内联成员函数
与重载成员函数

郭 炜　刘家瑛

北京大学

内联成员函数

内联成员函数

- inline + 成员函数
- 整个函数体出现在类定义内部

```
class B{
1   inline void func1();
    void func2()
2   {
    };
};
void B::func1() { }
```

成员函数的重载及参数缺省

- 重载成员函数
- 成员函数 -- 带缺省参数

```
#include <iostream>
using namespace std;
class Location {
        private :
                int x, y;
       public:
          void init( int x=0 , int y = 0 );
          void valueX( int val ) { x = val ; }
          int valueX() { return x; }
};
```

```
void Location::init( int X, int Y){
    x = X;
    y = Y;
}
int main() {
    Location A;
    A.init(5);
    A.valueX(5);
    cout << A.valueX();
    return 0;
}
```

输出：
5

- 使用缺省参数要注意避免有函数重载时的二义性

```
class Location {
    private:
        int x, y;
    public:
       void init( int x =0, int y = 0 );
        void valueX( int val = 0 ) { x = val; }
        int valueX() { return x; }
};

    Location A;
    A.valueX();     //错误, 编译器无法判断调用哪个valueX
```

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

构造函数(constructor)

基本概念(教材P179)

- 成员函数的一种
  - 名字与类名相同，可以有参数，不能有返回值(void也不行)
  - 作用是对对象进行初始化，如给成员变量赋初值
  - 如果定义类时没写构造函数，则编译器生成一个默认的无参数的构造函数
    - 默认构造函数无参数，不做任何操作

基本概念

- 如果定义了构造函数，则编译器不生成默认的无参数的构造函数
- 对象生成时构造函数自动被调用。对象一旦生成，就再也不能在其上执行构造函数
- 一个类可以有多个构造函数

基本概念

- 为什么需要构造函数：

1) 构造函数执行必要的初始化工作，有了构造函数，就不必专门再写初始化函数，也不用担心忘记调用初始化函数。

2) 有时对象没被初始化就使用，会导致程序出错。

```
class Complex {
        private :
                double real, imag;
        public:
                void Set( double r, double i);
}; //编译器自动生成默认构造函数

Complex c1;  //默认构造函数被调用
Complex * pc = new Complex;  //默认构造函数被调用
```

```
class Complex {
        private :
                double real, imag;
        public:
                Complex( double r, double i = 0);
};
Complex::Complex( double r, double i) {
        real = r; imag = i;
}
Complex c1;  // error, 缺少构造函数的参数
Complex * pc = new Complex; // error, 没有参数
Complex c1(2); // OK
Complex c1(2,4), c2(3,5);
Complex * pc = new Complex(3,4);
```

- 可以有多个构造函数，参数个数或类型不同

```
class Complex {
        private :
                double real, imag;
        public:
                void Set( double r, double i );
                Complex(double r, double i );
                Complex (double r );
                Complex (Complex  c1,  Complex  c2);
};
Complex::Complex(double r, double i)
{
        real = r; imag = i;
}
```

```
Complex::Complex(double r)
{
        real = r; imag = 0;
}
Complex::Complex (Complex  c1,  Complex  c2);
{
        real = c1.real+c2.real;
        imag = c1.imag+c2.imag;
}
Complex c1(3) , c2 (1,0), c3(c1,c2);
// c1 = {3, 0}, c2 = {1, 0}, c3 = {4, 0};
```

- 构造函数最好是public的，private构造函数不能直接用来初始化对象

```
class CSample{
        private:
                CSample() {
                 }
};
int main(){
        CSample Obj; //err. 唯一构造函数是private
        return 0;
}
```

构造函数在数组中的使用

```
class CSample {
        int x;
public:
        CSample() {
                cout << "Constructor 1 Called" << endl;
        }
        CSample(int n) {
                x = n;
                cout << "Constructor 2 Called" << endl;
        }
};
```

```
int main(){
    CSample array1[2];
    cout << "step1"<<endl;
    CSample array2[2] = {4,5};
    cout << "step2"<<endl;
    CSample array3[2] = {3};
    cout << "step3"<<endl;
    CSample * array4 =
            new CSample[2];
    delete []array4;
    return 0;
}
```

输出：

```
Constructor 1 Called
Constructor 1 Called
step1
Constructor 2 Called
Constructor 2 Called
step2
Constructor 2 Called
Constructor 1 Called
step3
Constructor 1 Called
Constructor 1 Called
```

构造函数在数组中的使用

```
class  Test {
      public:
            Test( int n) { }                  //(1)
            Test( int n, int m) { }       //(2)
            Test() { }                          //(3)
};
Test  array1[3] = { 1, Test(1,2) };
// 三个元素分别用(1),(2),(3)初始化
Test  array2[3] = { Test(2,3), Test(1,2) , 1};
// 三个元素分别用(2),(2),(1)初始化
Test  * pArray[3] = { new Test(4), new Test(1,2) };
//两个元素分别用(1),(2) 初始化
```

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

复制构造函数
(copy constructor)

基本概念(教材P183)

- 只有一个参数，即对同类对象的引用。
- 形如 X::X(X&)或X::X(const X &)，二者选一

  后者能以常量对象作为参数
- 如果没有定义复制构造函数，那么编译器生成默认复制构造函数。默认的复制构造函数完成复制功能。

基本概念

```
class Complex {
        private :
                double real,imag;
};
Complex c1;      //调用缺省无参构造函数
Complex c2(c1);//调用缺省的复制构造函数,将 c2 初始化成和c1一样
```

基本概念

➢如果定义的自己的复制构造函数，
则默认的复制构造函数不存在。

```
class Complex {
        public :
                double real,imag;
        Complex(){ }
        Complex( const Complex & c ) {
                real = c.real;
                imag = c.imag;
                cout << “Copy Constructor called”;
        }
};
Complex c1;
Complex c2(c1);//调用自己定义的复制构造函数，输出 Copy Constructor called
```

基本概念

➢不允许有形如 X::X(X)的构造函数。

```
class CSample {
        CSample( CSample c ) {
        } //错，不允许这样的构造函数
};
```

复制构造函数起作用的三种情况

1) 当用一个对象去初始化同类的另一个对象时。

```
Complex  c2(c1);

Complex c2 = c1;  //初始化语句，非赋值语句
```

复制构造函数起作用的三种情况

2)如果某函数有一个参数是类 A 的对象，
那么该函数被调用时，类A的复制构造函数将被调用。

```
class A
{
        public:
        A() { };
        A( A & a) {
            cout << "Copy constructor called" <<endl;
        }
};
```

复制构造函数起作用的三种情况

2)如果某函数有一个参数是类 A 的对象，
那么该函数被调用时，类A的复制构造函数将被调用。

```
void Func(A a1){  }
int main(){
        A a2;
        Func(a2);
        return 0;
}
```

程序输出结果为: ***Copy constructor called***

复制构造函数起作用的三种情况

3) 如果函数的返回值是类A的对象时，则函数返回时，
A的复制构造函数被调用:

```
class A
{
  public:
  int v;
  A(int n) { v = n; };
  A( const A & a) {
            v = a.v;
            cout << "Copy constructor called" <<endl;
  }
};
```

复制构造函数起作用的三种情况

3) 如果函数的返回值是类A的对象时，则函数返回时，
A的复制构造函数被调用：

```
A Func() {
        A b(4);
        return b;
}
int main() {
        cout << Func().v << endl;
        return 0;
}
```

输出结果：

Copy constructor called

4

类型转换构造函数

郭 炜 刘家瑛

北京大学

类型转换构造函数

- 目的
  - 实现类型的自动转换
- 特点
  - 只有一个参数
  - 不是复制构造函数
- 编译系统会自动调用 → 转换构造函数

  → 建立一个 临时对象 / 临时变量

```
class Complex {
    public:
        double  real, imag;
        Complex( int i )  {    //类型转换构造函数
            cout << “IntConstructor called” << endl;
            real = i;   imag = 0;
         }
        Complex( double r, double i )
        {     real = r;  imag = i;   }
};
int main () {
    Complex  c1(7, 8);
    Complex  c2 = 12;
    c1 = 9;   // 9被自动转换成一个临时Complex对象
    cout << c1.real << "," << c1.imag << endl;
    return 0;
}
```

输出:
IntConstructor called
IntConstructor called
9,0

析构函数

郭 炜　刘家瑛

北京大学

析构函数（Destructor）

- 成员函数的一种
  - 名字与类名相同
  - 在前面加‘~’
  - 没有参数和返回值
  - 一个类最多只有一个析构函数

构造函数

- 成员函数的一种
- 名字与类名相同
- 可以有参数，不能有返回值
- 可以有多个构造函数

用来初始化对象

析构函数 (Destructor)

- 对象消亡时 → 自动被调用
  - 在对象消亡前做善后工作
    - 释放分配的空间等
- 定义类时没写析构函数，则编译器生成缺省析构函数
  - 不涉及释放用户申请的内存释放等清理工作
- 定义了析构函数，则编译器不生成缺省析构函数

```
class String{
    private :
        char * p;
    public:
        String () {
            p = new char[10];
        }
        ~ String ();
};
String ::~ String() {
    delete [] p;
}
```

析构函数和数组

- 对象数组生命期结束时

→ 对象数组的每个元素的析构函数都会被调用

```
class Ctest {
    public:
      ~Ctest()  {   cout<< "destructor called" << endl;   }
};
int main () {
      Ctest array[2];
      cout << "End Main" << endl;
      return 0;
}
```

输出:

End Main

destructor called

destructor called

析构函数和运算符 delete

- delete 运算导致析构函数调用

```
Ctest  * pTest;
pTest = new Ctest;        //构造函数调用
delete pTest;             //析构函数调用
-----------------------------------------------------------
pTest = new Ctest[3];  //构造函数调用3次
delete [] pTest;          //析构函数调用3次
```

构造函数和析构函数

调用时机的例题

```
class Demo {
        int id;
    public:
        Demo( int i )
        {
            id = i;
            cout << “id=” << id << “ Constructed” << endl;
        }
        ~Demo()
        {
             cout << “id=” << id << “ Destructed” << endl;
        }
};
```

```
Demo d1(1);
void Func(){
        static Demo d2(2);
        Demo d3(3);
        cout << “Func” << endl;
}
int main (){
        Demo d4(4);
        d4 = 6;
         cout << “main” << endl;
        {    Demo d5(5);        }
        Func();
         cout << “main ends” << endl;
        return 0;
}
```

输出:

```
id=1 Constructed
id=4 Constructed
id=6 Constructed
id=6 Destructed
main
id=5 Constructed
id=5 Destructed
id=2 Constructed
id=3 Constructed
Func
id=3 Destructed
main ends
id=6 Destructed
id=2 Destructed
id=1 Destructed
```

构造函数和析构函数在不同编译器中的表现

- 各别调用情况不一致
  - 编译器有bug
  - 代码优化措施
- 前面讨论的是C++标准

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

静态成员变量和静态成员函数

基本概念

- 静态成员：在说明前面加了static关键字的成员。

```
class CRectangle
{
        private:
             int w, h;
            static int nTotalArea;   //静态成员变量
             static int nTotalNumber;
        public:
             CRectangle(int w_,int h_);
              ~CRectangle();
             static void PrintTotal();   //静态成员函数
};
```

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。

sizeof 运算符不会计算静态成员变量。

```
class CMyclass  {
    int n;
    static int s;
};
```

则　sizeof(CMyclass) 等于 4

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。
- 普通成员函数必须具体作用于某个对象，而静态成员函数并不具体作用与某个对象。
- 因此静态成员不需要通过对象就能访问。

如何访问静态成员

1） 类名::成员名

CRectangle::PrintTotal();

2） 对象名.成员名

CRectangle r; r.PrintTotal();

3） 指针->成员名

CRectangle * p = &r; p->PrintTotal();

4） 引用.成员名

CRectangle & ref = r; int n = ref.nTotalNumber;

基本概念

- 静态成员变量本质上是全局变量，哪怕一个对象都不存在，类的静态成员变量也存在。
- 静态成员函数本质上是全局函数。
- 设置静态成员这种机制的目的是将和某些类紧密相关的全局变量和函数写到类里面，看上去像一个整体，易于维护和理解。

静态成员示例

考虑一个需要随时知道矩形总数和总面积的图形处理程序

可以用全局变量来记录总数和总面积

用静态成员将这两个变量封装进类中，就更容易理解和维护

```
class CRectangle
{
    private:
        int w, h;
        static int nTotalArea;
        static int nTotalNumber;
    public:
        CRectangle(int w_,int h_);
        ~CRectangle();
        static void PrintTotal();
};
```

```
CRectangle::CRectangle(int w_,int h_)
{
    w = w_;
    h = h_;
    nTotalNumber ++;
    nTotalArea += w * h;
}
CRectangle::~CRectangle()
{
    nTotalNumber --;
    nTotalArea -= w * h;
}
void CRectangle::PrintTotal()
{
    cout << nTotalNumber << "," <<   nTotalArea << endl;
}
```

```
int CRectangle::nTotalNumber = 0;
int CRectangle::nTotalArea = 0;
// 必须在定义类的文件中对静态成员变量进行一次说明
//或初始化。否则编译能通过，链接不能通过。
int main()
{
        CRectangle r1(3,3), r2(2,2);
        //cout << CRectangle::nTotalNumber; // Wrong , 私有
        CRectangle::PrintTotal();
        r1.PrintTotal();
        return 0;
}
```

输出结果：

```
2,13
2,13
```

注意事项

- 在静态成员函数中，不能访问非静态成员变量，也不能调用非静态成员函数。

```
void CRectangle::PrintTotal()
{
        cout << w << "," << nTotalNumber << "," << nTotalArea << endl;  //wrong
}
CRetangle::PrintTotal(); //解释不通，w 到底是属于那个对象的？
```

```
CRectangle::CRectangle(int w_,int h_)
{
    w = w_;
    h = h_;
    nTotalNumber ++;
    nTotalArea += w * h;
}
CRectangle::~CRectangle()
{
    nTotalNumber --;
    nTotalArea -= w * h;
}
void CRectangle::PrintTotal()
{
    cout << nTotalNumber << "," <<   nTotalArea << endl;
}
```

此CRectangle类写法，有何缺陷？

➢在使用CRectangle类时，有时会调用复制构造函数生成临时的隐藏的CRectangle对象

调用一个以CRectangle类对象作为参数的函数时，

调用一个以CRectangle类对象作为返回值的函数时

➢ 临时对象在消亡时会调用析构函数，减少nTotalNumber 和nTotalArea的值，可是这些临时对象在生成时却没有增加nTotalNumber 和 nTotalArea的值。

➢ 解决办法：为CRectangle类写一个复制构造函数。

```
CRectangle :: CRectangle(CRectangle & r )
{
        w = r.w;   h = r.h;
        nTotalNumber ++;
        nTotalArea += w * h;
}
```

成员对象和封闭类

郭 炜　刘家瑛

北京大学

成员对象和封闭类

- 成员对象: 一个类的成员变量是另一个类的对象
- 包含 成员对象 的类叫 封闭类 (Enclosing)

```
class CTyre {       //轮胎类
    private:
        int radius;   //半径
        int width;    //宽度
    public:
        CTyre(int r, int w):radius(r), width(w) {   }
};
class CEngine {   //引擎类
};
```

```
class CCar {          //汽车类 → “封闭类”
    private:
        int price;  //价格
        CTyre  tyre;
        CEngine engine;
    public:
        CCar(int p, int tr, int tw);
};
CCar::CCar(int p, int tr, int w):price(p), tyre(tr, w){

};
int main(){
    CCar car(20000,17,225);
    return 0;
}
```

- 如果 CCar 类不定义构造函数, 则

  CCar car; // error → 编译出错

  - 编译器不知道 car.tyre 该如何初始化
  - car.engine 的初始化没有问题: 用默认构造函数
- 生成封闭类对象的语句 →明确 “对象中的成员对象”

→ 如何初始化

封闭类构造函数的初始化列表

- 定义封闭类的构造函数时，添加初始化列表：

类名::构造函数(参数表):成员变量1(参数表)，成员变量2(参数表)，…

{

…

}

- 成员对象初始化列表中的参数
  - 任意复杂的表达式
  - 函数 / 变量/ 表达式中的函数，变量有定义

调用顺序

- 当封闭类对象生成时，
  - S1: 执行所有成员对象 的构造函数
  - S2: 执行 封闭类 的构造函数
- 成员对象的构造函数调用顺序
  - 和成员对象在类中的说明顺序一致
  - 与在成员初始化列表中出现的顺序无关
- 当封闭类的对象消亡时，
  - S1: 先执行 封闭类 的析构函数
  - S2: 执行 成员对象 的析构函数
- 析构函数顺序和构造函数的调用顺序相反

封闭类例子程序

```
class CTyre {
    public:
        CTyre() {  cout << "CTyre contructor" << endl; }
        ~CTyre() { cout << "CTyre destructor" << endl; }
};
class CEngine {
    public:
        CEngine() { cout << "CEngine contructor" << endl; }
        ~CEngine() { cout << "CEngine destructor" << endl; }
};
```

```
class CCar {
    private:
        CEngine engine;
        CTyre tyre;
    public:
        CCar( ) { cout << “CCar contructor” << endl; }
        ~CCar() { cout << "CCar destructor" << endl; }
};

int main(){
    CCar car;
    return 0;
}
```

程序的输出结果是：

CEngine contructor
CTyre contructor
CCar contructor
CCar destructor
CTyre destructor
CEngine destructor

友元 (Friend)

郭 炜　刘家瑛

北京大学

友元

- 友元函数
- 友元类

友元函数

- 一个类的友元函数可以访问该类的私有成员

```
class CCar;   //提前声明 CCar类, 以便后面CDriver类使用
class CDriver {
    public:
      void ModifyCar( CCar * pCar) ; //改装汽车
};
class CCar {
     private:
       int price;
     friend int MostExpensiveCar( CCar cars[], int total);   //声明友元
     friend void CDriver::ModifyCar(CCar * pCar);            //声明友元
};
```

```
void CDriver::ModifyCar( CCar * pCar)
{
	pCar->price += 1000; //汽车改装后价值增加
}
int MostExpensiveCar( CCar cars[], int total) //求最贵汽车的价格
{
	int tmpMax = -1;
	for( int i = 0; i < total; ++i )
		if( cars[i].price > tmpMax)
			tmpMax = cars[i].price;
	return tmpMax;
}
int main()
{
	return 0;
}
```

- 将一个类的成员函数(包括构造, 析构函数)

→ 另一个类的友元

```
class B {
    public:
        void function();
};

class A {
    friend void B::function();
};
```

友元类

- A是B的友元类 → A的成员函数可以访问B的私有成员

```
class CCar {
        private:
              int price;
         friend class CDriver;    //声明CDriver为友元类
};
class CDriver {
      public:
              CCar myCar;
              void ModifyCar() {   //改装汽车
                    myCar.price += 1000; // CDriver是CCar的友元类→可以访问其私有成员
          }
};
int main() { return 0; }
```

Note:
友元类之间的关系
不能传递，不能继承

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

this 指针

C++程序到C程序的翻译

```
class CCar  {
  public:
          int price;
          void SetPrice(int p);
};
void CCar::SetPrice(int p)
 {   price = p;  }
int main()
{
          CCar car;
          car.SetPrice(20000);
          return 0;
}
```

```
struct CCar  {
          int price;
};
void SetPrice(struct CCar * this,
          int p)
{    this->price = p;    }
int main()  {
          struct CCar car;
          SetPrice( & car,
          20000);
          return 0;
}
```

this指针作用

- 其作用就是指向成员函数所作用的对象

this指针作用

➢非静态成员函数中可以直接使用this来代表指向该函数作用的对象的指针。

```
class Complex {
  public:
    double real, imag;
    void Print() {  cout << real << "," << imag ;  }
    Complex(double r,double i):real(r),imag(i)
    {        }
    Complex AddOne()        {
        this->real ++;  //等价于 real ++;
        this->Print();    //等价于 Print
        return * this;
    }
};
```

```
int main()  {
    Complex c1(1,1),c2(0,0);
    c2 = c1.AddOne();
    return 0;
} //输出 2,1
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << "hello" << endl; }
};
int main()
{
        A * p = NULL;
        p->Hello();  //结果会怎样？
}
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << "hello" << endl; }
};   → void Hello(A * this ) { cout << "hello" << endl; }

int main()
{
        A * p = NULL;
        p->Hello(); → Hello(p);
} // 输出：hello
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << i << "hello" << endl; }
}; → void Hello(A * this ) { cout << this->i << "hello"
        << endl; }
 //this若为NULL，则出错！！
int main()
{
        A * p = NULL;
        p->Hello(); → Hello(p);
} // 输出：hello
```

this指针和静态成员函数

静态成员函数中不能使用 this 指针！

因为静态成员函数并不具体作用与某个对象！

因此，静态成员函数的真实的参数的个数，就是程序中写出的参数个数！

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

常量对象、常量成员函数和常引用

常量对象 (教材P194)

- 如果不希望某个对象的值被改变，则定义该对象的时候可以在前面加const关键字。

常量对象 (教材P194)

➢如果不希望某个对象的值被改变，则定义该对象的时候可以在前面加const关键字。

```
class Demo{
        private :
                int value;
        public:
                void SetValue() {       }
};
const Demo Obj;  // 常量对象
```

常量成员函数

- 在类的成员函数说明后面可以加const关键字，则该成员函数成为常量成员函数。

常量成员函数

- 在类的成员函数说明后面可以加const关键字，则该成员函数成为常量成员函数。

- 常量成员函数执行期间不应修改其所作用的对象。因此，在常量成员函数中不能修改成员变量的值（静态成员变量除外），也不能调用同类的非常量成员函数(静态成员函数除外）。

常量成员函数

```
class Sample
{
public:
        int value;
        void GetValue()  const;
        void func() { };
        Sample() { }
};
void Sample::GetValue() const
{
        value = 0; // wrong
        func(); //wrong
}
```

常量成员函数

```
class Sample
{
public:
        int value;
        void GetValue()  const;
        void func() { };
        Sample() { }
};
void Sample::GetValue() const
{
        value = 0; // wrong
        func(); //wrong
}
```

```
int main()  {
    const Sample o;
    o.value = 100; //err.常量对象不可被修改
    o.func(); //err.常量对象上面不能执行非常量成员函数
    o.GetValue(); //ok,常量对象上可以执行常量成员函数
    return 0;
} //在Dev C++中，要为Sample类编写无参构造函数才可以，Visual Studio 2010中不需要
```

常量成员函数的重载

- 两个成员函数，名字和参数表都一样，但是一个是const，一个不是，算重载。

常量成员函数的重载

```
class CTest {
        private :
                int n;
        public:
                CTest() { n = 1 ; }
                int GetValue()  const  { return n ; }
                int GetValue() { return 2 * n ; }
};
int main()  {
        const CTest objTest1;
        CTest objTest2;
        cout << objTest1.GetValue() << "," <<   objTest2.GetValue() ;
         return 0;
}
```

常引用

- 引用前面可以加const关键字，成为常引用。

不能通过常引用，修改其引用的变量。

常引用

- 引用前面可以加const关键字，成为常引用。

不能通过常引用，修改其引用的变量。

```
const int & r = n;
r = 5; //error
n = 4; //ok
```

常引用

对象作为函数的参数时，生成该参数需要调用复制构造函数，效率比较低。用指针作参数，代码又不好看，如何解决？

常引用

可以用对象的引用作为参数，如：

```
class Sample {
    …
};
void PrintfObj(Sample &  o)
{
    ……
}
```

常引用

可以用对象的引用作为参数，如：

```
class Sample {
    …
};
void PrintfObj(Sample &  o)
{
    ……
}
```

对象引用作为函数的参数有一定风险性，若函数中不小心修改了形参o，则实参也跟着变，这可能不是我们想要的。如何避免？

常引用

可以用对象的常引用作为参数，如：

```
class Sample {
    …
};
void PrintfObj( const Sample &  o)
{
    ……
}
```

这样函数中就能确保不会出现无意中更改o值的语句了。