2023年春季学期

软件设计实践

C++面向对象程序设计(2)

类与对象进阶

谢 涛　　马 郓

本讲概览

- this指针
- 成员对象
  - 成员初始化列表
- 静态成员
- 只读成员
  - 只读对象
  - 只读成员函数
  - 只读成员变量
- 友元

this指针

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r, double i) {
        real = r;
        imag = i;
    }
    void setReal(int r) {
        real = r;
    }
};
```

在成员函数中可以使用成员变量、调用成员函数

如果成员函数中变量或参数的名字与成员变量的名字相同，可以吗？如果可以，如何区分使用谁呢？

```
Complex(double real, double imag) {
  //real = real???
  //imag = imag???
}
void setReal(int real) {
  //real = real???
}
```

使用this指针

- **this指针：在成员函数中可以使用的一个特殊的指针变量，该指针指向调用成员函数的那个对象**

```
Complex(double real, double imag) {
    this->real = real;
    this->imag = imag;
}
void setReal(int real) {
    this->real = real;
}
```

```
Complex c1(1,2); //调用构造函数，this指针指向对象c1
Complex c2(3,4); //调用构造函数，this指针指向对象c2
c1.setReal(5);   //调用setReal函数，this指针指向对象c1
```

this指针

- this指针常用于在成员函数的返回值中返回调用函数的对象自身

```
class Complex {
public:
    double real, imag;
    void print() {
        cout << real << "," << imag;
    }
    Complex(double r, double i) {
        real = r; imag = i;
    }
    Complex &addOne() {
        this->real++;  //等效于real++;
        this->print(); //等效于print()
        return *this;
    }
};
int main() {
    Complex c1(1, 1), c2(0, 0);
    c2 = c1.addOne();
    return 0;
}
```

输出结果：2,1

this指针的原理

- 早期的C++编译器会将C++面向对象部分的代码在编译时翻译成C程序的代码再进行编译，所以引入了this指针

```
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() {
    CCar car;
    setPrice(&car, 20000);
    return 0;
}
```

❑思考

```
class A {
public:
    int i;
    void hello() {
        cout << "hello" << endl;
    }
};
int main() {
    A *p = nullptr;
    p->hello(); //结果会怎样?
}
```

```
struct A {
    int i;
};
void hello(A *this) {
    cout << "hello" << endl;
}

int main() {
    A *p = NULL;
    hello(p);
}
```

能编译吗？能运行吗？结果是什么？

能编译，能运行，输出hello

如果hello函数是

```
void hello() {
    cout << i << "hello" << endl;
}
```

还能编译和运行吗？

注：本质上上述程序还是存在undefined behavior。编译时可加-fsanitize=undefined检查

本讲概览

- this指针
- 成员对象
  - 成员初始化列表
- 静态成员
- 只读成员
  - 只读对象
  - 只读成员函数
  - 只读成员变量
- 友元

❑ 成员对象：一个类的成员变量是另一个类的对象

- 某些材料将有成员对象的类称为封闭(enclosing)类，但该术语并没有在C++标准里出现过，所以本讲义不使用这个术语

```
class CTyre {    // 轮胎类
    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
    int price;  // 价格
    CTyre tyre;
    CEngine engine;
public:
    CCar(int p, int tr, int tw) {
        price = p;
        tyre = CTyre(tr,tw);
    }
};
```

CCar类有一个CTyre类的成员对象和一个CEngine类的成员对象

思考：CCar的构造函数是否有问题？

成员初始化列表

- 重新认识构造函数：构造函数中的语句真的是在给成员变量初始化吗？

```
class CCar {     // 汽车类
    int price;   // 价格
    CTyre tyre;
    CEngine engine;
public:
    CCar(int p, int tr, int tw) {
        price = p;
        tyre = CTyre(tr,tw);
    }
};
```

这里是赋值，并不是初始化！！！
CCar car(1,2,3) 会报编译错误

在C++中，成员变量的初始化是在调用构造函数之前。
所以在调用构造函数的那一刻起，每一个成员都已经有了自己的初始值

成员初始化列表

- 构造函数提供了一种特殊的语法，即在声明后附加一个**成员初始化列表**

```
构造函数(参数表) : 成员变量1 初始化器1, 成员变量2 初始化器2, ... {
    ......
}
```

- 如果在构造对象时匹配上了目前这个构造函数，则先根据初始化列表执行成员的初始化，再执行构造函数
- 初始化器可以是大括号的统一初始化，对于成员对象则可使用小括号调用对应的构造函数进行初始化
- 初始化器需要的值可以从构造函数的参数中获取，并且允许参数名和成员变量名具有相同的名字

```
class CCar {     // 汽车类
    int price;   // 价格
    CTyre tyre;
    CEngine engine;
public:
    CCar(int price, int tr, int tw): price{price}, tyre(tr, tw) { }
};
```

对象初始化与销毁的一般流程

- 对象初始化一般按照以下的流程：
  - 如果某个成员在 初始化列表 中提及，则按初始化列表中的初始化值初始化这个成员
  - 如果某个成员提供了 默认初始化器，则按默认初始化器的初始化值初始化这个成员
  - 如果某个成员不满足上述两条件，则默认初始化它
    - 简单类型什么都不做，类类型调用默认构造函数
  - 当成员全部初始化完毕后，再调用构造函数
- 初始化每个成员的顺序是按照成员列表的顺序执行的（即定义在最“上面”的最先被构造），与初始化列表的书写顺序无关
- 对象销毁时的析构顺序和其构造顺序是相反的
  - 先调用析构函数，再逆序析构每个成员（即写在最“下面”的最先被析构）

对象初始化与销毁的一般流程

❑ 默认初始化器

```
class Student {
public:
    int age;
    char name[10]{"hahaha"};
    Student(int age) : age{age} {}
};

int main() {
    Student alice(19);
    cout << alice.name << endl;
}
```

默认初始化器

对象初始化与销毁的一般流程

```
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; }
};

class CCar {
private:
    CEngine engine;
    CTyre tyre;
public:
    CCar() { cout << "CCar contructor" << endl; }
    ~CCar() { cout << "CCar destructor" << endl; }
};

int main() {
    CCar car;
    return 0;
}
```

输出结果

```
CEngine contructor
CTyre contructor
CCar contructor
CCar destructor
CTyre destructor
CEngine destructor
```

成员对象的复制构造

- 对于含有成员对象的类，其对象如果是用默认复制构造函数初始化的，那么它里面的成员对象也会用复制构造函数初始化
  - 如果自己写复制构造函数，也需要注意成员对象的初始化问题

```
class A {
public:
    A() { cout << "default" << endl; }
    A(const A &a) { cout << "copy" << endl; }
};

class B {
    A a;
};

int main() {
    B b1, b2(b1);
    return 0;
}
```

输出结果

```
default
copy
```

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时的实参就是b1

本讲概览

- this指针
- 成员对象
  - 成员初始化列表
- 静态成员
- 只读成员
  - 只读对象
  - 只读成员函数
  - 只读成员变量
- 友元

静态成员

- 在定义前面加了static关键字的成员
  - 作用：类似全局变量或全局函数，将类当做命名空间来用
  - 普通成员变量每个对象有各自的一份
    - 静态成员变量对于每个类就只有一份
  - 普通成员函数必须具体作用于某个对象
    - 静态成员函数并不具体作用于某个对象

```
class CRectangle{
private:
    int w, h;
    static int nTotalArea;   //静态成员变量
    static int nTotalNumber; //静态成员变量
public:
    CRectangle(int w_, int h_);
    ~CRectangle();
    static void PrintTotal(); //静态成员函数
};
```

静态成员

- 静态成员不需要通过对象就能访问
  - 除了和普通成员一样采取 对象名.成员名、指针->成员名、引用.成员名 访问以外
  - 还可以通过 类名::静态成员名 的方式访问
- 设置静态成员的目的
  - 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

- 成员列表中的静态成员变量大多只能是声明而非定义
  - 一般情况下，必须在类定义的外面专门对静态成员变量进行声明，同时可以初始化
  - 否则编译能通过，链接不能通过

```
class A {
public:
    static int a; // 是声明，不是定义
  //static int a{0} 这样的默认初始化也是不允许的!
};

int A::a{0};       // 要在类外定义（不带 static）

int main() {
    A::a = 42;
}
```

静态成员函数

- 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
- 静态成员函数中不能使用this指针

```
class A {
    static void f() { // 静态成员函数，用static修饰
        cout << "Hello" << endl;
    }
    static void g();  // 也可以把定义放在类外
};

void A::g() { }       // 类外定义需要加上 类名::，但不写static

int main() {
    A::f();           // 如同带有命名空间一样调用
}
```

本讲概览

- this指针
- 成员对象
  - 成员初始化列表
- 静态成员
- 只读成员
  - 只读对象
  - 只读成员变量
  - 只读成员函数
- 友元

只读对象

- 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，声明为只读对象
  - 只读对象只能使用构造函数、析构函数、有const说明的函数（只读成员函数）
  - 可读取所有成员变量，但不可修改（除非用mutable修饰）

```
class Sample {
private:
    int value;
public:
    void GetValue() { }
};
const Sample obj; //只读对象
obj.GetValue();   //错误，不能调用非const成员函数
```

  - 只读对象或只读对象引用作为函数形参，可避免在函数修改对象

只读成员函数

- 在类的成员函数说明后面可以加const关键字，则该成员函数成为只读成员函数
  - 只读成员函数内部不能改变该类的成员变量的值，也不能调用该类中的非只读成员函数，但可调用静态成员函数
  - 声明和定义只读成员函数时都必须使用const 关键字

```
class Sample {
public:
    int value;
    Sample() { }
    void setValue() const;
    void func() { };
};
void Sample::setValue() const {
    value = 0;  //编译错误，只读成员函数不能访问非只读成员变量
    func();     //编译错误，只读成员函数不能调用非只读成员函数
}
int main(){
    const Sample o;
    o.setValue();//只读对象可以调用只读成员函数
    return 0;
}
```

只读成员函数

- 两个函数，名字和参数表都一样，但是一个是const，另一个不是，这种情况也**算作重载**

```
class CTest{
    int n;
public:
    CTest() { n = 1; }
    int getValue() const { return n; } //重载
    int getValue() { return 2 * n ; }  //重载
};
int main() {
    const CTest objTest1;
    CTest objTest2;
    cout<< objTest1.getValue() << ", " <<objTest2.getValue();
    return 0;
}
```

mutable成员变量

- 用mutable关键字修饰的成员变量可以在const成员函数中修改

```
class CTest {
public:
    bool GetData() const {
        n++;  //n使用mutable修饰，可在只读函数中修改
        return flag;
    }
private:
    mutable int n;
    bool flag;
};
```

只读成员变量

- 使用const修饰的成员变量为只读成员变量
  - 没有办法通过任何方式被赋值（在构造函数和只读函数中也都不行）
  - 必须被显式地初始化，即必须拥有一个默认成员初始化器，或者出现在成员初始化列表中

```
class A {
    const int data{0}; // 只读成员变量，默认初始化器
    A() {
        data = 1;      // 错误：不能对只读类型赋值
    }
    void f() const {   // 这与函数 const 限定与否无关
        data = 1;      // 错误：不能对只读类型赋值
    }
};
```

```
class A {
    const int data; // 如果不带默认初始化器的话
    A(): data{42} { // 那么必须使用初始化列表为它初始化
    }
};
```

本讲概览

- this指针
- 成员对象
  - 成员初始化列表
- 静态成员
- 只读成员
  - 只读对象
  - 只读成员函数
  - 只读成员变量
- **友元**

友元

- 友元：让某个类以外的函数可以访问这个类的私有成员
  - 友元函数：一个类的友元函数可以访问该类的私有成员
  - 友元类：如果A是B的友元类，那么A的所有成员函数都可以访问B的私有成员
- 声明友元：friend关键字
  - 如果想允许函数 f 访问类 A 的私有成员，只需要把 f 的声明抄一遍放在 A 的成员列表中，然后加上 friend 关键字修饰
  - 声明友元可在类定义的任何位置，不受访问范围关键词的约束
- 为什么要用友元？

友元函数

```
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;
    //声明友元，mostExpensiveCar函数是CCar的友元
    friend int mostExpensiveCar(CCar cars[], int total);
    //声明友元，CDriver类的modifyCar函数是CCar类的友元
    friend void CDriver::modifyCar(CCar *pCar);
    //可以将一个类的成员函数（包括构造和析构函数）声明为另一个类的友元
};

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;
}
```

友元类

- 友元类之间的关系不能传递，不能继承

```
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
};
```

本讲小结

- 正确性/鲁棒性
  - 成员对象需要使用初始化列表进行初始化
  - 只读对象可以防止对象的值被修改
- 可扩展性
  - this指针可以用于返回调用函数的当前对象，可以支持表达式的扩展
  - 友元使得私有成员具备例外访问的能力
- 可复用性
  - 类的成员变量可以是对象，从而复用其他类的属性和操作
- 可理解性
  - 使用成员对象可以将对象之间的关系显式化
  - 静态成员将与某个类关系密切的全局函数和变量与类关联在一起

上机安排/作业二

- 3月11日 13:00-15:00 理一1235机房

- 上机安排
  - 第4讲、第5讲知识点答疑和拓展补充

- 作业二：
  - 时间：3月11日 12:00—3月21日 23:59
  - 完成OpenJudge的第二次作业题目，AC即通过

谢谢

欢迎在课程群里填写问卷反馈