课程目标

1. C++类的成员
2. 友元函数和友元类的特点
3. 重载运算符
4. C++中const的用法

课程实验

- this指针的使用
- 拷贝构造函数实验
- 设计立方体类(Cube)
- 设计点和圆的关系类
- 友元函数和友元类实验
- 重载操作符实验

课堂引入

1. 类

在C++中添加类(Class)可以对程序结构和功能产生很大的影响。类是面向对象编程(Object-Oriented Programming,OOP)的基本构建块,它引入了封装、继承和多态等概念,使得代码更加模块化、可维护和可重用。类是C++能实现其他如运算符重载,抽象数据,模板等强大功能的基础。

2. 友元

友元可以分为友元类和友元函数。在C++中,友元(friend)是一种机制,允许一个类或函数访问另一个类的私有成员。友元关系可以在类的声明中通过`friend`关键字来建立。友元关系的主要作用包括访问私有成员，增强封装,运算符重载,类之间的共享等功能。

3. 重载运算符

C++中允许程序员对已有的运算符进行重载,为特定类或自定义类型定义新的行为。

重载运算符对C++有很重要的意义,可以提高代码的可读性,模块化和可重用性。

授课进程

一 类的成员

1.1 访问限制

✧ `public`,`private`,`protected` 为属性/方法限制的关键字。

> 1. 在类的内部(作用域范围内),没有访问权限之分,所有成员可以相互访问
> 2. 在类的外部(作用域范围外)，访问权限才有意义：public,private,protected
> 3. 在类的外部,只有public修饰的成员才能被访问,在没有涉及继承与派生时,private和protected是同等级的,外部不允许访问

访问属性	属性	对象内部	对象外部
public	公有	可访问	可访问
protected	保护	可访问	不可访问
private	私有	可访问	不可访问

> 类的数据成员中不能使用auto、extern和register等进行修饰**(C++11中支持在定义时进行初始化)**

```
// 封装两层含义
// 1. 属性和行为合成一个整体
// 2. 访问控制,现实事物本身有些属性和行为是不对外开放
class Person {
// 人具有的行为(函数)
public:
    // 内联函数
    void show() {
        cout << "我有钱,年轻,个子又高,就爱嘚瑟!" << endl;
    }
// 人的属性(变量)
public:
    int m_nTall;         // 多高,可以让外人知道
protected:
    int m_nMoney;        // 有多少钱,只能儿子知道
private:
    int m_nAge;          // 年龄,不想让外人知道
};

int main(int argc, char *argv[]) {
    Person p;
    p.m_nTall = 180;
    // p.m_nMoney 保护成员外部无法访问
    // p.m_nAge 私有成员外部无法访问
    p.show();

    return 0;
}
```

注意: 将成员变量设置为private

> 1. 可赋予客户端访问数据的一致性
>
>   如果成员变量不是public,客户端唯一能够访问对象的方法就是通过成员函数。如果类中所有public权限的成员都是函数,客户在访问类成员时只会默认访问函数,不需要考虑访问的成员需不需要添加()。
>
> 2. 可细微划分访问控制
>
>   使用成员函数可使得我们对变量的控制处理更加精细。如果我们让所有的成员变量为public,每个人都可以读写它。如果我们设置为private,我们可以实现"不准访问"、"只读访问"、"读写访问",甚至你可以写出"只写访问"。

```
class AccessLevels {
public:
    // 对只读属性进行只读访问
    int getReadOnly() { return readOnly; }
```

```
    // 对读写属性进行读写访问
    void setReadWrite(int val) { readWrite = val; }
    int getReadWrite() { return readWrite; }
    // 对只写属性进行只写访问
    void setWriteOnly(int val) { writeOnly = val; }
private:
    int readOnly;   // 对外只读访问
    int noAccess;   // 外部不可访问
    int readWrite;  // 读写访问
    int writeOnly;  // 只写访问
};
```

> 课堂练习
>
> 设计一个Person类,Person类具有m_strName和m_nAge属性,提供初始化函数(init),并提供对m_strName和m_nAge的读写函数(set,get),但必须确保m_nAge的赋值在有效范围内(0-100),超出有效范围则拒绝赋值,并提供方法输出姓名和年龄。

1.2 类成员

1.2.1 成员函数

成员函数必须在类内部声明,可以在类内部定义,也可以在类外部定义。如果在类内部定义,就默认是内联函数。内联函数的定义通常应该放在类定义的同一头文件中,而不是在源文件中。这是为了保证内联函数的定义在调用该函数的每个源文件中是可见的。成员函数可被重载。

1.2.2 成员变量

类的数据成员,可以是基本类型或自定义类型,通常用于存储对象的状态信息。

示例:

✧ 定义一个点(Point)类,具有以下属性和方法:

- 属性: `x` 坐标, `y` 坐标
- 方法: 设置x,y的坐标值; 输出坐标的信息;

[例4]

```
class Point {
public:
    void setPoint(int x, int y);        // 成员函数
    void displayPoint();                // 成员函数
private:
    int m_nXpos;                        // 成员变量
    int m_nYpos;                        // 成员变量
};
```

1.3 C++默认的成员函数

C++编译器对于一个空类,即使程序没定义任何成员,编译器也会添加以下的函数成员:

> 1. 默认构造函数
> 2. 析构函数
> 3. 拷贝构造函数
> 4. 赋值运算符(等号:operator=)
> 5. 取址运算符(operator&)(一对,一个非const的,一个const的)

注意:

> 1. 构造函数可以被重载,可以带参数;
> 2. 析构函数只有一个,不能被重载,不能带参数;
> 3. 默认构造函数没有参数,它什么也不做。当没有重载无参构造函数时,会调用默认构造函数(也可以使用`default`)。

1.4 类对象

✧ 定义类对象时,将为其分配存储空间。

```
Point pt;       // 编译器分配了足以容纳一个 Point 对象的存储空间。pt指的就是那个存储空间。
// Point *p = new Point;
```

1.5 this指针

1.5.1 什么是this指针

在C++中,`this`指针是一种特殊的指针,它指向当前对象本身。其作用就是指向**非静态成员函数**所作用的对象,每个成员函数的第一个参数实际上都有默认`this`指针参数。**静态成员函数无法使用this指针。**

我们都知道类的成员函数可以访问类的数据,那么成员函数如何知道哪个对象的数据成员要被操作呢,原因在于**每个对象都拥有一个指针:** this指针,通过this指针来访问自己的地址。注意:this指针并不是对象的一分,this指针所占的内存大小不会反应在sizeof操作符上。

1.5.2 this指针和静态成员

静态成员不能使用 this 指针,因为静态成员相当于是共享的,不属于某个对象。

this只能在成员函数中使用。全局函数,静态函数都不能使用this。

静态函数如同静态变量一样,不属于具体的哪一个对象,静态函数表示了整个类范围意义上的信息,而this指针却实实在在的对应一个对象,所以this指针当然不能被静态函数使用了,同理,全局函数也一样。

(1)this指针是什么时候创建的?

this在成员函数的开始执行前构造的,在成员的执行结束后清除。

(2)this指针如何传递给类中函数的?绑定?还是在函数参数的首参数就是this指针.那么this指针又是如何找到类实例后函数的?

this是通过函数参数的首参数来传递的。this指针是在调用之前生成的。类在实例化时,只分配类中的变量空间,并没有为函数分配空间。

(3)this指针只有在成员函数中才有定义

在获得一个对象后,也不能通过对象使用this指针。我们也无法知道一个对象的this指针的位置(在成员函数里才有this指针的位置)。当然,在成员函数里,你是可以知道this指针的位置的(可以&this获得),也可以直接使用的。

1.5.3 this指针的使用

在类的非静态成员函数中返回类对象本身的时候,我们可以使用圆点运算符`(*this).`,箭头运算符this->,另外我们也可以返回关于 *this 的引用,这样我们可以像输入输出流那样进行"级联"操作。

1.5.4 何时使用 this 指针

当我们需要将一个对象作为整体引用而不是引用对象的一个成员时,常见情况是在该函数返回对调用该类对象的引用(*this)。

[例1]

```
#include<iostream>
#include<string>

using namespace std;

class StuInfoManage {
public:
    StuInfoManage(int s = 0, string n = " ", int a = 0, int g = 0) {
        m_nNo = s;
        m_strName = n;
        m_nAge = a;
        m_nGrade = g;
    }

    // 使用 this 指针进行赋值
    void setName(string name) {
        m_strName = name;
    }

    int setAge(int a) {
        this->m_nAge = a;   // m_nAge = a;
        return (*this).m_nAge;  // 使用this指针返回该对象的年龄
    }

    void print() {
        cout << "the m_strName is " << this->m_strName << endl;
        cout << "the m_nNo is " << m_nNo << endl;  // 隐式使用this指针打印
        cout << "the m_nAge is " << (*this).m_nAge << endl;
        cout << "the m_nGrade is " << this->m_nGrade << endl;
    }

private:
    int m_nNo;
    string m_strName;
    int m_nAge;
```

```
    int m_nGrade;
};

int main(int argc, char *argv[]) {
    StuInfoManage sim1(761, "Zhangsan", 19, 3);
    sim1.print();       // 输出信息
    sim1.setAge(12);    // 使用 this 指针修改年龄
    sim1.print();       // 再次输出
    return 0;
}
```

```
the m_strName is Zhangsan
the m_nNo is 761
the m_nAge is 19
the m_nGrade is 3

the m_strName is Zhangsan
the m_nNo is 761
the m_nAge is 12
the m_nGrade is 3
```

以上的例子中,我们要设置一个学生的信息,需要对每一个相关变量所属的成员函数进行调用(名字调用设置名字的成员函数),我们还可以使用this的另外一个功能,让他实现级联,方便我们调用,对此,我们对上述代码进行修改,如下:

[例2]

```
#include <iostream>
#include <string>

using namespace std;

class StuInfoManage {
private:
    int m_nNo;
    string m_strName;
    int m_nAge;
    int m_nGrade;

public:
    StuInfoManage(int s = 0, string n = " ", int a = 0, int g = 0) {
        m_nNo = s;
        m_strName = n;
        m_nAge = a;
        m_nGrade = g;
    }

    // 所有的相关函数,都返回该对象的引用,这样就可以实现级联,方便编码
    StuInfoManage &setName(string s) {
        this->m_strName = s;
        return (*this);
    }

    StuInfoManage &setNo(int sno) {
        this->m_nNo = sno;
        return *this;
```

```
    }

    StuInfoManage &setGrade(int grade) {
        this->m_nGrade = grade;
        return *this;
    }

    StuInfoManage &setAge(int age) {
        this->m_nAge = age;
        return *this;
    }

    void print() {
        cout << " the sname is " << this->m_strName << endl;
        cout << " the sno is " << this->m_nNo << endl;
        cout << " the age is " << this->m_nAge << endl;
        cout << " the grade is " << this->m_nGrade << endl << endl;
    }
};

int main(int argc, char *argv[]) {
    StuInfoManage sim;    // 使用默认参数
    sim.setName("Zhangsan").setNo(457).setGrade(2012).setAge(20);    // 级联
    // 使用this指针使串联的函数调用成为可能
    sim.print();
    return 0;
}
```

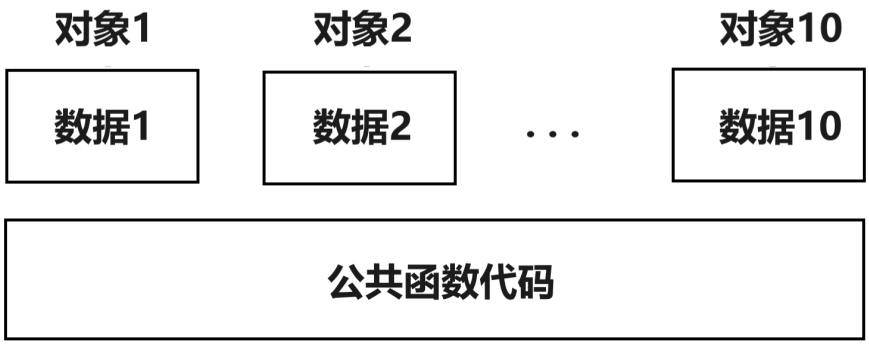
```
the sname is Zhangsan
the sno is 457
the age is 20
the grade is 2012
```

this指针的使用

- 当形参和成员变量同名时,可用this指针来区分
- 在类的非静态成员函数中返回对象本身,可使用 return *this

1.5.5 this指针指针工作原理

C++的数据和操作也是分开存储，并且每一个非内联成员函数(non-inline member function)只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码，那么问题是这一块代码是如何区分哪个对象调用自己的呢?



c++规定,this指针是隐含在对象成员函数内的一种指针。当一个对象被创建后,它的每一个成员函数都含有一个系统自动生成的隐含指针this,用以保存这个对象的地址,也就是说虽然我们没有写上this指针,编译器在编译的时候也是会加上。因此this也称为"指向本对象指针",this指针并不是对象的一部分,不会影响sizeof(对象)的结果。

this指针是C++实现封装的一种机制,它将对象和该对象调用的成员函数连接在一起,在外部看来,每一个对象都拥有自己的函数成员。一般情况下,并不写this,而是让系统进行默认设置。this指针永远指向当前对象。

成员函数通过this指针即可知道操作的是那个对象的数据。this指针是一种隐含指针,它隐含于每个类的非静态成员函数中。this指针无需定义,直接使用即可。

> 注意：静态成员函数内部没有this指针,静态成员函数不能操作非静态成员变量。

> **关于this指针的一个精典回答:**
>
> 当你进入一个房子后,你可以看见桌子、椅子、地板等,但是房子你是看不到全貌了。对于一个类的实例来说,你可以看到它的成员函数、成员变量,但是实例本身呢？this是一个指针,它时时刻刻指向你这个实例本身。

1.6 构造函数

✧ 构造函数是一个特殊的、与类同名的成员函数，该函数没有返回值，用来保证每个对象的数据成员具有合适的初始值。

✧ 在创建类的对象时,编译器就会运行一个构造函数。

✧ 构造函数可以重载(可以有0-n个形参)

```
class Sales_item {
public:
    Sales_item();             // 默认构造函数
    Sales_item(const std::string&);
    Sales_item(std::istream&);
};                            // 构造函数自动执行
```

✧ 只要创建该类型的一个对象,编译器就运行一个构造函数:

```
Sales_item item1("0-201-54848-8");
Sales_item *ptr = new Sales_item();
```

第一种情况下,运行接受一个 string 实参的构造函数,来初始化变量item1。

第二种情况下,动态分配一个新的 Sales_item 对象,通过运行默认构造函数初始化该对象。

1.6.1 默认构造函数

✧ 默认构造函数不含形参。只要定义一个对象时没有提供初始化式,就使用默认构造函数。如: A a;

✧ 为所有形参提供默认实参的构造函数也定义了默认构造函数。例如:

```
class A {
public:
    A(int a = 1, char c = '') {}
private:
    int ia;
    char c1;
};
```

同样,还可指定使用默认的构造函数,这里需要使用 `default` 关键字。

1.6.2 合成的默认构造函数

✧ **只有当一个类没有定义构造函数时,编译器才会自动生成一个默认构造函数。**

✧ 一个类只要定义了一个构造函数,编译器也不会再生成默认构造函数。

1.6.3 构造函数初始化数据成员

构造函数初始化数据成员有以下几种常见方法:

1. 使用赋值运算符初始化数据成员

可以在构造函数中使用赋值运算符将数据成员初始化为特定的值。例如:

```
class MyClass {
private:
    int x;
public:
    MyClass(int a) { x = a; } // 使用赋值运算符初始化x
    void print() { cout << "x = " << x << endl; }
};
```

2. 使用初始化列表初始化数据成员

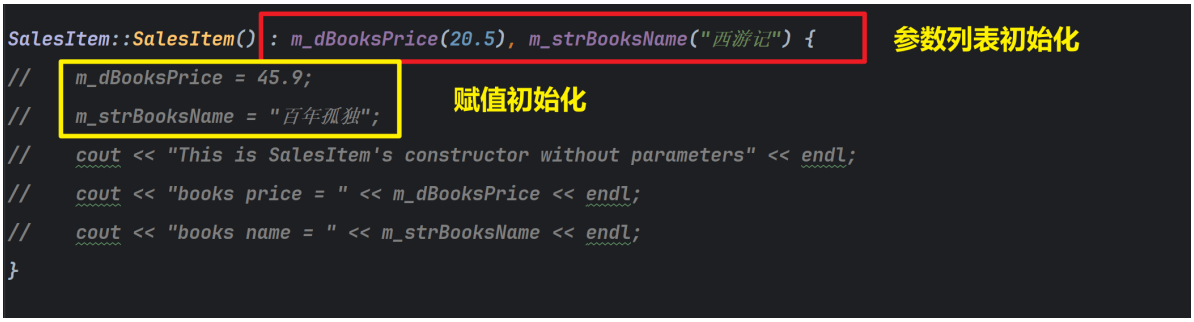
与其他函数一样,构造函数具有名字、形参表和函数体。与其他函数不同的是,构造函数可以包含一个构造函数初始化列表。

初始化列表是一种更高效的方法,它可以在构造函数执行之前初始化数据成员。初始化列表使用冒号(:)和逗号(,)分隔的成员初始化列表。例如:

```
Sales_item::Sales_item(const string &book) :
    isbn(book),
    units_sold(0),
    revenue(0.0) {
    ......
}
```

构造函数初始化列表以一个冒号开始,接着是一个以逗号分隔的数据成员列表,每个数据成员后面跟一个放在圆括号中的初始化式。

构造函数可以定义在类的内部或外部。构造函数初始化只在构造函数的定义中指定。



构造函数分两个阶段执行:

1. 初始化阶段;
2. 普通的计算阶段。

初始化列表属于初始化阶段(1),构造函数函数体中的所有语句属于计算阶段(2)。

初始化列表比构造函数体先执行。不管成员是否在构造函数初始化列表中显式初始化,类类型的数据成员总是在初始化阶段初始化。

哪种类需要初始化式

const对象或引用类型的对象,可以初始化,但不能对它们赋值,而且在开始执行构造函数的函数体之前要完成初始化。初始化 const 或引用类型数据成员的唯一机会是构造函数初始化列表中,在构造函数函数体中对它们赋值会出现错误。

没有默认构造函数的类类型的成员,以及 const 或引用类型的成员,必须在初始化列表中完成初始化。

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};

ConstRef::ConstRef(int ii) {
   i = ii;   // ok
   ci = ii;  // error
   ri = ii;  // error
}
```

应该这么初始化:

```
ConstRef::ConstRef(int ii) : i(ii), ci(i), ri(ii) {}
```

1.6.4 成员初始化的次序

每个成员在构造函数初始化列表中只能指定一次。重复初始化,编译器一般会有提示。

成员被初始化的次序就是成员声明的次序,跟初始化列表中的顺序无关。

1.6.5 初始化式表达式

初始化式可以是任意表达式

```
Sales_item(const std::string &book, int cnt, double price)
    : isbn(book),
    units_sold(cnt),
    revenue(cnt*price) {
}

Sales_item(const std::string &book, int cnt, double price) {
    isbn = book;
    units_sold = cnt;
    revenue = cnt*price;
}
```

1.6.6 类类型的数据成员的初始化式

初始化类类型的成员时,要指定实参并传递给成员类型的一个构造函数,可以使用该类型的任意构造函数。

```
Sales_item():isbn(10, '9'), units_sold(0), revenue(0.0) {}
```

1.6.7 类对象的数据成员的初始化

✧ 在类A的构造函数初始化列表中没有显式提及的每个成员,使用与初始化变量相同的规则来进行初始化。

✧ 类类型的数据成员,运行该类型的默认构造函数来初始化。

✧ 内置或复合类型的成员的初始值依赖于该类对象的作用域：在局部作用域中不被初始化,在全局作用域中被初始化为0。

```
class A {
private:
    int ia;
    B b;
};
```

✧ A类对象A a;不管a在局部作用域还是全局作用域,b使用B类的构造函数来初始化,ia的初始化取决于a的作用域,a在局部作用域,ia不被初始化,a在全局作用域,ia初始化0。

建议:

✧ 如果定义了其他构造函数,也提供一个不带形参的构造函数。

如果类包含内置或复合类型(如int& 或string*)的成员,它应该定义自己的构造函数来初始化这些成员。每个构造函数应该为每个内置或复合类型的成员提供初始化。

1.6.8 隐式类类型转换

只含单形参的构造函数能够实现从形参类型到该类类型的一个隐式转换

```
#include <iostream>

class A {
public:
    A(int x) {
        b = x;
    }
    bool EqualTo(const A& a1) {
        return this->b == a1.b;
    }
private:
    int b;
};

int main() {
    A a(1);
    bool bEq = a.EqualTo(1);     // 参数为1,实现从int型到A的隐式转换
    return 0;
}
```

通过将构造函数声明为 explicit,来防止在需要隐式转换的上下文中使用构造函数:

```
class A {
public:
    explicit A(int a) {
        ia = a;
    }
    bool EqualTo(const A& a) {
        return ia == a.ia;
    }
private:
    int ia;
};

int main() {
    A a(1);
    bool bEq = a.EqualTo(1);     // 参数为1,实现从int型到A的隐式转换
    return 0;
```

```
}
```

```
error: cannot convert 'int' to 'const A&'
```

抑制由构造函数定义的隐式转换

✧ **通常,除非有明显的理由想要定义隐式转换,否则单形参构造函数应该声明为explicit。将构造函数设置为explicit可以避免错误。**

1.6.9 禁止复制

✧ 有些类需要完全禁止复制。例如 iostream 类就不允许复制。延伸：容器内元素不能为 iostream

✧ 为了防止复制,类必须显式声明其复制构造函数为private。(比如单例模式)

饿汉式单例模式指的是在类加载时就立即初始化，并且创建单例对象。

注意事项

- 为了确保其他对象不能复制单例类的实例，我们需要显式地禁止拷贝构造函数和赋值运算符。
- 饿汉式单例在类加载时就创建了单例实例，因此不存在线程安全问题。但是，它也可能导致资源的浪费，如果程序启动后并未使用到这个实例。
- 懒汉式单例（特别是线程安全版本）在需要时才创建实例，但可能会在多线程环境下引入额外的同步开销。

1.6.10 赋值操作符

✧ 与复制构造函数一样,如果类没有定义自己的赋值操作符,则编译器会合成一个。

(1)重载赋值操作符

```
Sales_item& operator=(const Sales_item &);
```

1.7 拷贝构造函数

复制构造函数是一种特殊构造函数,只有1个形参,该形参(常用 const&修饰)是对该类类型的引用。

当定义一个新对象并用一个同类型的对象对它进行初始化时,将**显式使用**复制构造函数。

```
People a1;
People a2 = a1;
```

当将该类型的对象传递给函数或函数返回该类型的对象时,将**隐式使用**复制构造函数。

```
People Func(People b) {...}
```

先看一个例子,有一个学生类,数据成员是学生的人数和名字:

```cpp
#include <iostream>
using namespace std;

class Student {
public:
    //Student(const A&);
    Student();
    ~Student();     // 析构函数,用以释放资源
private:
    int num;
    char *name;
};

Student::Student() {
    name = new char(20);
    cout << "Student" << endl;
}

Student::~Student() {
    cout << "~Student " << (int)name << endl;
    delete name;
    name = nullptr;
}

int main(int argc, char *argv[]) {
    // 花括号让s1和s2变成局部对象,方便测试
    Student s1;
    Student s2(s1);// 复制对象

    return 0;
}
```

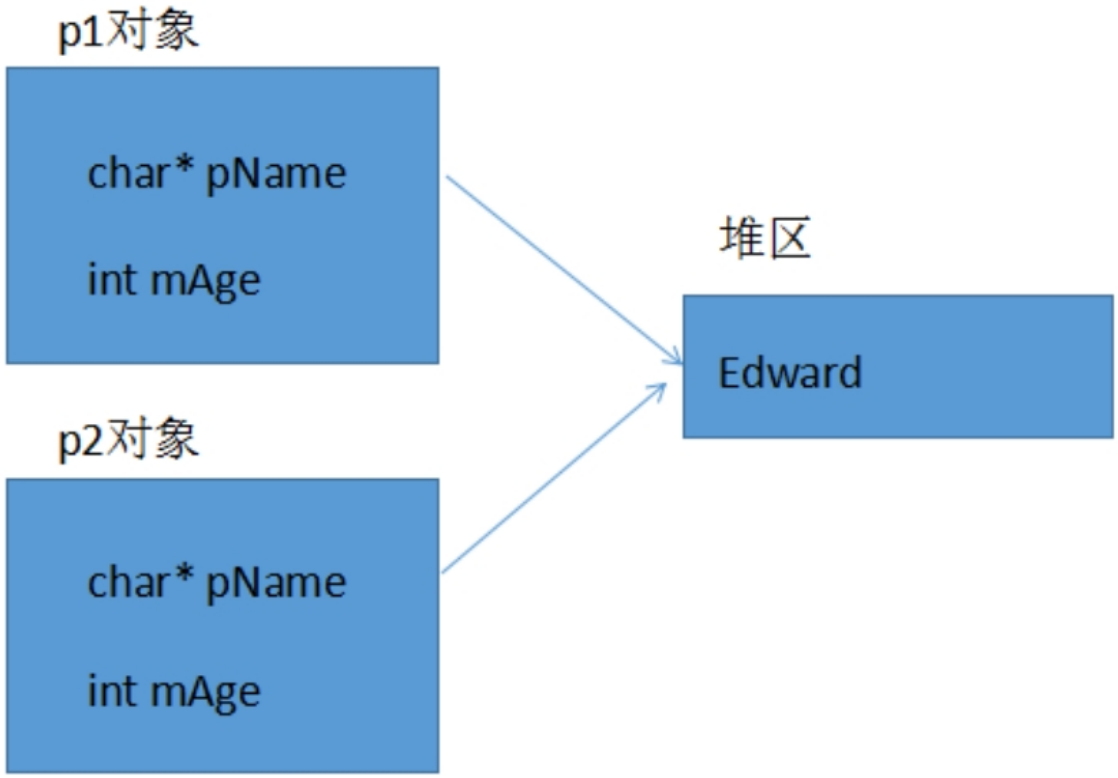
```
C:\Users\ciyeer\Desktop\蜗牛-南京\day04\copyConstructor\cmake-build-debug\copyConstructor.exe
This is Student's constructor
This is Student's destructor
This is Student's destructor
```

```
ciyeer@ubuntu:~/day04$ g++ test_07.cpp
ciyeer@ubuntu:~/day04$ ./a.out
This is Student's constructor
This is Student's destructor
This is Student's destructor
free(): double free detected in tcache 2
Aborted
ciyeer@ubuntu:~/day04$
```

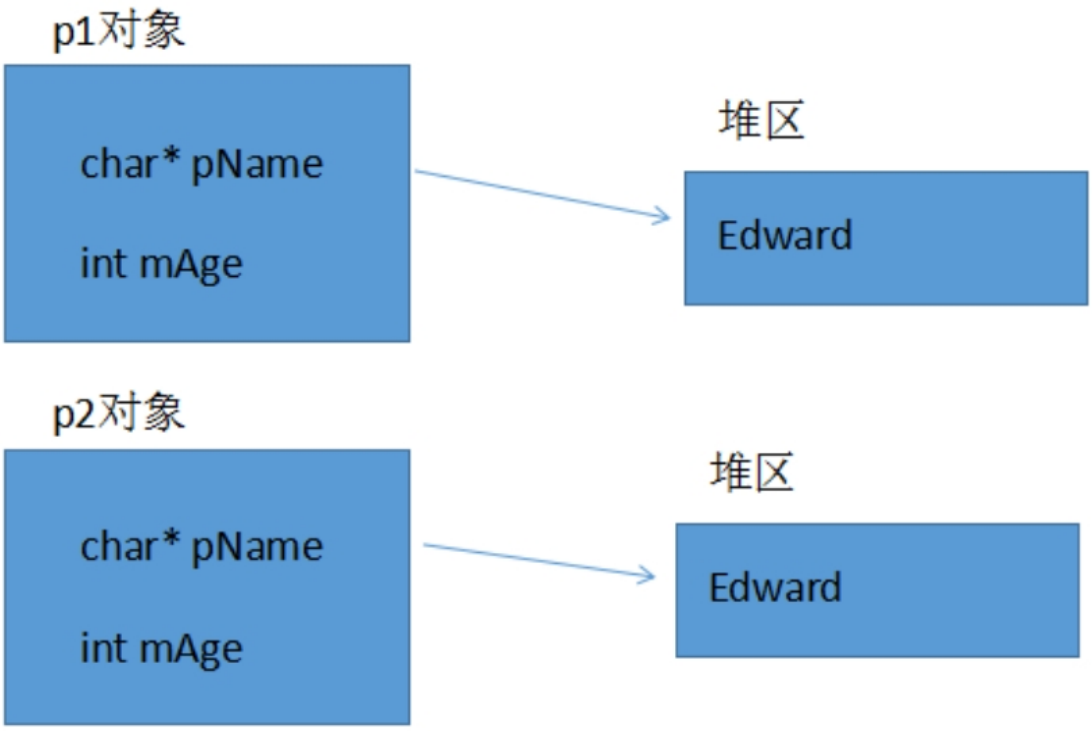
执行结果：调用一次构造函数,调用两次析构函数,两个对象的指针成员所指内存相同,这会导致什么问题呢？name指针被分配一次内存,但是程序结束时该内存却被释放了两次,会导致崩溃！

对一个已知对象进行拷贝,编译系统会自动调用拷贝构造函数,如果用户未定义拷贝构造函数,则会调用默认拷贝构造函数。

这是由于编译系统在我们没有自己定义拷贝构造函数时,会在拷贝对象时调用默认拷贝构造函数,进行的是**浅拷贝**！即对指针name拷贝后会出现两个指针指向同一个内存空间。



在对含有指针成员的对象进行拷贝时,必须要自己定义拷贝构造函数,使拷贝后的对象指针成员有自己的内存空间,即进行深拷贝,这样就避免了内存重复释放发生。



添加了自己定义拷贝构造函数的例子：(深拷贝)

```
#include <iostream>
#include <cstring>
using namespace std;

class Student {
public:
    Student();
```

```
    Student(const Student &);
    ~Student();
private:
    int m_nNum;
    char *m_pName;
};

Student::Student() {
    m_pName = new char(20);
    cout << "This is Student's constructor" << endl;
}

Student::Student(const Student &s) {
    m_pName = new char(20);
    memcpy(m_pName, s.m_pName, strlen(s.m_pName));
    cout << "copy Student constructor" << endl;
}

Student::~Student() {
    cout << "This is Student's destructor " << endl;
    delete m_pName;
    m_pName = nullptr;
}

int main() {
    Student s1;
    Student s2(s1);
    return 0;
}
```

执行结果：调用一次构造函数,一次自定义拷贝构造函数,两次析构函数。两个对象的指针成员所指内存不同。

```
This is Student's constructor
copy Student constructor
This is Student's destructor
This is Student's destructor
```

> **总结:**
>
> 浅拷贝只是对指针的拷贝,拷贝后两个指针指向同一个内存空间,深拷贝不但对指针进行拷贝,而且对指针指向的内容进行拷贝,经深拷贝后的指针是指向两个不同地址的指针。浅拷贝带来问题的本质在于析构函数释放多次堆内存,使用智能指针(std::shared_ptr),可以完美解决这个问题。

> **注意:**
>
> 当对象中存在指针成员时,除了在复制对象时需要考虑自定义拷贝构造函数,还应该考虑以下情形:
>
> 1. 当函数参数为对象时,实参传递给形参的实际上是实参的拷贝对象,系统自动通过拷贝构造函数实现;
> 2. 当函数的返回值为一个对象时,该对象实际上是函数内对象的一个拷贝,用于返回函数调用处。

1.8 析构函数

✧ 析构函数是一个特殊的、与构造函数名类似的成员函数,需要在构造函数之前添加一个"~", 该函数没有返回值,用于释放该类的资源。构造函数的用途之一是自动获取资源;与之相对的是,析构函数的用途之一是回收资源。除此之外,析构函数可以执行任意类设计者希望在该类对象的使用完毕之后执行的操作。

✧ 析构函数可用于释放构造对象时或在对象的生命期中所获取的资源。

✧ 不管类是否定义了自己的析构函数,编译器都自动执行类中非 static 数据成员的析构函数。

何时调用析构函数

✧ 撤销(销毁)类对象时会自动调用析构函数。

✧ 变量(类对象)在超出作用域时应该自动撤销(销毁)。

✧ 动态分配的对象(new A)只有在指向该对象的指针被删除时才撤销(销毁)。