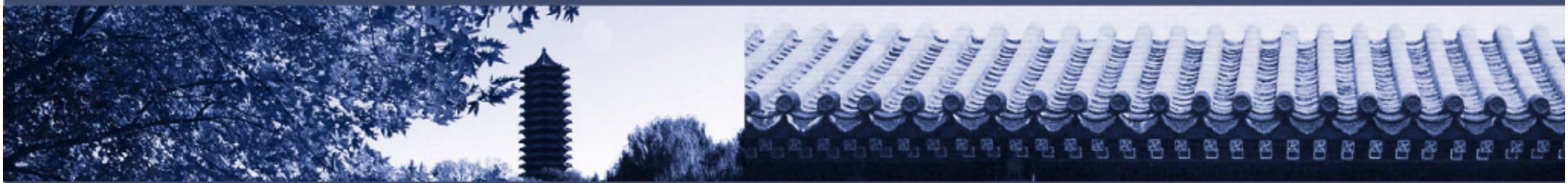
2018年春

程序设计实习(I): C++程序设计

第三讲 类和对象(2)

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

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

想到答案邮件TA 569155712@qq.com

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) };
    // 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;
}
```


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

输出结果:
Copy constructor called
4

经过优化的编译器可能导致结果不同

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

常量引用参数的使用

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

- 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
}
```


类型转换构造函数

- 定义转换构造函数
  - 目的: 实现类型的自动转换
  - 只有一个参数
  - 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**

- 需要时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)

```
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

构造函数(定义)

- 一种类的**成员函数**
- 用来**初始化**对象
- 名字与类名相同
- 可以有参数
- 可以有一个或多个
- 如果不定义, 自动生成, 什么也不做
- 如果定义, 无缺省

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<u>本类的引用</u>
- 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
- 三种调用情况:
  - 对象初始化: **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

内联函数

- 函数**调用本身**是有时间开销的
- 如果函数本身只有几条语句，执行非常快

Vs. 但是函数被反复执行很多次，调用函数所产生的这个开销就会显得比较大

内联函数

- 定义函数时，在返回值类型前面加 **inline** 关键字，可以使得函数成为内联函数
- 编译器处理内联函数的调用语句时 → 将整个函数的代码插入到调用语句处，而不会生出调用函数的语句

```
inline int Max(int a, int b)
{
    if( a > b) return a;
    return b;
}
```


内联成员函数

- 在成员函数前面加上 **inline** 关键字后，成员函数就成为内联成员函数
- 将整个函数体写在类定义内部，函数也会成为内联成员函数

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

func1和func2都是内联成员函数

函数重载

- 一个或多个函数，名字相同，然而参数个数或参数类型互不相同，这叫做 函数的重载

  如：

  ```
  int Max(double f1, double f2) {  }
  int Max(int n1, int n2) {  }
  int Max(int n1, int n2, int n3) {  }
  ```

- 函数重载使得函数命名变得简单
- 编译器根据调用语句中的实参判断应该调用哪个函数
- 类的成员函数也可以重载

函数的缺省参数

- C++中，写函数的时候可以让参数有缺省值，
- 则调用函数的时候，若不写参数，参数就是缺省值

```
void func(int x1 = 2, int x2 = 3) { }

func( ) ;   //等效于 func(2, 3)
func(8) ;   //等效于 func(8, 3)
func( , 8) ; //不行
```

- 函数参数可缺省的目的在于提高程序的可扩充性
- 如果某个写好的函数要添加新的参数，而原先那些调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用缺省参数

函数的缺省参数

- 任何有定义的表达式都可以成为函数的缺省参数:

```
int Max( int m, int n);
int a, b;
void Function2 ( int x, int y = Max(a,b), int z = a*b)  {
                    ……
}
Function2(4);
// 正确, 等效于 Function(4, Max (a, b), a*b);
Function2(4, 9); // 正确，等效于 Function(4, 9, a*b);
Function2(4, 2, 3); //当然正确
Function2(4,  , 3); //错误！这样的写法不允许，省略的参数一定是最右边连续的几个
```


成员函数的重载及参数缺省

- 成员函数也可以重载 (普通函数也可以)
- 成员函数和构造函数可以带缺省参数 (普通函数也可以)

```
#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, B;
        A.init(5);
        A.valueX(5);
        cout << A.valueX();
        return 0;
}
```


- 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
```


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
  - 一个类最多只有一个析构函数
- 析构函数对象消亡时 → 自动被调用
  - 定义析构函数 → 对象消亡前做善后工作

    e.g. 释放分配的空间
- 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
}
```


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
```

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？

```
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;
}
```


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;
}
```


析构函数

- 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( ){ … }