

2019年春

程序设计实习(I): C++程序设计

第二讲 类和对象 (1)

刘家瑛

liujiaying@pku.edu.cn



主要内容

□ 面向对象的基本概念

- 例子——矩形类
- 对象的内存分配 与 运算符
- 三种方式使用
- 引用 & 常引用

□ 构造函数

□ 复制构造函数

□ 析构函数



面向对象的程序设计

□ 面向对象的程序设计方法:

- 将某类客观事物**共同特点** [属性] 归纳出来

- 形成一个**数据结构** (用多个变量描述事物的属性)

- 将这类事物所能进行的**行为** 也归纳出来

- 形成一个个**函数**

- 这些函数可以用来操作数据结构

抽象



面向对象的程序设计

□ 通过某种语法形式,

将数据结构和操作该数据结构的函数 "捆绑"

在一起 → 形成一个 "类"

□ 使得数据结构和操作该数据结构的算法呈现出显而易见的紧密关系

封装



从客观事物抽象出类的例子

- 写一个程序, 输入矩形的长和宽, 输出面积和周长
 - 如对于 "矩形" 这种**对象**, 要用一个**类**来表示, 该如何做 "抽象" 呢?
 - 矩形的**属性**就是长和宽
 - 因此需要两个变量, 分别代表长和宽
 - 可以对矩形进行哪些**操作**
 - 矩形可以有设置长和宽, 计算面积和计算周长这三种行为
 - 这三种行为, 可以各用一个**函数**来实现, 都需要用到长和宽这两个变量



从客观事物抽象出类的例子

- 将长, 宽变量和设置长, 宽, 求面积, 以及求周长的三个函数封装在一起, 就能形成一个**"矩形类"**
 - 长和宽变量成为该"矩形类"的**"成员变量"**
 - 三个函数成为该类的**"成员函数"**
 - 成员变量和成员函数统称为类的成员
- **类 → 带函数的结构**



从客观事物抽象出类的例子

```
class CRectangle
{
    public:
        int w, h;
        int Area() {
            return w * h;
        }
        int Perimeter() {
            return 2 * (w + h);
        }
        void Init(int w_, int h_) {
            w = w_; h = h_;
        }
}; //必须有分号
```



从客观事物抽象出类的例子

```
int main( ) {  
    int w, h;  
    CRectangle r; //r是CRectangle类的一个对象  
    cin >> w >> h;  
    r.Init(w, h);  
    cout << r.Area() << endl << r.Perimeter();  
    return 0;  
}
```

- 通过类, 可以定义变量
- 类定义出来的变量, 也称为类的实例, 即 "对象"
- C++中, 类的名字就是用户自定义的类型的名字
可以像使用基本类型那样来使用它
 - CRectangle 就是一种用户自定义的类型



对象的内存分配

- 和结构变量一样, **对象**所占用的**内存空间的大小**,
= 所有成员变量的大小之和
- 对于上面的**CRectangle**类, **sizeof(CRectangle) = 8**
- 每个对象各有自己的存储空间
- 一个对象的某个成员变量被改变了, 不会影响到另一个对象



对象间的运算

- 和结构变量一样, 对象之间可以用 "=" 进行赋值
- 不能用 "==" "!=" ">" "<" ">=" "<=" 进行比较, 除非这些运算符经过了 "重载"

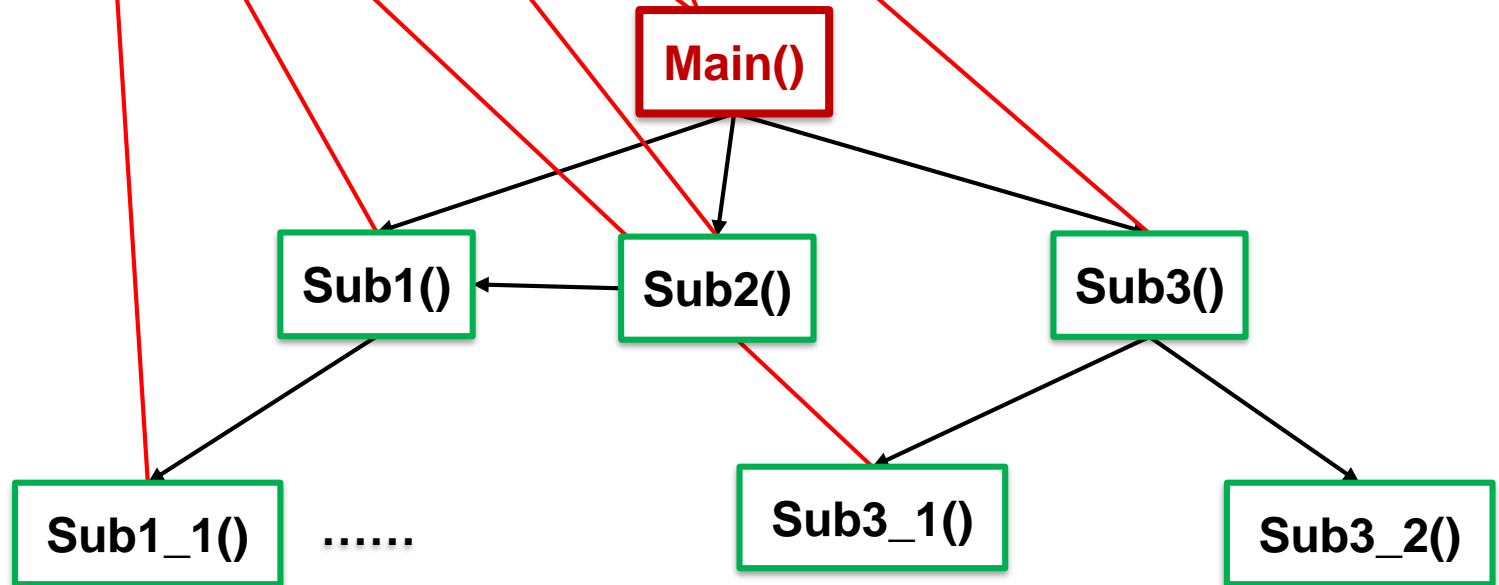


结构化程序设计的程序模式

变量:



函数:



面向对象的程序设计

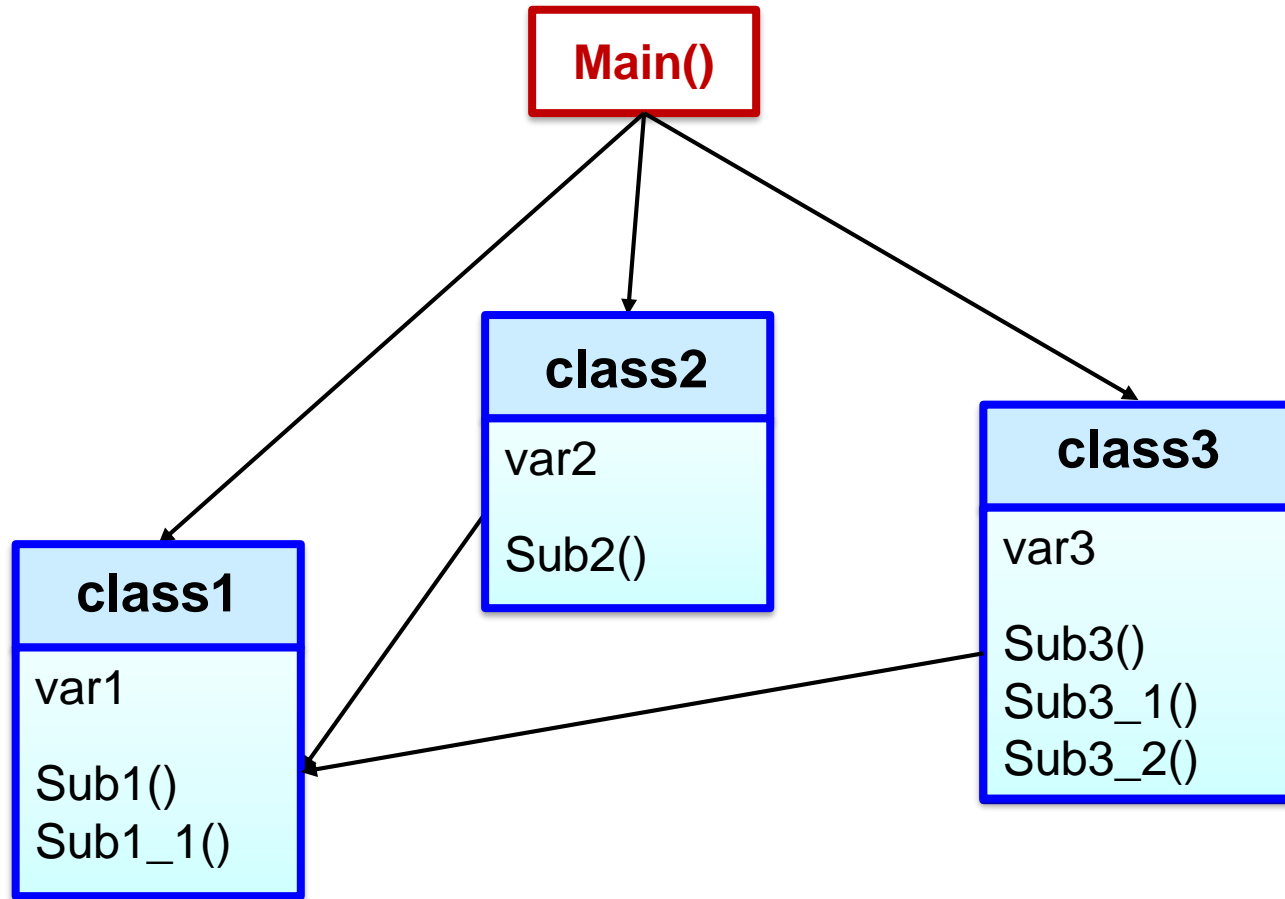
- 面向对象的程序设计方法，能够较好解决上述问题

面向对象的程序 = 类 + 类 + ... + 类

- 设计程序的过程，就是设计类的过程



面向对象的程序模式



面向对象的程序设计

□ 面向对象的程序设计方法：

- 将某类客观事物共同特点(属性)归纳出来，形成一个数据结构(可以用多个变量描述事物的属性)；
 - 将这类事物所能进行的行为也归纳出来，形成一个个函数，这些函数可以用来操作数据结构(这一步叫“抽象”)
- ## □ 通过某种语法形式，将数据结构和操作该数据结构的函数“捆绑”在一起，形成一个“类”
- ## □ 使得数据结构和操作该数据结构的算法呈现出显而易见的紧密关系，这就是“封装”



如何使用类的成员变量和成员函数

用法1: 对象名.成员名

```
CRectangle r1, r2;
```

```
r1.w = 5;
```

```
r2.Init(3,4);
```

- **Init**函数作用在 **r2**上, 即**Init**函数执行期间访问的**w**和**h**是属于**r2**这个对象的, 执行**r2.Init**不会影响到**r1**

用法2: 指针->成员名

```
CRectangle r1, r2;
```

```
CRectangle * p1 = & r1;
```

```
CRectangle * p2 = & r2;
```

```
p1->w = 5;
```

```
p2->Init(3,4); //Init作用在p2指向的对象上
```



如何使用类的成员变量和成员函数

用法3: 引用名.成员名

```
CRectangle r2;  
CRectangle & rr = r2;  
rr.w = 5;  
rr.Init(3, 4); //rr的值变了, r2的值也变
```

```
void PrintRectangle(CRectangle & r)  
{  
    cout << r.Area() << ", " << r.Perimeter();  
}  
CRectangle r3;  
r3.Init(3, 4);  
PrintRectangle(r3);
```



引用的概念

类型名 & 引用名 = 某变量名;

- 定义了一个引用, 并将其初始化为引用某个变量
- 某个变量的引用, 和这个变量是一回事, 相当于该变量的一个别名

```
int n = 4;  
int & r = n;  
r = 3;  
cout << r; //输出 3  
cout << n; //输出 3  
n = 5;  
cout << r; //输出 5
```

- 定义引用时一定要将其初始化成引用某个变量, 不初始化编译不过
- 引用只能引用变量, 不能引用常量和表达式



引用的作用

□ C语言中, 交换两个整型变量值的函数, 只能通过指针

```
void swap( int * a, int * b )  
{  
    int tmp;  
    tmp = * a; * a = * b; * b = tmp;  
}  
int n1, n2;  
swap(&n1, &n2) ;    // n1, n2的值被交换
```



引用的作用

□ 使用引用：

```
void swap( int & a, int & b )  
{  
    int tmp;  
    tmp = a; a = b; b = tmp;  
}  
int n1, n2;  
swap(n1, n2) ; // n1, n2的值被交换
```



引用作为函数的返回值

□ 函数的返回值可以是引用, 如:

```
#include <iostream>
using namespace std;
int n = 4;
int & SetValue() { return n; }
//返回对n的引用
int main()
{
    SetValue() = 40; //对返回值进行赋值→对n赋值
    cout << n;
    return 0;
} //程序输出结果是40
```



常引用

□ 定义引用时,

在前面加 **const** 关键字 — 该引用为"常引用"

```
int n;
```

```
const int & r = n;
```

不能通过常引用去修改其引用的内容

```
int n = 100;
```

```
const int & r = n;
```

```
r = 200; //编译出错,
```

//不能通过常引用修改其引用的内容

```
n = 300; //没问题,n的值变为300
```



常引用

- 请注意, **const T &** 和 **T &** 是不同的类型
- **T &**类型的引用或 **T**类型的变量可以用来初始化 **const T &**类型的引用
- **const T** 类型的常变量和**const T &**类型的引用则不能用来初始化**T &**类型的引用, 除非进行强制类型转换



类的成员函数的另一种写法

□ 成员函数体和类的定义分开写

```
class CRectangle
{
    public:
        int w, h;
        int Area(); //成员函数仅在此处声明
        int Perimeter();
        void Init( int w_, int h_ );
};
```



类的成员函数的另一种写法

```
int CRectangle::Area() {  
    return w * h;  
}  
int CRectangle::Perimeter() {  
    return 2 * ( w + h );  
}  
void CRectangle::Init( int w_, int h_ )  
{  
    w = w_;    h = h_;  
}
```

CRectangle::说明后面的函数是**CRectangle**类的成员函数,而非普通函数

一定要通过对象或对象的指针或对象的引用才能调用



对象成员的访问权限

在类的定义中，

用下列权限关键字——说明对象成员的访问权限：

- **private**: 私有成员, 只能在成员函数内访问
- **public**: 公有成员, 可以在任何地方访问
- **protected**: 保护成员 [以后再说]

以上三种关键字出现的次数和先后次序都没有限制



对象成员的访问权限

- 定义一个类

```
class className {
```

```
    private:
```

私有属性和函数

```
    public:
```

公有属性和函数

```
    protected:
```

保护属性和函数

```
};
```

说明类成员
的可见性



对象成员的访问权限

- 如果某个成员前面没有上述关键字，
则缺省认为是私有成员

```
class Man {  
    int nAge;           //私有成员  
    char szName[20];    //私有成员  
public:  
    void SetName(char * szName) {  
        strcpy(Man::szName, szName);  
    }  
};
```



对象成员的访问权限

- 在类的成员函数内部, 能够访问:
 - 当前对象的全部属性, 函数
 - 同类其它对象的全部属性, 函数
- 在类的成员函数以外的地方, 只能够访问该
类对象的公有成员



```
class CEmployee {  
    private:  
        char szName[30]; //名字  
    public:  
        int salary; //工资  
        void setName(char * name);  
        void getName(char * name);  
        void averageSalary(CEmployee e1,  
CEmployee e2);  
};  
void CEmployee::setName(char * name) {  
    strcpy( szName, name); //ok  
}  
void CEmployee::getName(char * name) {  
    strcpy( name, szName); //ok  
}
```




```
void CEmployee::averageSalary(CEmployee e1,
CEmployee e2)
{
    salary = (e1.salary + e2.salary)/2;
}
int main()
{
    CEmployee e;
    strcpy(e.szName, "Tom1234567889"); //编译错,
                                         //不能访问私有成员
    e.setName("Tom"); // ok
    e.salary = 5000;  // ok
    return 0;
}
```



```
int main() {  
    CEmployee e;  
    strcpy(e.szName, "Tom1234567889");  
    //编译错, 不能访问私有成员  
    e.setName( "Tom"); // ok  
    e.salary = 5000;    // ok  
    return 0;  
}
```

□ 设置私有成员的目的

- ① 强制对成员变量的访问一定要通过成员函数进行
- ② 方便修改成员变量的类型等属性→只需更改成员函数

Vs. 否则, 所有直接访问成员变量的语句都需要修改

□ 设置私有成员的机制叫隐藏



□ 例如, 如将上述程序 → 内存空间紧张的手持设备上
将 `szName` 改为 `char szName[5]`,
若 `szName` 不是私有, 那么就要找出所有类似

```
strcpy(man1.szName, "Tom1234567889");
```

这样的语句进行修改, 以防止数组越界 → 太麻烦!

□ 如果将 `szName` 变为私有,
那么程序中就不会出现 (除非在类的内部)

```
strcpy(man1.szName, "Tom1234567889");
```



所有对 **szName** 的访问都是通过成员函数来进行,

例如:

```
man1.setName( "Tom12345678909887" );
```

□ 如需将 **szName** 改短, 也不需将上面的语句找出来修改,
只要改 setName 成员函数, 在里面确保不越界即可



主要内容

□ 面向对象的基本概念

- 例子—矩形类
- 对象的内存分配 与 运算符
- 三种方式使用
- 引用 & 常引用

□ 构造函数

□ 复制构造函数

□ 析构函数



主要内容

□ 面向对象的基本概念

- 例子—矩形类
- 对象的内存分配 与 运算符
- 三种方式使用
- 引用 & 常引用

□ 构造函数

□ 复制构造函数

□ 析构函数



构造函数 (Constructor)

- 在程序没有明确进行初始化的情况下,
 - 全局基本类型变量—被自动初始化成全0
 - 局部基本类型变量—初始值随机
- 构造函数是**成员函数**的一种, 用来**初始化对象**
 - 名字与类名相同, 可以有参数, 不能有返回值
 - Note: void也不行**
 - 作用: 为对象进行初始化, 如给成员变量赋初值
- 如果定义类时没写构造函数
 - 编译器生成一个缺省无参数的构造函数
 - 缺省构造函数无参数, 什么也不做



- 如果定义了构造函数, 则编译器不生成缺省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
- 为类编写构造函数是好的习惯, 能够保证对象生成的时候总是有合理的值
- 一个类可以有多个构造函数
- 构造函数执行时对象的内存空间已经分配好了, 用于初始化这片空间



```
class Complex {  
    private :  
        double real, imag;  
    public:  
        void Set( double r, double i );  
}; // 缺省构造函数
```

`Complex c1;` // 构造函数被调用

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



```
class Complex {  
    private :  
        double real, imag;  
    public:  
        Complex(double r, double i = 0); //构造函数  
};  
Complex::Complex(double r, double i){  
    real = r;    imag = i;  
}  
Complex c1; // error, 没有参数  
Complex * pc = new Complex; // error, 没有参数  
Complex c2(2); // OK, Complex c2(2,0)  
Complex c3(2, 4), c4(3, 5); // OK  
Complex * pc = new Complex(3, 4); // OK
```



□ 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() { }  
};  
main(){  
    CSample Obj; //err. 唯一构造函数是private  
}
```



开放思考题

- 如何写一个类，
使该类只能有一个对象



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;  
}
```



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
```



例题改一改

```
class Test {  
    public:  
        Test (int n) { cout << n << "(1)" << endl; }           //(1)  
        Test (int n, int m) { cout << n << m << "(2)" << endl; } // (2)  
        Test ( ) { cout << "(3)" << endl; }                     //(3)  
};  
  
int main() {  
    Test array1[3] = { 1, Test(1, 2) };  
  
    Test array2[3] = { 1, (1, 2) };  
  
    return 0;  
}
```



例题改一改

```
class Test {  
    public:  
        Test (int n) { cout << n << "(1)" << endl; }           //(1)  
        Test (int n, int m) { cout << n << m << "(2)" << endl; } //(2)  
        Test ( ) { cout << "(3)" << endl; }                       //(3)  
};  
  
int main() {  
    Test array1[3] = { 1, Test(1, 2) };  
    // 三个元素分别用(1),(2),(3)初始化  
    Test array2[3] = { 1, (1, 2) };  
    // 三个元素分别用(1),(1),(3)初始化  
    return 0;  
}
```



例题改一改

```
class Test {  
    public:  
        Test (int n) { cout << n << "(1)" << endl; }           //(1)  
        Test (int n, int m) { cout << n << m << "(2)" << endl; } // (2)  
        Test ( ) { cout << "(3)" << endl; }                     //(3)  
};  
  
int main() {  
    Test array1[3] = { 1, Test(1, 2) };  
    // 三个元素分别用(1),(2),(3)初始化  
    Test array2[3] = { 1, (1, 2) };  
    // 三个元素分别用(1),(1),(3)初始化  
    return 0;  
}
```



主要内容

□ 面向对象的基本概念

- 例子—矩形类
- 对象的内存分配 与 运算符
- 三种方式使用
- 引用 & 常引用

□ 构造函数

□ 复制构造函数

□ 析构函数



复制构造函数 (拷贝构造函数)

- 形如 **X::X(X&)**, 只有一个参数即对同类对象的引用
- 如果没有定义复制构造函数, 那么编译器生成缺省复制构造函数

- 缺省的复制构造函数完成复制功能

```
class Complex {  
    private :  
        double real, imag;  
};
```

```
Complex c1;           //调用缺省构造函数  
Complex c2(c1);       //调用缺省的复制构造函数  
                      //将 c2 初始化成和c1一样
```



□ 如果定义的自己的复制构造函数, 则缺省的复制构造函数不存在

```
class Complex {  
    public :  
        double real, imag;  
    Complex(){}  
    Complex(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
```

❑ 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
```

```
};
```



复制构造函数在以下三种情况被调用:

a. 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
```



b.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;  
        }  
};  
void Func(A a){ }  
int main(){  
    A a;  
    Func(a);  
    return 0;  
}
```

**程序输出结果为:
Copy constructor called**



c. 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;  
        }  
};
```

输出结果:

Copy constructor called
4

经过优化的编译器可能导致结果不同

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

