2019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

课前多叽歪

本周末上机安排

- **时间: 周日 下午15:00 – 17:00**
- **地点: 院机房1236**
- 分组名单：教学网查名单
- 安排：面查 + 答疑 + 针对性讲解
- 第一次：**登记联系方式 / 助教辅导课**
- 注意错峰 / 关于时间调整 / 关于请假

课程回顾

- 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

课程回顾

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

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

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

[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

- **类型转换构造函数**
  - **目的**: 实现类型的自动转换
  - 只有**一个参数**
  - **又不是复制构造函数**的构造函数

→ 该构造函数起到类型自动转换的作用

- 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

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

函数参数的默认值

- C++中，声明函数时，可以为函数参数指定默认值，
- 则调用函数的时候，若不写参数，参数就是默认值

```
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，省略的参数一定是最右边连续的几个
```

- 函数默认参数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，就可以使用默认参数

函数参数的默认值

- 任何有定义的表达式都可以成为函数参数的默认值:

```
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默认

- 成员函数也可以重载 (普通函数也可以)
- 成员函数/构造函数可以带默认值参数 (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) 被用过后,
    return 0;        //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) 被用过后,
    return 0;        //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;
}
```


```
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;
}
```


```
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

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

54

- **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;
}
```