第三周

第一节 this 指针

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

图 1 从 C++到 C

C++中的 class 在 C 语言中并不存在，因此可以用 struct（结构体）来代替，成员变量也可以直接用域来代替，但是类中的成员函数却无法在 C 语言中直接表示，所以只能把 SetPrice 设置为全局函数。由于是全局函数，为了能让其使用结构体内的数据，故在 SetPrice 中增加了一个参数 struct CCar *this，这样就可以在函数体中把 p 的值传递给结构体中的 price，实现了跟成员函数一样的功能，同理，在 main()中 SetPrice 函数执行也得把 car 的地址传递过去。

故，this 的作用就是**指向成员函数所作用的对象**。

2.this 指针作用

（1）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;
  void Print() {  cout << real << "," << imag ;  }
  Complex(double r,double i):real(r),imag(i)
  {       }
  Complex AddOne()       {
      this->real ++;  //等价于 real ++;
      this->Print();    //等价于 Print
      return * this;
  }
};
```

```
int main()  {
    Complex c1(1,1),c2(0,0);
    c2 = c1.AddOne();
    return 0;
} //输出 2,1
```

题 2 this 指针作用

在 main()函数中，首先对对象 c1 执行了 AddOne()函数，然后开始执行 AddOne()，对 c1 中的 real 加 1，然后执行对 c1 的 Print()函数，这之后，返回一个 this 指针所指向对 Complex 对象（即 c1）给 c2。

对 this 指针再加深一个理解，如下例：

```
class A
{
    int i;
```

```
    public:
        void Hello()
        {cout<<""Hello"<<endl;}
};
int main()
{
    A *p=NULL;
    p->Hello();
}
```

系统会报错吗？答案是否定的，并且输出 Hello。原因在于，虽然我们看着这个*p 指向的空，但是实际上，由于 C++在转换成 C 语言的时候，成员函数会被转换成全局函数并增加一个 this 指针的参数，即变为了

void Hello(A *this){cout <<”Hello”<<endl;

主函数中的 p->Hello();也变成了 Hello(p);

因此就算 p 为空指针，在这里传递给全局函数后并没有用到 p 做任何输出，所以直接输出了 Hello。**但如果我们在这个函数的输出时用到了成员变量 i，那么成员函数在变换到全局函数的时候再来执行就用到了这个 this 所指向的对象，而此时 this 指向的 p 为空，所以报错。**

静态成员函数不能使用 this 指针，因为静态成员函数并不具体作用于某个对象，因此静态成员函数的真是的参数的个数，就是程序中写出的参数个数！

第二节 静态成员变量

1.静态成员基本概念

在定义前面加了 static 关键字的成员，称为静态成员。静态成员包括静态成员变量和静态成员函数两种。

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**。

sizeof 运算符在计算类的大小的时候不会计算静态成员变量。

普通成员函数必须具体作用于某个对象，而静态成员函数并**不具体作用于某个对象**。

因此静态成员**不需要通过对象**就能访问。

静态成员变量本质上是全局变量!!!

静态成员函数本质上是全局函数!!!

设置静态成员的机制是将和某些类紧密相关的全局变量和函数写到类里面，看上去像一根整体，容易维护和理解。

2.如何访问静态成员

（1）类名::成员名

例如 CRectangle::PrintTotal();

（2）对象名.成员名

例如 CRectangle r; r.PrintTotal();

（3）指针->成员名

CRectangle *p=&r;p->PrintTotal();

（4）引用.成员名

CRectangle &ref =r; int n = ref.nTotalNumber;

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。

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。

如下所示的写法，存在缺陷。

```
CRectangle::CRectangle(int w_,h_) {
    w = w_;
    h = h_;
    nTotalNumber++;
    nTotalArea += w* h;
}
CRectangle::~CRectangle(){
    nTotalNumber--;
    nTotalArea -= w*h;
}
void CRectangle::PrintTotal(){
    cout<<nTotalNumber<<","<<nTotalArea<<endl;
}
```

在使用 CRectangle 类时，有时会调用复制构造函数生成临时的隐藏的 CRectangle 对象，比如：①我们调用了一个以 CRectangle 类对象作为参数的函数；②调用一个以 CRectangle 类对象作为返回值的函数。这个时候就如前面介绍复制构造函数时所说，会产生一个隐藏的临时对象（此函数没写复制构造函数，因此就算直接复制值过去，并没有执行 Crectangle（int w_,int h_））， 而这个临时对象在消亡的时候会调用析构函数， 减少了 nTotalNumber 和 nTotalArea 的值，而它们这些对象在生成时却没有增加这两个值，由于这两个值是静态成员变量，所以实际上这两个值是减少了。

解决方案很显然就算写一个复制构造函数：

```
CRectangle::CRectangle(CRectangle &r)
{
    w = r.w;h=r.h;
    nTotalNumber ++;
    nTotalArea += w*h;
}
```

第三节 成员对象和封闭类

1.基本概念

有成员对象的类交封闭（enclosing）类。

有一个类的成员是其它类的对象，被称为成员对象。

如下例程：

```
class CTyre
{
    private:
        int radius;
        int width;
    public:
        CTyre(int r int w):radius(r),width(w){}//初始化列表，可以为每一个成员变量提供一
```

个初始值。
};
class CEngine{};
class CCar{//封闭类
    private:
        int price;
        Ctyre tyre;//成员对象
        CEngine engine;//成员对象
    public:
        CCar(int p, int tr,int tw);
};
CCar::CCar(int p,int tr,int w):price(p),tyre(tr,w)//engine 使用无参构造函数构造
{};
int main()
{
    CCar car(20000,17,225);
    return 0;
}

在这个例子，如果 CCar 类不定义构造函数，则 CCar car;这个语句会报错，因为编译器不明白 car.tyre 该如何初始化。

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。

具体做法是：**通过封闭类的构造函数的初始化列表进行**。

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。

2.封闭类构造函数和析构函数的执行顺序

（1）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；

（2）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；

（3）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。

3.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的，那么它里面包含的成员对象，也会用复制构造函数初始化。

例如：

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： default copy

说明 b2.a 是用类 A 的复制构造函数初始化的，而且调用复制构造函数的实参就是 b1.a。

第四节 友元（friends）

友元分为友元函数和友元类两种。

1.友元函数

一个类的友元函数可以访问该类的私有成员（这说明友元函数不是这个类的成员函数）

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();
};
```

这就代表 function()可以访问 A 里面的私有成员了。

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(){
            myCar.price += 1000;
//因为 CDriver 是 CCar 的友元类，故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！

第三节 常量对象、常量成员函数和常引用

1.常量成员函数

如果不希望某个对象值被改变，则定义该对象的时候可以在前面加 const 关键词。例如：

```
class Sample{
    private:
        int value;
    public:
    Sample(){}
    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、析构函数和右 const 说明的函数（const 方法）
```

在类的成员函数说明后面加 const 关键字，则该成员函数为常量成员函数。常量成员函数执行期间不应修改其作用的对象。因此，在常量成员函数中不能修改成员变量的值（静态成员变量除外），也不能调用同类的非常量成员函数（静态成员函数除外）。

如下：

```
class Sample{
    public:
    int value;
    void func(){};
    Sample(){}
    void GetValue() const;
};
void Sample::GetValue() const{
    value = 0;//wrong
    func();//wrong
}
int main(){
    const Sample o;
    o.value = 100;//err 常量对象不可被修改
    o.func();//常量对象上面不能执行非常量成员函数
    o.GetValue();//OK 常量对象可以执行常量成员函数
    return 0;
}
```

需要注意，在 Dev C++中，要为 Sameple 类编写无参构造函数才可以，在 VS 中不需要。

2.常量成员函数的重载

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

3.常引用

引用前面加 const 称为常引用。不能通过常引用修改其引用的变量。

可以用对象的引用作为参数，如：

```
class Sample{
};
```

```
void PrintfObj(Sample & o){
}
```

对象引用作为函数的参数有一定风险性，若函数中不小心修改了形参 o，则实参也跟着变。为了避免这个问题，我们可以用对象的常引用作为参数，这样就能确保不会出现无意中更改 o 值的语句了。

```
class Sample{
};
void PrintfObj(const Sample & o){
}
```