新标准C++程序设计

北京大学信息学院　郭 炜

GWPL@PKU.EDU.CN

类和对象(二)

内容提要

- 构造函数，析构函数，各种构造函数和析构函数的调用时机
- 静态成员
- 常量对象和常量方法
- 成员对象和封闭类
- const成员和引用成员
- 友元
- this指针

构造函数（constructor）

- 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，则编译器生成一个缺省的无参数的构造函数。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。

```
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);
};
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 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
}
```

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
```

复制构造函数(拷贝构造函数)

➢形如 X::X(X&)或X::X(const 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( 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
```

➢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
};
```

➢即使缺省的不带参数的构造函数不存在，缺省的复制构造函数仍然存在。

```
class C1{
public:
      C1() {
      }
};
C1 c1,c2(c1);
```

➢ 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;
```

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。

```
void f( A  a ) {
    a.x = 1;
};
A  aObj;
f ( aObj) ;  // 导致A的复制构造函数被调用，生成形
             // 参传入函数
```

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:

```
A f( ) {
    A a;
    return a;  // 此时A的复制构造函数被调用,
               //即调用A(a);
}
 int main( ) {
    A b;
    b = f();
    return 0;
}
```

注意：对象间用等号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

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)。

```
#include <iostream>
using namespace std;
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;
}
```

程序输出结果是：

```
IntConstructor called
IntConstructor called
9,0
```

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
}
```

析构函数（destructors）

- 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。

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

- 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
```

- ✓ 总之，在临时对象生成的时候会有构造函数被调用；临时对象消亡导致析构函数调用。

构造函数和析构函数什么时候被调用？

```
//假定 chunk 是一个类的名字
chunk ck1,ck2;  //main 执行前即调用构造函数
void MyFunction() {
    static chunk ck; //函数执行前调用构造函数
    chunk tmpCk; //tmpCk 构造函数调用
      ....//函数退出时 tmpCk消亡，调用析构函数
}
int main () {
      chunk tmpCk2; chunk * pCk;
      MyFunction ();
      pCk = new chunk;  // 构造函数被调用
      ....
      delete pCk ; // 析构函数被调用
} //main函数退出时 tmpCk2消亡，调用析构函数。
//程序结束前， ck ,ck1, ck2 的析构函数被调用。
```

```
class Demo {
        int id;
public:
        Demo(int i)
         {
                id = i;
                printf( "id=%d, Construct\n", id);
        }
        ~Demo()
         {
                  printf( "id=%d, Destruct\n", id);
        }
};
```

```
Demo d1(1);
void fun(){
        static Demo d2(2);
        Demo d3(3);
        printf( "fun \n");
}
int main (){
        Demo d4(4);
        printf( "main \n");
        {    Demo d5(5);    }
        fun();
        printf( "endmain \n");
        return 0;
}
```

输出：

```
id=1,Construct
id=4,Construct
main
id=5,Construct
id=5,Destruct
id=2,Construct
id=3,Construct
fun
id=3,Destruct
endmain
id=4,Destruct
id=2,Destruct
id=1,Destruct
```

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;
}
```

运行结果：

```
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
```

静态成员变量和静态成员函数

- 静态成员：在说明前面加了 static关键字的成员。
  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。
- 如果是public的话，那么静态成员在没有对象生成的时候也能直接访问。

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

静态成员

- 访问静态成员

（1）可以通过 ：

类名::成员名的方式: CRectangle::PrintTotal();

（2）也可以和普通成员一样采取。

对象名.成员名　　r.PrintTotal();

指针->成员名　　CRectangle * p = &r;

p->PrintTotal();

(3)引用.成员名　　CRectangle & ref = r;

int n = ref.nTotalNumber;

上面这三种方式，效果和类名::成员名 没区别，静态成员变量不会属于某个特定对象，静态成员函数不会作用于某个特定对象。

- 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

- 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。

静态成员变量和静态成员函数

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，
当然可以用全局变量来记录这两个值，但是用静态成员封装进类中，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
```

➢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
}
```

因为：

```
CRectangle r;
r.PrintTotal(); // 解释得通
CRetangle::PrintTotal(); //解释不通，w 到底是属于那个对象的？
```

复制构造函数和静态变量的一个例子

```
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
```

- 上面CRectangle 类的不足之处：在使用CRectangle类的过程中，有时会调用复制构造函数生成临时的隐藏的CRectangle对象(如，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)。
- 那么，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。
- 因此，要为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;
}
```

对象的赋值

- 缺省的情况下，对象间的赋值执行的是按位拷贝。

```
Complex c1,c2;
c1 = c2; // c1的所有成员变量变成和c2一样
```

- 如果赋值运算符被重载，那么情况就会不一样了，以后会介绍。

构造函数和析构函数小结

- 构造函数
  - 定义
  - 调用方式(隐式)
  - 复制构造函数
- 析构函数
  - 定义
  - 调用时机

- 各种构造函数和析构函数的调用时机。

构造函数（调用方式）

- 定义对象变量时：

```
Complex c1, c2(2),c3(3,5);
```

- 创建新对象变量时：

```
Complex * pc1 = new Complex ;
Complex * pc2 = new Complex(3,4);
```

- 创建对象数组时：

```
Test  array1[3] = { 1, Test(1,2) };
Test  * pArray[3] = { new Test( 4), new Test(1,2) };
```

构造函数（复制构造函数）

- 特殊构造函数,只有一个参数,类型为本类的引用;
- 如果没有定义,生成缺省的复制构造函数;
- 如果定义,没有缺省复制构造函数;
- 与前面说的构造函数无关;

  Complex c2(c1);　　Complex c2 = c1;
- 参数传递时,复制参数;
- 函数返回时复制返回值。

析构函数（定义）

- 只有一个；
- 没有参数和返回值；
- 如果不定义，自动生成，什么也不做；
- 如果定义，没有缺省的；
- 完成对象消亡前的收尾工作；
- ~类名(){ … }

析造函数（调用时机）

- 变量消亡；
  - 出作用域
  - delete
- 函数返回值返回后；

各种构造函数和析构函数的调用时机

- 构造函数
  - 全局变量: 程序运行前;
  - main中变量: main开始前;
  - 函数中静态变量: 函数开始前;
  - 函数参数:函数开始前;
  - 函数中变量:函数执行时;
  - 函数返回值： 函数返回前;

各种构造函数和析构函数的调用时机

- 析构函数
  - 全局变量: 程序结束前;
  - main中变量: main结束前;
  - 函数中静态变量: 程序结束前;
  - 函数参数:函数结束前;
  - 函数中变量:函数结束前;
  - 函数返回值： 函数返回值被使用后.

构造函数和析构函数在不同编译器中的表现

有些编译器有bug，或者加了代码优化措施，会造成构造函数或析构函数调用的情况和前面所说的不一致。但前面所说是C++的标准。

考试题一定是Dev C++和VS 2008结果完全一致的

构造函数和析构函数在不同编译器中的表现

```
#include <iostream>
using namespace std;
class A {
    public:
        int x;
        A(int x_):x(x_) {
                cout << x << " constructor called" << endl;
        }
        ~A() { cout << x << " destructor called" << endl; }
};
A c(2);
int main( ) {
     return 0;
}
```

构造函数和析构函数在不同编译器中的表现

上面的程序，在vc6输出：

2 constructor called

在DEV和VS2008都输出：

2 constructor called
2 destructor called

结论：有时全局对象不析构，是vc6的一个bug

构造函数和析构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 b(10);
A f( )
{
    return b;
}
int main( ) {
       A a(1);
       a = f();
       return 0;
}
```

输出结果是:

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

构造函数和析构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。VC6和VS2008无此问题
```

在dev中，将全局变量b变为局部变量后，输出结果是：

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

const 的用法

1)定义常量

```
const int MAX_VAL = 23;
const string SCHOOL_NAME = “Peking University”;
```

2)用在指针定义上，则不可通过该指针修改其指向的地方的内容

```
int n,m;
const int * p = & n;
* p = 5; //编译出错
n = 4;  //ok
p = &m; //ok
```

注意：不能把常量指针赋值给非常量指针，反过来可以

```
const int * p1; int * p2;
p1 = p2;  //ok
p2 = p1;  //error
p2 = (int * ) p1; //ok
```

const 的用法

✓ 如果不希望在函数内部不小心写了改变参数指针所指地方的内容的语句，那么就可以使用常数指针参数。

```
void MyPrintf( const char * p )
{
   strcpy( p,"this"); //编译出错
   printf("%s",p); //ok
}
```

3）用在引用上

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

常量对象和常量方法

➢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 Sample {
        private :
                int value;
        public:
                void SetValue() {
                }
};
const Sample Obj;  // 常量对象
Obj.SetValue ();   //错误 。常量对象只能使用构造函数、析构函数和 有 const 说明的函数(常量方法）
```

➢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。

➢常量成员函数内部不能改变属性的值，也不能调用非常量成员函数。

```
class Sample {
        private :
                int value;
        public:
                void func() { };
                void SetValue()  const {
                        value = 0; // wrong
                        func(); //wrong
                }
};
const Sample Obj;
Obj.SetValue (); //常量对象上可以使用常量成员函数
```

➢在定义常量成员函数和声明常量成员函数时都应该使用const 关键字。

```
class Sample {
        private :
                int value;
        public:
                void SetValue()  const;
};
void Sample::SetValue()  const  {    //此处不使用const会
                                     //导致编译出错
        cout << value;
}
```

const 的用法

➢如果觉得传递一个对象效率太低（导致复制构造函数调用，还费时间）又不想传递指针，又要确保实际参数的值不能在函数中被修改，那么可以使用：

const T & 类型的参数。

```
void PrintfObj( const Sample & o )
{
  o.func(); //error
  o.SetValue(); //ok
  Sample & r = (Sample &) o;//必须强制类型转换
  r.func();  //ok
}
```

常量成员函数的重载

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

```
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;
}
```

输出： 1， 2

成员对象和封闭类

➢有成员对象的类叫 封闭（enclosing)类。

```
#include <iostream>
using namespace std;
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
        private:
                int price;  //价格
                CTyre tyre;
                CEngine engine;
        public:
                CCar(int p,int tr,int tw );
};
CCar::CCar(int p,int tr,int tw):price(p),tyre(tr,tw)
{
};
```

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

✓封闭类对象生成时，先执行所有对象成员的构造函数，然后才执行封闭类的构造函数。

✓对象成员的构造函数调用次序和对象成员在类中的说明次序一致，与它们在成员初始化列表中出现的次序无关。

✓当封闭类的对象消亡时，先执行封闭类的析构函数，然后再执行成员对象的析构函数。次序和构造函数的调用次序相反。

封闭类例子程序

```
#include <iostream>
using namespace std;
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

- 成员对象初始化列表中的参数可以是任意复杂的表达式，可以包括函数，变量，只要表达式中的函数或变量有定义就行。

其他特殊成员：const成员和引用成员

➢初始化const 成员和引用成员时，必须在成员初始化列表中进行。

```
class example {
        private :
                const int num;
                int & ret;
                int value;
        public:
                example( int n,int f) :num(n),ret(f),value(4)
                {       }
};
```

友元（friends）

分为友元函数和友元类两种

1) 友元函数：

. 可以将一个全局函数说明为一个类的友元。

. 可以将一个类的成员函数(包括构造、析构函数)说明为另一个类的友元。

```
class CCar ; //提前声明 CCar类，以便后面的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);
    friend void CDriver::ModifyCar(CCar * pCar);
};
```

```
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;     }
```

2)友元类： 如果A是B的友元类，那么A的成员函数可以访问B的私有成员。

```
class CCar
{
        private:
                int price;
        friend class CDriver;
};
class CDriver
{
        public:
          CCar myCar;
           void ModifyCar()  //改装汽车
           {      myCar.price += 1000       }
};
```

✓友元类之间的关系不能传递，不能继承。

this 指针

➢每个对象的 this 指针指向它自己，但是this指针不是成员变量。

```
class Complex {
        float real, imag;
        public:
                Complex * ReturnAddress ( ) {
                        return this;
                }//  c.ReturnAddress ()等效于 & c
                float ReturnReal() {
                        return this -> real;//等效于return real

                }
};
```

➢ * this 代表对象自身。

```
#include <iostream>
using namespace std;
class Complex {
        public:
        double real, imag;
    Complex(double r,double i):real(r),imag(i) { }
        Complex AddOne()
        {
            this->real ++;
            return * this;
        }
};
```

```
int main()
{
        Complex c1(1,1),c2(0,0);
        c2 = c1.AddOne();
  cout << c2.real << "," << c2.imag << endl;  //输出 2,1
    return 0;
}
```

非静态成员函数实际上参数的个数，比写出来的多一个，多出来的参数，就是this指针。

```
#include <iostream>
using namespace std;
class A
{
        int i;
  public:
         void Hello() { cout << "hello" << endl; }
};
int main()
{
        A * p = NULL;
        p->Hello();
}
```

输出结果：

hello