2018年春

程序设计实习(I): C++程序设计

第四讲 类和对象(3)

刘家瑛

liujiaying@pku.edu.cn

上节课知识点复习

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();

上节课知识点复习

- 以下对类A的定义，哪个是正确的？

A)
```
class A {
    private: int v;
    public: void Func() { }
}
```

B)
```
class A {
    int v;
    A * next;
    void Func() { }
};
```

C)
```
class A {
        int v;
    public:
        void Func();
};
A::void Func() { }
```

D)
```
class A {
        int v;
    public:
        A next;
        void Func() { }
};
```

上节课知识点复习

- 以下程序哪个不正确?

A)
```
int main() {
    class A {
    public:   int v;  A * p;  };
    A a; a.p = new A; delete a.p;
    return 0;
}
```

B)
```
int main()  {
    class A {
    public:
        int v;  A * p;  };
    A a;  a.p = & a;  return 0;
}
```

C)
```
int main()  {
    class A {
    public:   int v;  };
    A * p = new A;
    p->v = 4;  delete p;
    return 0;
}
```

D)
```
int main(){
    class A { int v; };
    A a; a.v = 3;  return 0;
}
```

上节课知识点复习

- 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);

上节课知识点复习

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;

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

Vs. 静态成员变量一共就一份，为所有对象共享

- 如果是public, 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

访问静态成员方式:

(1) 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();
- 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**

  - 将与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

- 设计一个需随时知道矩形总数和总面积的图形处理程序
  - 可以用全局变量来记录这两个值

Vs. 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;
// 必须在定义类定义的外面专门对静态成员变量进行声明
// 同行可以初始化; 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

```
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;
}
```


- 现有设计的CRectangle 类的<u>不足之处</u>:
  - 在使用中,如果调用复制构造函数

  → 临时隐藏的CRectangle对象
    - 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
  - 则临时对象在消亡时 →调用析构函数
    - 减少nTotalNumber 和 nTotalArea的值
    - 但临时对象生成时，却没有增加 nTotalNumber 和 nTotalArea的值

- 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;
}
```


const 的用法

- 定义常量

```
const int MAX_VAL = 23;
const string SCHOOL_NAME = "Peking University";
```

- 对指针定义，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

- 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 → 使用常数指针参数

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

- 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 GetValue() {
    }
};
const Sample Obj;  // 常量对象
Obj.GetValue();    //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非静态属性的值, 也不能调用同类的非常量成员函数 (静态成员函数除外)

```
class Sample {
        public: int value;
                void GetValue()  const;
                void func() { };
};
void Sample::GetValue() const {
        value = 0; // wrong
        func(); //wrong
}
int main(){
        const Sample o;
        o.GetValue();  //常量对象上可以执行常量成员函数
        return 0;
} //Visual Studio 2010中没有问题, 在Dev C++中, 要为Sample类编写无参构造函数才可以
```

- 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 GetValue()  const;
};
void Sample::GetValue()  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.GetValue(); //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;
}
```

输出结果:1, 2

成员对象和封闭类

- 成员对象: 一个类的成员变量是另一个类的对象
- 有成员对象的类叫 封闭类 (enclosing)

```
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 w): price(p), tyre(tr, w)
{
};
int main()
{
        CCar car(20000, 17, 225);
        return 0;
}
```


- 如果 CCar类不定义构造函数，则：

CCar car;　　// compile error

因为编译器不明白 **car.tyre**该如何初始化;

car.engine 的初始化没问题，用默认构造函数即可

- 任何生成封闭类对象的语句，都要让编译器明白

→ 对象中的成员对象，是如何初始化的

- 具体的做法就是

通过**封闭类的构造函数的初始化列表**

封闭类构造函数的初始化列表

- 定义封闭类的构造函数时，添加初始化列表：

类名::构造函数(参数表):成员变量1(参数表)，成员变量2(参数表)，…

{

…

}

- 成员对象初始化列表中的参数
  - 任意复杂的表达式
  - 函数 / 变量/ 表达式中的函数，变量有定义

30

- 封闭类对象生成时
  - 先执行所有成员对象的构造函数
  - 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

```
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; }
};
```


封闭类例子程序

```
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
```


封闭类的复制构造函数

- 封闭类的对象，如果是用默认复制构造函数初始化

→ 它里面包含的成员对象，也会用复制构造函数初始化

```
class A{
    public:
        A() { cout << “default” << endl; }
        A(A & a) { cout << "copy" << endl;}
};
class B { A a; };
int main(){
    B b1,  b2(b1);
    return 0;
}
```

输出结果：
default
copy

- 说明**b2.a**是用类A的复制构造函数初始化
- 而调用复制构造函数时的实参就是b1.a

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

```
int f;
class CDemo {
        private :
                const int num; //常量型成员变量
                int & ref;      //引用型成员变量
                int value;
        public:
                CDemo(int n): num(n),  ref(f),  value(4) {  }
};
int main(){
        cout << sizeof(CDemo) << endl;
        return 0;
} //输出结果是：12
```


友元 (friends)

- 友元分为友元函数和友元类两种
  - 友元函数: 一个类的友元函数可以访问该类的私有成员

```
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); //声明友元
    friend void CDriver::ModifyCar(CCar * pCar);        //声明友元
};
```


```
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;
}
```


- 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 B {
    public:
        void function();
};

class A {
        friend void B::function();
};
```


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;    //声明CDriver为友元类
};
class CDriver{
       public:
             CCar myCar;
             void ModifyCar(){  //改装汽车
                   myCar.price += 1000;  //因CDriver是CCar的友元类,
                                         //故此处可以访问其私有成员
             }
};
int main(){    return 0;    }
```

- 友元类之间的关系不能传递, 不能继承

this 指针

- C++程序到C程序的翻译:

```
class CCar{
    public:
        int price;
        void SetPrice(int p);
};
void CCar::SetPrice(int p){
    price = p;
}
int main(){
    CCar car;
    car.SetPrice(20000);
    return 0;
}
```

```
struct CCar{
    int price;
};
void SetPrice(CCar * this,  int p){
    this->price = p;
}
int main(){
    struct CCar car;
    SetPrice(& car,  20000);
    return 0;
}
```


this 指针

- 非静态成员函数中可以直接使用 this 来代表指向该函数作用的对象的指针

```
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;
}
```

Big Brother Is Watching You!