



北京大学
PEKING UNIVERSITY

信息科学技术学院

程序设计实习

C++面向对象程序设计

郭 炜

微信公众号



微博: <http://weibo.com/guoweiofpku>

学会程序和算法，走遍天下都不怕!

讲义内照片均为郭炜拍摄



北京大学
PEKING UNIVERSITY

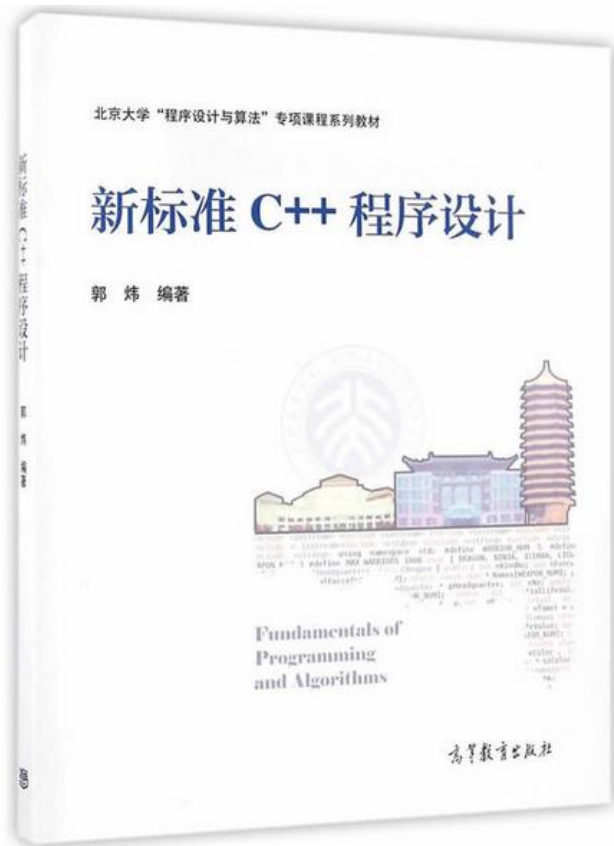
信息科学技术学院

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著





类和对象 (2)



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

类成员的 可访问范围



韩国济州岛火山口

类成员的可访问范围

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围：
 - **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。

类成员的可访问范围

➤ 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  
};
```



说明类成员的可访问范围

类成员的可访问范围

- 如过某个成员前面没有上述关键字，则缺省地被认为是私有成员。

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

类成员的可访问范围

- 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) {
    cout << e1.szName; //ok, 访问同类其他对象私有成员
    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;
}
```

➤ 设置私有成员的机制, 叫 “隐藏”

➤ “隐藏” 的目的是强制对成员变量的访问一定要通过成员函数进行, 那么以后成员变量的类型等属性修改后, 只需要更改成员函数即可。否则, 所有直接访问成员变量的语句都需要修改。

“隐藏”的作用

➤如果将上面的程序移植到内存空间紧张的手持设备上，希望将 szName 改为 char szName[5]，若szName不是私有，那么就要找出所有类似

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

这样的语句进行修改，以防止数组越界。这样做很麻烦。

“隐藏”的作用

- 如果将szName变为私有，那么程序中就不可能出现（除非在类的内部）

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

这样的语句，所有对 szName的访问都是通过成员函数来进行，比如：

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

- 那么，就算szName改短了，上面的语句也不需要找出来修改，只要改 setName成员函数，在里面确保不越界就可以了。

用struct定义类

```
struct CEmployee {  
    char szName[30]; //公有!!  
public :  
    int salary; //工资  
    void setName(char * name);  
    void getName(char * name);  
    void averageSalary(CEmployee  
        e1,CEmployee e2);  
};
```

和用"class"的**唯一**区别，就是未说明是公有还是私有的成员，就是**公有**



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

成员函数的 重载及参数缺省



内蒙古阿斯哈图石林

成员函数的重载及参数缺省

- 成员函数也可以重载
- 成员函数可以带缺省参数。

```
#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; }
};
```


成员函数的重载及参数缺省

- 成员函数也可以重载
- 成员函数可以带缺省参数。

```
void Location::init( int X, int Y)
{
    x = X;
    y = Y;
}
```

```
int main() {  
    Location A,B;  
    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**



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

构造函数 (constructor)



美国加州太浩湖

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非参数的构造函数
- 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。

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;  
}
```

单选题 1分



有类A如下定义:

```
class A {  
    int v;  
    public:  
    A ( int n) { v = n; }  
};
```

下面哪条语句是编译不会出错的?

- ☒ A A a1(3);
- ☐ B A a2;
- ☐ C A * p = new A();
- ☐ D 都错了

提交



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

构造函数 在数组中的使用



内蒙古阿斯哈图石林

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;  
}
```

```
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

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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) };  
Test array2[3] = { 1, {1,2} };
```

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
```

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};
```

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
```

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) };
```

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
```

假设 A 是一个类的名字，下面的语句生成了几个类A的对象？

`A * arr[4] = { new A(), NULL, new A() };`

- ☐ A 1
- ☐ B 4
- ☒ C 2
- ☐ D 142



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

复制构造函数 copy constructor



美国黄石公园大棱镜温泉

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);
```

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。

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

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。

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。

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;  
}
```

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：

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;  
}
```


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

注意：对象间赋值并不导致复制构造函数被调用

```
class CMyclass {  
    public:  
    int n;  
    CMyclass() {};  
    CMyclass( CMyclass & c) { n = 2 * c.n ; }  
};  
  
int main() {  
    CMyclass c1,c2;  
    c1.n = 5;      c2 = c1;      CMyclass c3(c1);  
    cout <<"c2.n=" << c2.n << ", ";  
    cout <<"c3.n=" << c3.n << endl;  
    return 0;  
}
```

输出： c2.n=5,c3.n=10

常量引用参数的使用

```
void fun(CMyclass obj_ ) {  
    cout << "fun" << endl;  
}
```

- 这样的函数，调用时生成形参会引发复制构造函数调用，开销比较大。
- 所以可以考虑使用 `CMyclass &` 引用类型作为参数。
- 如果希望确保实参的值在函数中不应被改变，那么可以加上 `const` 关键字：

```
void fun(const CMyclass & obj) {  
    //函数中任何试图改变 obj值的语句都将是变成非法  
}
```

假设A 是一个类的名字，下面哪段程序不会调用A的复制构造函数？

- ☒ A `A a1,a2; a1 = a2;`
- ☐ B `void func(A a) { cout << "good" << endl; }`
- ☐ C `A func() { A tmp; return tmp; }`
- ☐ D `A a1; A a2(a1);`

为什么要自己写复制构造函数？

为什么要自己写复制构造函数？

To be continued.....



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

类型转换构造函数



黄山

什么是类型转换构造函数

- ❑ 定义转换构造函数的目的是实现类型的自动转换。
- ❑ 只有一个参数，而且不是复制构造函数的构造函数，一般就可以看作是转换构造函数。
- ❑ 当需要的时候，编译系统会自动调用转换构造函数，建立一个无名的临时对象(或临时变量)。

类型转换构造函数实例

```
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;
}
```

类型转换构造函数实例

```
class Complex {
public:
    double  real, imag;
    explicit Complex( int i)  { //显式类型转换构造函数
        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 = Complex(12);
    c1 = 9; // error, 9不能被自动转换成一个临时Complex对象
    c1 = Complex(9) //ok
    cout << c1.real << "," << c1.imag << endl;
    return 0;
}
```

单选题 1分



类A定义如下:

```
class A {  
    int v;  
    public:  
        A(int i) { v = i; }  
        A() {}  
};
```

下面哪段程序不会引发类型转换构造函数被调用?

A

A a1(4)

C

A a3; a3 = 9;

B

A a2 = 4;

D

A a1,a2; a1 = a2;

提交



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

析构函数 destructors



华山

什么是析构函数

- 名字与类名相同，在前面加 '~' ， 没有参数和返回值，一个类最多只能有一个析构函数。

什么是析构函数

- 名字与类名相同，在前面加 '~'，没有参数和返回值，一个类最多只能有一个析构函数。
- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。

什么是析构函数

- 名字与类名相同，在前面加 '~'，没有参数和返回值，一个类最多只能有一个析构函数。
- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。
- 如果定义类时没写析构函数，则编译器生成缺省析构函数。缺省析构函数什么也不做。

什么是析构函数

- 名字与类名相同，在前面加 '~'，没有参数和返回值，一个类最多只能有一个析构函数。
- 析构函数对象消亡时即自动被调用。可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等。
- 如果定义类时没写析构函数，则编译器生成缺省析构函数。缺省析构函数什么也不做。
- 如果定义了析构函数，则编译器不生成缺省析构函数。

析构函数实例

```
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;  
}
```

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
```

- 若new一个对象数组，那么用delete释放时应该写 []。否则只delete一个对象(调用一次析构函数)

析构函数在对象作为函数返回值返回后被调用

```
class CMyclass {
public:
    ~CMyclass() { cout << "destructor" << endl; }
};

CMyclass obj;

CMyclass fun(CMyclass sobj ) { //参数对象消亡也会导致析
                               //构造函数被调用
    return sobj;              //函数调用返回时生成临时对象返回
}

int main(){
    obj = fun(obj);           //函数调用的返回值（临时对象）被
    return 0;                 //用过后，该临时对象析构函数被调用
}
```

析构函数在对象作为函数返回值返回后被调用

```
class CMyclass {
public:
    ~CMyclass() { cout << "destructor" << endl; }
};

CMyclass obj;

CMyclass fun(CMyclass sobj ) { //参数对象消亡也会导致析
                               //构造函数被调用
    return sobj;              //函数调用返回时生成临时对象返回
}

int main(){
    obj = fun(obj);           //函数调用的返回值（临时对象）被
    return 0;                  //用过后，该临时对象析构函数被调用
}
```

输出:

```
destructor
destructor
destructor
```



北京大学
PEKING UNIVERSITY

信息科学技术学院 郭炜

构造函数和析构函数
什么时候被调用？



梵蒂冈

构造函数和析构函数什么时候被调用？（P190）

```
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;  
}
```

```
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;
}
```

输出结果:
id=1 constructed

```
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;  
}
```

输出结果:

id=1 constructed

id=4 constructed

```
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;  
}
```

输出结果:

id=1 constructed

id=4 constructed

id=6 constructed

```
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;
}
```

输出结果:

id=1 constructed
id=4 constructed
id=6 constructed
id=6 destructed
main

```
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;  
}
```

输出结果:

id=1 constructed
id=4 constructed
id=6 constructed
id=6 destructed
main
id=5 constructed

```
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;  
}
```

输出结果:

id=1 constructed
id=4 constructed
id=6 constructed
id=6 destructed
main
id=5 constructed
id=5 destructed

```
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;
}
```

输出结果:

id=1 constructed
id=4 constructed
id=6 constructed
id=6 destructed
main
id=5 constructed
id=5 destructed
id=2 constructed


```
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;
}
```

输出结果:

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

```
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;  
}
```

输出结果:

*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*

```
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;
}
```

输出结果:

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

```
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;
}
```

输出结果:

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

```
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;
}
```

输出结果:

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

假设A是一个类的名字，下面的程序片段会类A的调用析构函数几次？

```
int main() {  
    A * p = new A[2];  
    A * p2 = new A;  
    A a;  
    delete [] p;  
}
```

A 1

☒ C 3

B 2

D 4

提交

关于复制构造函数和析构函数的又一个例子

```
#include <iostream>
using namespace std;
class CMyclass {
public:
    CMyclass() {} ;
    CMyclass( CMyclass & c)
    {
        cout << "copy constructor" << endl;
    }
    ~CMyclass() { cout << "destructor" << endl; }
};
```

```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

copy constructor


```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

copy constructor
fun

```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

copy constructor

fun

destructor

//参数消亡

```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

copy constructor

fun

destructor

test

//参数消亡

```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

copy constructor
fun
destructor //参数消亡
test
copy constructor

```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

```
copy constructor
fun
destructor          //参数消亡
test
copy constructor
destructor          // 返回值临时对
                     象消亡
```

```
void fun(CMyclass obj_ )
{
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test( )
{
    cout << "test" << endl;
    return c;
}
int main(){
    CMyclass c1;
    fun( c1);
    Test();
    return 0;
}
```

输出结果:

```
copy constructor
fun
destructor          //参数消亡
test
copy constructor
destructor          // 返回值临时对
                    象消亡
destructor          // 局部变量消亡
destructor          // 全局变量消亡
```

复制构造函数在不同编译器中的表现

```
class A {  
    public:  
        int x;  
        A(int x_):x(x_)  
            { cout << x << " constructor called" << endl; }  
        A(const A & a ) { //本例中dev需要此const其他编译器不要  
            x = 2 + a.x;  
            cout << "copy called" << endl;  
        }  
        ~A() { cout << x << " destructor called" << endl; }  
};
```

```
A f( ){    A b(10);    return b; }  
  
int main( ){  
    A a(1);  
    a = f();  
    return 0;  
}
```

**说明dev出于优化目的
并未生成返回值临时对
象。vs无此问题**

dev C++输出结果:

```
1 constructor  
called  
10 constructor  
called  
10 destructor  
called  
10 destructor  
called  
10 destructor  
called
```

**Visual Studio
输出结果:**

```
1 constructor  
called  
10 constructor  
called  
10 destructor  
called  
copy called  
12 destructor  
called  
12 destructor  
called
```