了系统默认的拷贝构造函数  
`Student stu2(stu);`  
默认拷贝构造函数完成 `stu` 到 `stu2` 的逐位复制（浅拷贝）
- 浅拷贝在对象释放时会出现问题







(续)

- 问题原因

二个对象 name 指向同样内存， stu1 调用析构函数后， name 指向内存释放（ delete ）， 那么 stu2 调用析构函数释放谁呢？

- 问题解决

自定义拷贝构造函数（或赋值运算符， 后面讲解）， 自定义完成过程即为深拷贝！

- 深拷贝示例

```
#include <iostream.h>
class Student
{
    // ...
public:
    Student(int a, char* n);
    Student(const Student& s2); // 新增内容
    ~Student();
    // ...
}; // 待续...
```

### 3 浅拷贝与深拷贝 (续)

```
Student::Student(const Student& s2 )  
{  
    age=s2.age;  
    name=new char[strlen(s2.name)+1];  
    strcpy(name,s2.name);  
}  
// 其余代码不变...
```

- 函数中代码就是深拷贝，其示意图如下页



### 3 浅拷贝与深拷贝 (续)

141

