第一节 类和对象的基本概念（2）

1.类的成员函数和类的定义分开写

例如我们在类里面定义了一个类别还有函数，那么类的定义可以写在类的外面，以例子给出格式（仍以上一节 CRectangle 类为例）：

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

一定要通过对象或对象的指针或对象的引用才能引用。

2.类成员的可访问范围

（1）private：私有成员，只能在成员函数内访问；

（2）public：公有成员，可以在任何地方访问；

（3）protected：保护成员。

具体用法如下：

```
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
}
```

注意：如果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。

例如：

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

3.类成员的可访问范围

在类的成员函数内部，能够访问：

（1）当前对象的全部属性、函数；

（2）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，只能访问该类的公有成员。

设置私有成员的机制，称为“**隐藏**”。其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。

4.成员函数的重载及参数缺省

成员函数重载，成员函数也可以带缺省参数。

第二节 构造函数

1.基本概念

构造函数是成员函数的一种它的名字与类名相同，可以有参数，但**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。如果定义类时没有写构造函数，则编译器生成一个默认的无参数的构造函数，默认的构造函数不做任何操作。如果我们定义了构造函数，那么系统就不生成构造函数。

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。

打个比方，我们新建成员变量是建房子，那么构造函数只能做装修房子，不能做建房子的操作。

为什么需要构造函数呢？

（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。

例如：

```
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，生成一个可变的
```

自己写出构造函数，例如：

```
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;//错误，缺少构造函数的参数
Complex *pc = new Complex;//错误，缺少构造函数的参数
Complex c1(2);//Ok，第二个参数使用缺省参数 i=0
Complex c1(2,4),c2(3,5);//OK
```

```
Complex *pc = new Complex(3,4);//OK
```

多个构造函数举例：

```
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 d1(3),d2(1,0),d3(d1,d2);
//d1 = {3,0}, d2 = {1,0}, d3 = {4,0}
```

注意：**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。**

2.构造函数在数组中的使用

以例子来讲解。例 1 如下：

```
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];//由于数组为空，当作一个无参来初始化。
    //然后数组里面是两个空的元素，实际上相当于初始化然后数组里面是两个空的元素，
    //实际上相当于初始化了两边。输出结果是两遍 Constructor 1 Called
```

```
    cout <<"step1"<<endl;//输出 step1
    CSample array2[2] = {4,5};//此时是有参构造，相当于创建两个对象，一个对象里面 x 为 4，一个为 5
    //所以输出是两遍 Constructor 2 Called
    cout <<"step2"<<endl; //输出 step2
    CSample array3[2] = {3};//此时是有参构造和无参构造结合，相当于创建另两个对象，一个里面 x=3
    //另一个无参数，输出一遍 Constructor 2 Called，Constructor 1 Called
    cout <<"step3"<<endl;//step3
    CSample *array4 = new CSample[2];//与 array1[2]同理
    delete []array4;//回收
    return 0;
```

例 2：

```
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
```

注意，**指针不会被初始化**。例如：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() };
```

答案是 2 个。因为 new A()会生成对象，NULL 为空，所以它还是一个指针，没有指向一个对象的地址，故无法生成对象，第四个元素同理。因为只有两个 new A()，故有 2 个对象。

第三节 复制构造函数

1.基本概念

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，**参数地方必须为引用！**。后者能以常量对象作为参数。

如果没有定义复制构造参数，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对象间的复制功能。例如：

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

```
};
Complex c1;//调用缺省无参构造参数
Complex c2(c1);//调用缺省的复制构造函数，将 c2 初始化成和 c1 一样
```

如果自己写，例如：

```
class Complex{
    public:
        double real,imag;
    Complex(){}
    Complex(const Complex &c){
        real = c.real;
        imag = c.imag;
        cout<<"Copy Constructor called";
    }

};
Complex c1;//调用缺省无参构造参数
Complex c2(c1);//调用编写的复制构造函数，将 c2 初始化成和 c1 一样
```

2.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**。例如：

```
Complex c2(c1);
Complex c2 = c1; //初始化语句，非赋值语句
```

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;
        }
};
void Func(A a1){}
int main(){
    A a2;
    Func(a2);//参数为类 A 的对象
    return 0;
}
```

输出：Copy constructor called

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;
    }
};

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

输出结果：

```
Copy constructor called
4
```

复制构造函数的参数是 b。Func().v 是 b.v 的一个复制品。

注意：**对象间赋值并不导致复制构造函数被调用**。

例如：

```
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;
}
```

在这里 c2=c1 不是一个初始化语句，而是一个赋值语句，所以并不导致复制构造函数被调用。因此输出的结果 c2.n = 5，c3.n = 10（因为复制构造函数所计算值是 2*c.n）。

3.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，实际上形参成为了实参的应用，成为了一回事（原来的时候在调用函数时，形参实际上是复制了一份实参，这个太浪费时间）。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
}
```

第四节 类型转换构造函数和析构函数

1.什么是类型转换构造函数

它定义转换构造函数的目的是实现类型的自动转换。如果只有一个参数，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)。

实例如下：

```
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; //它调用 Complex( int i)
    c1 = 9; // 9 被自动转换成一个临时 Complex 对象
    cout << c1.real << "," << c1.imag << endl;
    return 0;
}
```

解析：c1=9 这里经历的这样一个过程：首先新建一个临时的 Complex 对象，我假定为 m，那么这个 m 进行初始化（即相当于执行 Complex a(9)，或者 Complex a=9），然后调用 Complex(int i)这个类型转换构造函数，构造一个 m 里面的 real=9，imag=0，然后把 m 复制给 c1。

习题：类 A 定义如下：

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

```
};
```

下面段程序不会引发类型转换构造函数被调用？

A) A a1(4);

B) A a2 = 4;

C) A a3; a3 = 9;

D) A a1,a2; a1 = a2;

答案：D。解析如下：A、B、C 都显然调用了 A(int i)，而 D 里面 a1=a2 是赋值语句。

2.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;
}
```

对象数组声明期结束时，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;
}
```

输出结果：

```
End Min
destructor called
destructor called
```

数组的两个元素都用完了，所以构造了析构函数两遍。

3.析构函数和运算符 delete

（1）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
```

前面说过：**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)**。

4.析构函数在对象作为函数返回值返回后被调用

例如：

```
class CMyclass {
    public:
    ~CMyclass() { cout << "destructor" << endl; }
};
CMyclass obj;
CMyclass fun(CMyclass sobj ) {//次数对象作为了形参，调用复制构造函数
//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
}
```

第五节 构造函数析构函数调用时机

构造函数和析构函数什么时候被调用呢？见课本 P190 的这个例子：

```
class Demo {
    int id;
    public:
        Demo(int i) {//这个也可以看做是类型转换构造函数
            id = i;
            cout << "id=" << id << " constructed" << endl;
        }
        ~Demo() { cout << "id=" << id << " destructed" << endl; }
};

Demo d1(1);//全局对象，在 main 运行前先运行构造函数，因此输出 id=1 constructed
void Func() {
    static Demo d2(2);//静态局部变量
    Demo d3(3);
    cout << "func" << endl;
}
```

```
int main () {
    Demo d4(4);//先运行构造函数，输出 id=4 constructed
    d4 = 6;//运行类型转换构造函数，输出 id=6 constructed
//临时对象调用完之后，就会消亡。输出 id=6 destructed。但是消亡的是临时对象
    cout << "main" << endl;
    { Demo d5(5);
    }//局部对象。输出 id=5 constructed。因为是局部对象，花括号结束后就消亡。
//因此输出 id=5 destructed
    Func();//调用 Func 函数，输出 id=2 constructed 和 id=3 constructed，然后输出 func
//局部静态变量在函数结束的时候仍然存在，仅此只有 d3(3)消亡，输出
id=3 destructed
    cout << "main ends" << endl;
    return 0;//整个程序结束后，局部变量消亡，输出 id=6 destructed。main 的变量 d4 和全
局变量 d1 消亡，输出 id=2 destructed 和 id=1 destructed
}
```

输出结果为：

```
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
```

形象举例，析构函数在拆房子过程中不是负责拆房子，而是在拆房子之前在房子里面做一些善后工作，比如说把东西都搬走。

习题：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;
}
```

答案：3 次。第一句话 new 出来 2 个，第二句话 new 出来 1 个。**new 出来的东西，如果你不 delete 它，主程序结束它也不会消亡**。所以调用西沟函数是 A a 在最后调用 1 次，delete 调用 2 次，一共 3 次。

复制构造函数在不同编译器中的表现不一定相同！

例如下：

```
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;
}
```

正常情况下执行过程如下，以下为在 Visula Studio 中输出情况：

首先执行 main 函数里面的第一句话，输出 **1 constructor called**。然后执行第二句话，调用函数 f()，生成 b，调用构造函数使 m=10，因此输出 **10 constructor called**，b 返回后，任务结束，调用析构函数，输出 **10 destructor called**。接着 f 的返回值是一个临时对象，需要进行初始化，所以调用 A(const A & a)，首先执行加法操作，临时对象的值变为 12，然后输出 **copy called**。将临时对象赋值给 a 后，临时对象消亡，调用析构函数，输出 **12 destructor called**，然后主程序执行完毕，a 也消亡，调用析构函数，输出 **12 destructor called**。

而在 dev C++则输出：

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

说明 dev 出于优化目的并未生成返回值临时对象。