

第 9 章 继承与多态

2021 年 3 月 26 日

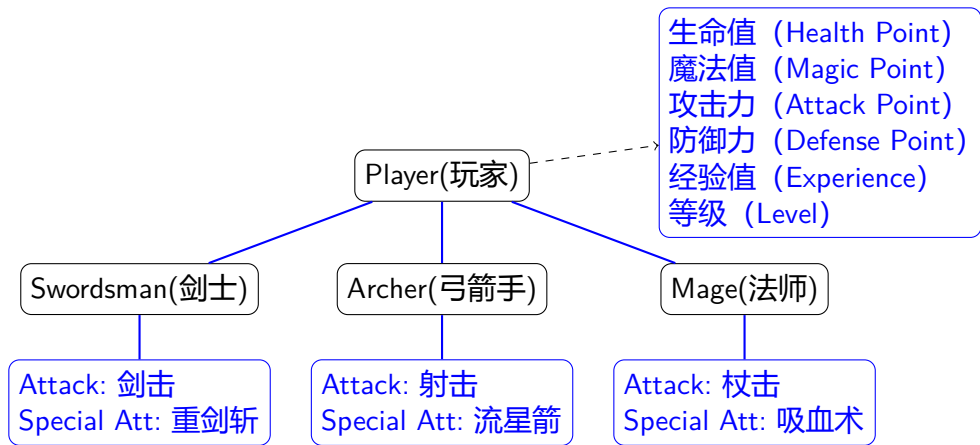
- 函数—面向过程: `void fun() {}`

- 函数—面向过程: `void fun(){}`
- 类—基于对象: `class mytype{};`

- 函数—面向过程: `void fun(){}`
- 类—基于对象: `class mytype{};`
- 模板—泛型编程: `template < typename T >`

- 函数—面向过程: `void fun(){}`
- 类—基于对象: `class mytype{};`
- 模板—泛型编程: `template < typename T >`
- 继承—面向对象





1 继承

- 定义基类
- 定义派生类
- 访问控制
- 类型转换

2 构造、拷贝控制与继承

- 派生类对象的构造
- 拷贝控制与继承

3 虚函数与多态性

- 虚函数
- 动态绑定
- 抽象类
- 继承与组合
- 再探计算器

学习目标

- 理解继承的内涵和基本语法;
- 掌握拷贝控制成员与继承的关系;
- 掌握并学会运用动态绑定技术。

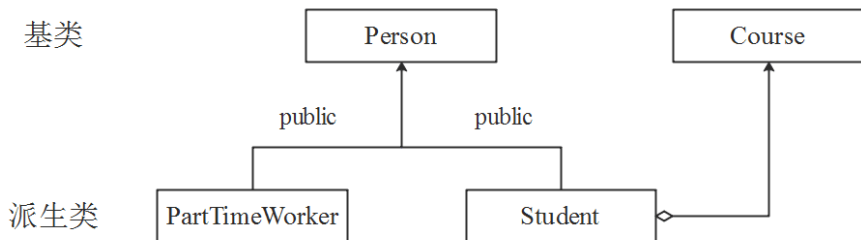
9.1 继承—定义基类和派生类

继承

基类：被继承的类；派生类：通过继承产生的新类

例 9.1:

下面设计一个简单的人员系统，包括两类人员：学生（指大学生）和兼职员工。该系统包含以下几个类：Person、Student、PartTimeWorker 和 Course。



9.1 继承—定义基类和派生类

例 9.1 中定义基类 Person:

```
class Person {                                //人员类
protected:
    string m_name;                            //名字
    int m_age;                                //年龄
public:
    Person(const string &name = "", int age = 0):m_name(name), m_age(
        age){}
    virtual ~Person() = default;              //default关键字见教材6.2.1节
    const string& name() const { return m_name; }
    int age() const { return m_age; }
    void plusOneYear() { ++m_age; }           //年龄自增
};
```

9.1 继承—定义基类和派生类

例 9.1 中定义类 Course:

```
class Course {                //课程类
    string m_name;            //课程名
    int m_score;              //成绩
public:
    Course(const string &name = "", int score = 0):m_name(name),
        m_score(score) {}
    void setScore(int score) { m_score = score; }
    int score() const { return m_score; }
    const string& name() const { return m_name; }
};
```

9.1 继承—定义基类和派生类

类名后紧随一个冒号，后跟以逗号分隔的基类列表

定义派生类 PartTimeWorker (基类访问限定符: public、protected、private):

```
class PartTimeWorker : public Person { //兼职人员类，公有继承Person
private:
    double m_hour;                    //工作小时数
    static double ms_payRate;         //每小时工资
public:
    PartTimeWorker(const string &name, int age, double h=0): Person(
        name, age), m_hour(h){}
    void setHours(double h) { m_hour = h; }
    double salary() { return m_hour * ms_payRate; }
};
double PartTimeWorker::ms_payRate = 7.53; //静态成员初始化
```

9.1 继承—定义基类和派生类

定义派生类 Student:

```
class Student : public Person { //学生类, 公有继承Person
private:
    Course m_course;           //课程信息
public:
    Student(const string &name, int age, const Course &c):Person(name,
        age), m_course(c) {}
    Course& course() { return m_course; }
};
```

9.1 继承—定义基类和派生类

提示：使用关键字 `final` 防止被继承

可以利用 C++11 提供的关键字 `final` 来阻止继承的发生：

```
class NoDerived final { }; //NoDerived不能作为基类被继承
```



如果我们想让例 9.1 中派生类 `Student` 和 `PartTimeWorker` 不再被任何类继承，我们应该如何做？

9.1 继承—访问控制

派生类访问基类成员：private VS public

private 只能在类内部访问 (i.e., 派生类无权访问基类成员), 而 public 又失去了隐私保护

9.1 继承—访问控制

派生类访问基类成员：private VS public

private 只能在类内部访问 (i.e., 派生类无权访问基类成员), 而 public 又失去了隐私保护

访问限定声明

	a	b	c	d
访问位置	该类成员函数	派生类成员函数	该类友元	该类对象

- **public** : 可以被 a、b、c 和 d 访问。
- **protected** : 可以被 a、b 和 c 访问。
- **private**: 可以被 a 和 c 访问。

9.1 继承—访问控制

下面代码正确吗？

```
class Base {  
    private:  int m_pri;           //private成员  
    protected:  int m_pro;       //protected成员  
    public:    int m_pub;         //public成员  
};  
  
class PubDerv : public Base {  
    void foo() {                  //派生类成员函数  
        m_pri = 10;              //错误：不能访问Base类私有成员  
        m_pro = 1;              //正确：可以访问Base类受保护成员  
    }  
};  
  
void test() {                    //全局函数  
    Base b;                      //类对象  
    b.m_pro = 10;                //错误：不能访问Base类受保护成员  
}
```

三类继承方式

- `public` 继承: 基类的 `protected` 和 `public` 属性在其派生类中**保持不变**。
- `protected` 继承: 基类的 `protected` 和 `public` 属性在派生类中变为 `protected`。
- `private` 继承: 基类的 `protected` 和 `public` 属性在派生类中变为 `private`。

9.1 继承—访问控制

三类继承方式

- `public` 继承: 基类的 `protected` 和 `public` 属性在其派生类中**保持不变**。
- `protected` 继承: 基类的 `protected` 和 `public` 属性在派生类中变为 `protected`。
- `private` 继承: 基类的 `protected` 和 `public` 属性在派生类中变为 `private`。

访问权限不仅取决于访问限定符，还取决于继承方式

以上三种继承，基类中的 `private` 属性在其派生类中均**保持不变**。

9.1 继承—访问控制

下面代码正确吗？

```
class PriDerv : private Base { //私有继承不影响派生类成员对 基类的访问
    void foo() {
        m_pro = 1;    //正确：可以访问Base类受保护成员
        m_pub = 1;    //正确：可以访问Base类公有成员
    }
};
void test() {
    PubDerv d1;
    d1.m_pub = 10;    //正确：m_pub在PubDerv中是公有的
    PriDerv d2;
    d2.m_pub = 1;     //错误：m_pub在PriDerv中是私有的
}
```

提示：公有继承是主流

由于私有继承和受保护继承均具有**局限性**，所以公有继承是主流的继承方式。

9.1 继承—访问控制

使用 using 声明

通过使用 using 声明，可以改变派生类中基类成员的访问权限：

```
class PubDerv : public Base {  
public:  
    using base::m_pro;      //声明为公有的  
};  
void test() {  
    PubDerv d;  
    d.m_pro;                 //正确  
}
```

注意：

派生类只能为它可以访问的名字提供 using 声明。

命名冲突

定义在派生类（内层作用域）的名字将会屏蔽掉基类（外层作用域）的同名成员：

```
class Base {
    protected:  int m_data;
    public:     void foo(int) { /*...*/ }
};

class Derived : public Base {
protected:
    int m_data;                //基类m_data被隐藏
public:
    int foo() {                //基类foo成员被隐藏
        return m_data;        //返回Derived::m_data
    }
};
```

9.1 继承—访问控制

命名冲突

如果在派生类里面需要访问基类的同名成员，则可以使用基类的**作用域运算符**：

```
class Derived : public Base {  
    /*...*/  
    int foo() {  
        return Base::m_data;  
    }  
};
```

```
Base b;  
Derived d;  
b.foo(10); //ok:调用Base::foo  
d.foo();  //ok:调用Derived::foo  
d.foo(10); //错误:Base::foo被隐藏
```

如果我们想调用基类中的 foo 函数，我们应该如何做？

```
d.Base::foo(10); //正确  
class Derived: public Base{  
public:
```

```
    using Base::foo;  
};  
d.foo(10); //正确
```


派生类到基类的转换

一个派生类不仅包含自己定义的（非静态）成员，而且还包含其从基类继承的成员。因此，可以将派生类对象当成基类对象使用，也就是说可以将基类的**指针或引用**与派生类对象**绑定**，例如：

```
PartTimeWorker w("Kevin", 21);  
Person p, *ptr;  
ptr = &w;           //基类指针ptr指向派生类对象w  
Person &p2 = w;      //基类引用绑定到派生类对象w  
p = w;              //派生类对象赋值给基类对象
```

9.1 继承—类型转换

派生类到基类的转换

虽然派生类可以自动转换为基类的引用或指针，但没有从基类到派生类的自动转换。这是因为基类对象不能提供派生类对象**新定义的部分**，例如：

```
PartTimeWorker *w2 = &p;  
w = p;
```

//错误：不能将基类转换为派生类

//错误：不能将基类转换为派生类

9.1 继承—类型转换

派生类到基类的转换

虽然派生类可以自动转换为基类的引用或指针，但没有从基类到派生类的自动转换。这是因为基类对象不能提供派生类对象**新定义的部分**，例如：

```
PartTimeWorker *w2 = &p;           //错误：不能将基类转换为派生类  
w = p;                             //错误：不能将基类转换为派生类
```

派生类与基类关系 IS-A，即用派生类对象来创建一个基类对象：

```
PartTimeWorker w("Kevin", 21);    //派生类对象  
Person p(w);                      //利用派生类对象构造基类对象
```

9.1 继承—类型转换

派生类到基类的转换

虽然派生类可以自动转换为基类的引用或指针，但没有从基类到派生类的自动转换。这是因为基类对象不能提供派生类对象**新定义的部分**，例如：

```
PartTimeWorker *w2 = &p;           //错误：不能将基类转换为派生类
w = p;                             //错误：不能将基类转换为派生类
```

派生类与基类关系 IS-A，即用派生类对象来创建一个基类对象：

```
PartTimeWorker w("Kevin", 21);    //派生类对象
Person p(w);                       //利用派生类对象构造基类对象
```

如果派生类以私有方式或受保护的方式继承基类，那么派生类将**不能自动转换**为基类类型，例如：

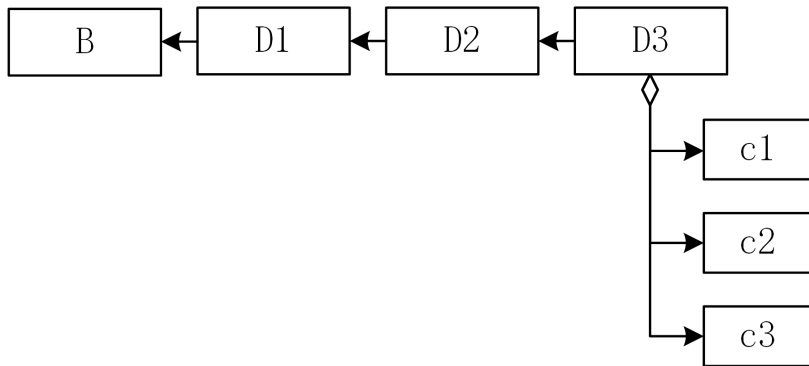
```
PriDerv d;                          //PriDerv私有继承Base
Base b(d);                          //错误：PriDerv不能转换为Base
```

提示：从派生类到基类的转换原则

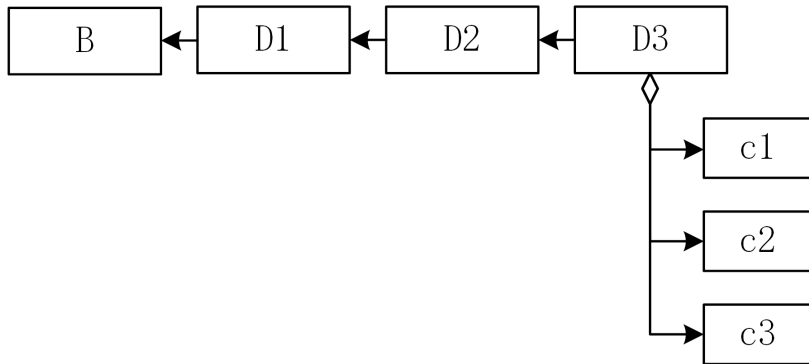
理解从派生类到基类的**隐式自动转换**需要明白三点：

- 这种转换只限于指针或引用类型；
- 转换的前提是公有继承；
- 没有从基类到派生类的隐式自动转换。

9.2 构造、拷贝控制与继承

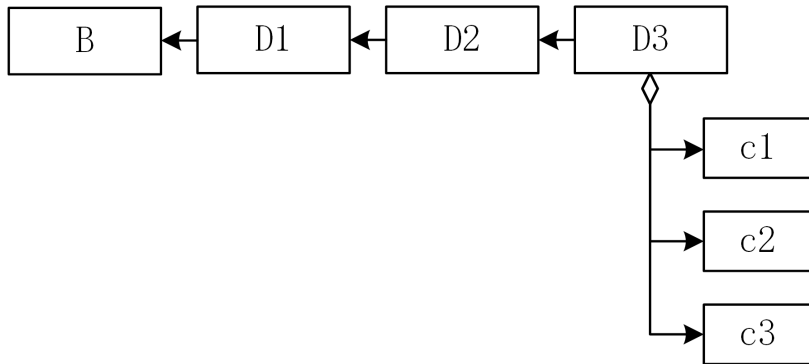


9.2 构造、拷贝控制与继承



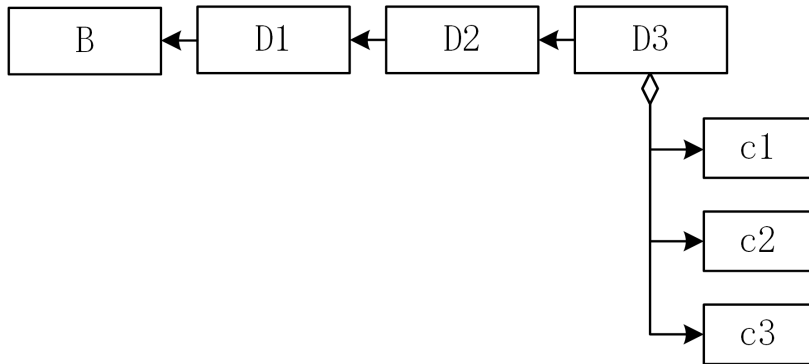
D3 obj;
构造顺序: B

9.2 构造、拷贝控制与继承



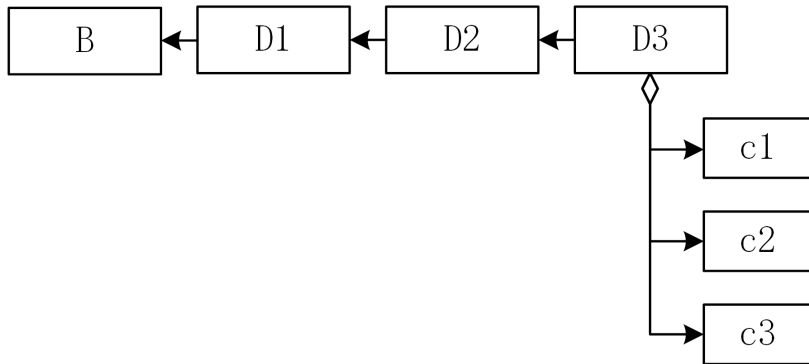
D3 obj;
构造顺序: B D1

9.2 构造、拷贝控制与继承



D3 obj;
构造顺序: B D1 D2

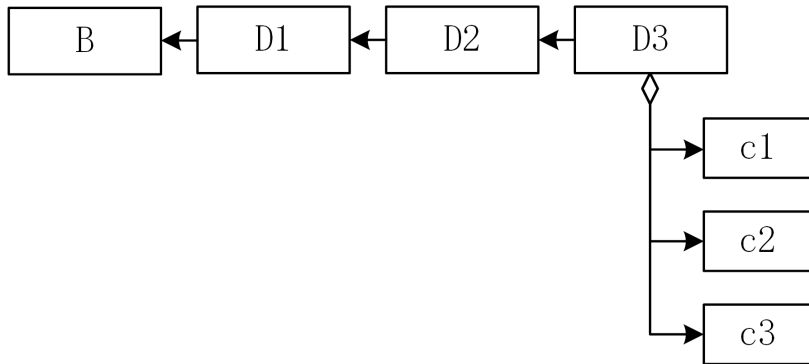
9.2 构造、拷贝控制与继承



D3 obj;

构造顺序: B D1 D2 D3

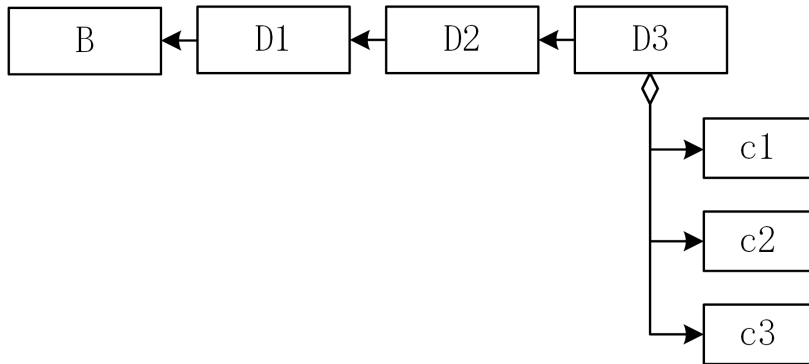
9.2 构造、拷贝控制与继承



D3 obj;

构造顺序: B D1 D2 D3(c1

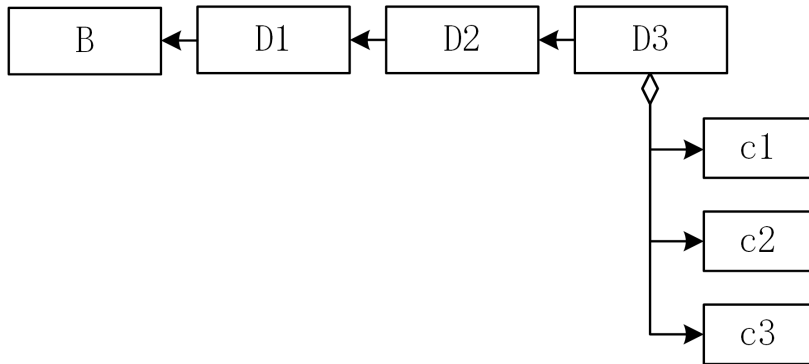
9.2 构造、拷贝控制与继承



D3 obj;

构造顺序: B D1 D2 D3(c1 c2

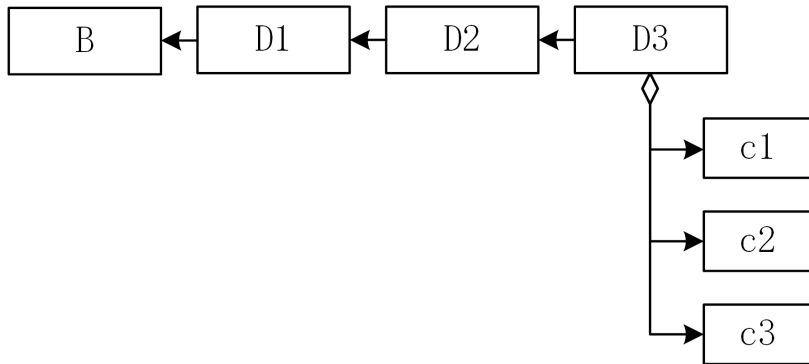
9.2 构造、拷贝控制与继承



D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

9.2 构造、拷贝控制与继承

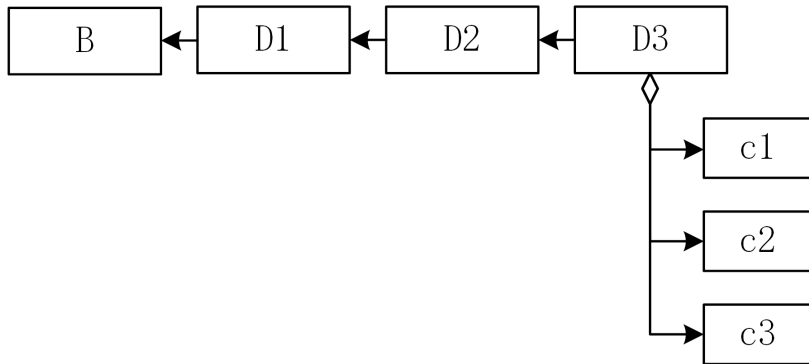


D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3

9.2 构造、拷贝控制与继承

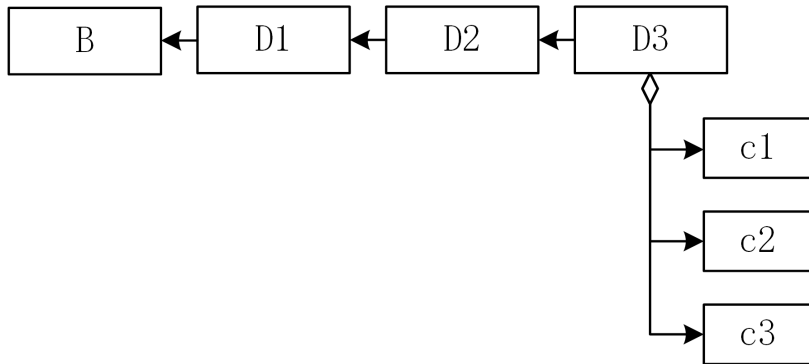


D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3(c3

9.2 构造、拷贝控制与继承

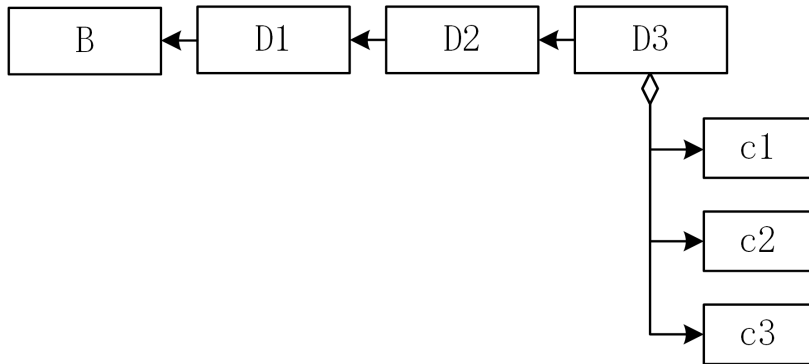


D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3(c3 c2

9.2 构造、拷贝控制与继承

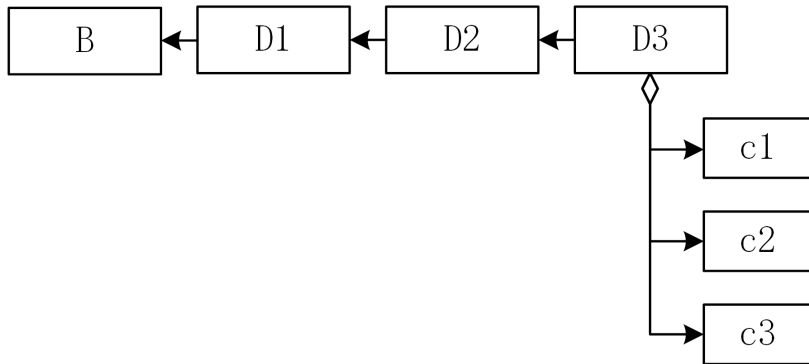


D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3(c3 c2 c1)

9.2 构造、拷贝控制与继承

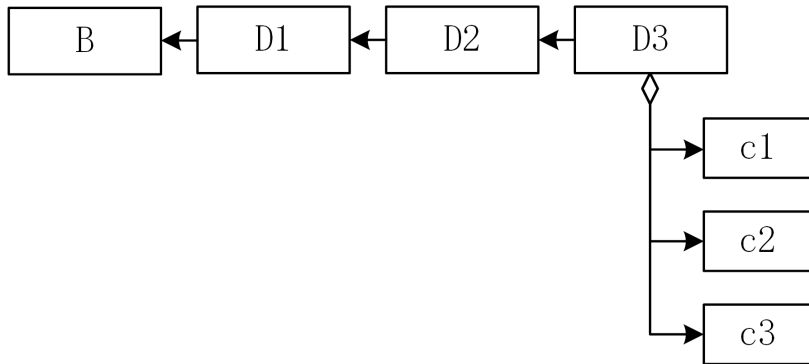


D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3(c3 c2 c1) D2

9.2 构造、拷贝控制与继承

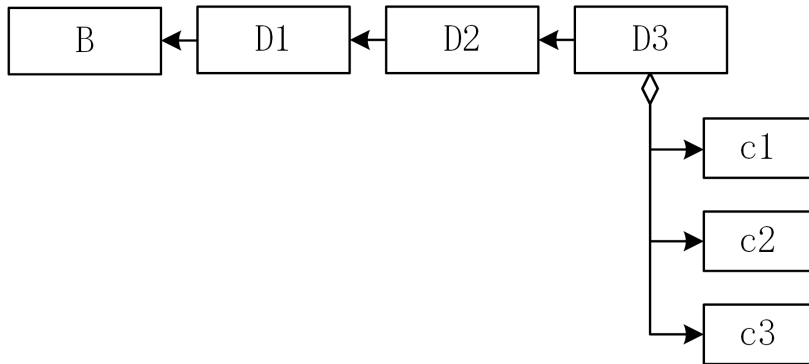


D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3(c3 c2 c1) D2 D1

9.2 构造、拷贝控制与继承



D3 obj;

构造顺序: B D1 D2 D3(c1 c2 c3)

析构顺序: D3(c3 c2 c1) D2 D1 B

9.2 构造、拷贝控制与继承—派生类对象的构造

派生类 Student 对象的构造

以上述 Student 类为例：

```
Student::Student(const string &name, int age, const Course &c): Person(name,
    age), /*初始化基类成员*/ m_course(c) /*初始化自有成员*/ {
    cout<<"Constr of Student"<<endl;
}
```

Student 类中成员 m_course 以复制构造的方式初始化。如下：

```
Course::Course(const Course &rhs): m_name(rhs.name), m_score(rhs.m_score) {
    cout<< "Copy constr of Course" <<endl;
}
```

9.2 构造、拷贝控制与继承—派生类对象的构造

派生类 Student 对象的构造

```
Person::Person(const string &name = "", int age = 0):m_name(name), m_age(age) {  
    cout<<"Constr of Person"<<endl; // 为Person类初始化添加标记:  
}
```

当创建 Student 类对象时: `Student s("Kevin", 19, Course("Math"));`

输出结果:

```
Constr of Person  
Copy constr of Course  
Constr of Student
```

9.2 构造、拷贝控制与继承—派生类对象的构造

派生类 Student 对象的构造

```
Person::Person(const string &name = "", int age = 0):m_name(name), m_age(age) {  
    cout<<"Constr of Person"<<endl; // 为Person类初始化添加标记:  
}
```

当创建 Student 类对象时: `Student s("Kevin", 19, Course("Math"));`

输出结果:

```
Constr of Person  
Copy constr of Course  
Constr of Student
```

提示: 存在继承关系的类的成员初始化

在派生类对象构造过程中, 每个类**仅负责**自己的成员的初始化。

9.2 构造、拷贝控制与继承—拷贝控制与继承

析构与继承

类似于构造函数，Course、Student 和 Person 类的析构函数的函数如下：

```
Person::~~Person() { cout<< "Destr of Person" <<endl; }  
Student::~~Student() { cout<< "Destr of Student" <<endl; }  
Course::~~Course() { cout<< "Destr of Course" <<endl; }
```

利用如下代码创建 Student 类对象：

```
Course c("Math");  
{  
    Student s("Kevin", 19, c); //思考：输出结果会是怎样的？  
}
```


析构与继承

输出结果：

Destr of Student

Destr of Course

Destr of Person

复制、移动与继承

一个派生类对象在**复制**或**移动**的时候，除复制或移动自有成员外，还要复制或移动基类部分的成员。因此，通常在复制或移动构造函数的初始化列表中调用基类的**复制或移动构造函数**。

```
class A{/*...*/};  
class B : public A {  
    string m_d;  
public:  
    B(const B &d):A(d), m_d(d.m_d){ }  
    B(B &&d):A(std::move(d)),m_d(std::move(d.m_d)){ }  
};
```

复制、移动与继承

一个派生类对象在**复制**或**移动**的时候，除复制或移动自有成员外，还要复制或移动基类部分的成员。因此，通常在复制或移动构造函数的初始化列表中调用基类的**复制或移动构造函数**。

```
class A{/*...*/};  
class B : public A {  
    string m_d;  
public:  
    B(const B &d):A(d), m_d(d.m_d){ }  
    B(B &&d):A(std::move(d)), m_d(std::move(d.m_d)){ }  
};
```

复制 A 的成员 复制 B 的成员

移动 A 的成员 移动 B 的成员

9.2 构造、拷贝控制与继承—拷贝控制与继承

赋值与继承

与复制和移动构造函数类似，必须在派生类的赋值运算符中**显式**调用基类的赋值运算符，才能正确地完成基类成员的赋值：

```
B& B::operator=(const B &d) {  
    if(this == &d) return *this;  
    A::operator=(d);           //赋值A的成员  
    m_d = d.m_d;               //赋值自身成员  
    return *this;  
}
```

9.2 构造、拷贝控制与继承—拷贝控制与继承

赋值与继承

与复制和移动构造函数类似，必须在派生类的赋值运算符中**显式**调用基类的赋值运算符，才能正确地完成基类成员的赋值：

```
B& B::operator=(const B &d) {  
    if(this == &d) return *this;  
    A::operator=(d);           //赋值A的成员  
    m_d = d.m_d;               //赋值自身成员  
    return *this;  
}
```

提示：派生类中使用基类的构造或赋值成员

如果基类中合成的构造函数、复制构造函数或赋值运算符是删除的或者是不可以访问的，那么派生类中对应的合成成员也是删除的。

9.3 虚函数与多态性

思考

如果基类成员的实现在派生类里面不再适用，怎么办？

9.3 虚函数与多态性

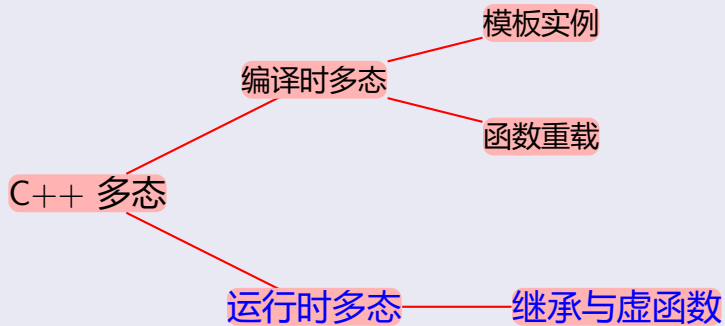
思考

如果基类成员的实现在派生类里面不再适用，怎么办？

```
class Animal{
    public:
        void eat() { std::cout << "I'm eating generic food."; }
};

class Cat : public Animal{
    public:
        void eat() { std::cout << "I'm eating a rat."; }
};
```

9.3 虚函数与多态性



9.3 虚函数与多态性—动态绑定

静态类型

指对象声明时的类型或表达式生成时的类型，在编译时就已经确定，例如：

```
class Base { }  
Base *p;      //指针p的静态类型为Base
```

9.3 虚函数与多态性—动态绑定

静态类型

指对象声明时的类型或表达式生成时的类型，在编译时就已经确定，例如：

```
class Base { }  
Base *p;      //指针p的静态类型为Base
```

动态类型

指**指针或引用**所绑定的对象的类型，仅在运行时可知，例如：

```
class Derived : public Base { };  
Derived d;      //非指针或引用，动态类型与静态类型相同  
Base *p = &d;   //指针p的动态类型为Derived
```

9.3 虚函数与多态性—虚函数

Shape 类

```
class Shape {  
protected:    string m_name;  
public:  
    Shape(const string &s = ""):m_name(s) { }  
    //虚函数  
    virtual double area() const { return 0; }  
    const string& name() { return m_name; }  
};
```

9.3 虚函数与多态性—虚函数

Shape 类

```
class Shape {  
protected:    string m_name;  
public:  
    Shape(const string &s = ""):m_name(s) { }  
    //虚函数  
    virtual double area() const { return 0; }  
    const string& name() { return m_name; }  
};
```

Circle 类

```
class Circle : public Shape {  
private:    double m_rad;  
public:  
    Circle(double r=0, const string &s = ""):  
        Shape(s),m_rad(r) { }  
    double area() const {  
        return 3.1415926*m_rad*m_rad; }  
};
```

9.3 虚函数与多态性—虚函数

Shape 类

```
class Shape {  
protected:    string m_name;  
public:  
    Shape(const string &s = ""):m_name(s) { }  
    //虚函数  
    virtual double area() const { return 0; }  
    const string& name() { return m_name; }  
};
```

Circle 类

```
class Circle : public Shape {  
private:    double m_rad;  
public:  
    Circle(double r=0, const string &s = ""):  
        Shape(s),m_rad(r) { }  
    double area() const {  
        return 3.1415926*m_rad*m_rad; }  
};
```

Square 类

```
class Square : public Shape {  
private:    double m_len;  
public:  
    Square(double l=0, const string &s = ""):Shape(s),m_len(l) {}  
    double area() const { return m_len*m_len; }  
};
```

9.3 虚函数与多态性—动态绑定

动态绑定

除需要重写基类的虚函数外，还必须用**基类的指针或引用**才能触发动态绑定

9.3 虚函数与多态性—动态绑定

动态绑定

除需要重写基类的虚函数外，还必须用**基类的指针或引用**才能触发动态绑定

利用指针触发动态绑定

```
Shape sh, *p = &sh; //p指向Shape类对象
Square sq(1.0);
cout<<p->area()<<endl; //打印输出0
```

```
p = &sq; //p的动态类型为Square
cout<<p->area(); //打印输出1.0
```

9.3 虚函数与多态性—动态绑定

动态绑定

除需要重写基类的虚函数外，还必须用**基类的指针或引用**才能触发动态绑定

利用指针触发动态绑定

```
Shape sh, *p = &sh; //p指向Shape类对象
Square sq(1.0);
cout<<p->area()<<endl; //打印输出0
```

```
p = &sq;           //p的动态类型为Square
cout<<p->area(); //打印输出1.0
```

利用引用触发动态绑定

```
bool operator>(const Shape &a, const Shape &b){return a.area()>b.area();}
```

```
Shape *p = nullptr;
Square sq(2.0);
Circle ci(1.2);
```

```
if(sq>ci)
    p = &sq;
cout<<p->area()<<endl;
```


9.3 虚函数与多态性—动态绑定

虚析构函数

通常情况下，基类的析构函数应该是虚函数，保证正确 delete 一个动态派生类对象，例如：

```
class Shape {
public:
    virtual ~Shape() { cout<<"Destr of Shape"<<endl; }
};
class Circle : public Shape {
public:
    ~Circle() { cout<<"Destr of Circle"<<endl; }
};
Shape *p = new Circle();
delete p; //输出Destr of Circle~~Destr of Shape
```

9.3 虚函数与多态性—动态绑定

虚析构函数

通常情况下，基类的析构函数应该是虚函数，保证正确 delete 一个动态派生类对象，例如：

```
class Shape {
public:
    virtual ~Shape() { cout<<"Destr of Shape"<<endl; }
};
class Circle : public Shape {
public:
    ~Circle() { cout<<"Destr of Circle"<<endl; }
};
Shape *p = new Circle();
delete p; //输出Destr of Circle~~Destr of Shape
```

注意：如果基类析构函数为非虚函数，则 delete 一个指向派生类对象的基类指针将产生未定义的行为

9.3 虚函数与多态性—动态绑定

注意

- 动态绑定必须通过基类**指针**或**引用**绑定到派生类对象才能触发。

9.3 虚函数与多态性—动态绑定

注意

- 动态绑定必须通过基类**指针**或**引用**绑定到派生类对象才能触发。
- 派生类中与基类虚函数对应的重写版本**自动为虚函数**，不必进行 virtual 声明。

注意

- 动态绑定必须通过基类**指针**或**引用**绑定到派生类对象才能触发。
- 派生类中与基类虚函数对应的重写版本**自动为虚函数**，不必进行 virtual 声明。
- **内联成员、静态成员和模板成员**均不能声明为虚函数☹。

注意

- 动态绑定必须通过基类**指针**或**引用**绑定到派生类对象才能触发。
- 派生类中与基类虚函数对应的重写版本**自动为虚函数**，不必进行 virtual 声明。
- **内联成员、静态成员和模板成员**均不能声明为虚函数😞。
- 派生类版本的声明必须与基类版本的声明完全一致，包括**函数名、形参列表和返回值类型**。

9.3 虚函数与多态性—动态绑定

注意

- 动态绑定必须通过基类**指针**或**引用**绑定到派生类对象才能触发。
- 派生类中与基类虚函数对应的重写版本**自动为虚函数**，不必进行 virtual 声明。
- **内联成员、静态成员和模板成员**均不能声明为虚函数☹。
- 派生类版本的声明必须与基类版本的声明完全一致，包括**函数名、形参列表和返回值类型**。
- 动态绑定的实现是有代价的，大量的虚函数会导致程序性能的下降。

9.3 虚函数与多态性—动态绑定

```
class Base{
public:
    virtual Base* foo() { cout << "Base" << endl;
        return this; }
};

class Derived : public Base{
public:
    Derived* foo() { cout << "Derived" << endl;
        return this; }
};

void test() {
    Derived d;
    Base *p = &d;
    p->foo();
    d.foo();
}
```

例外

基类版本返回基类指针或引用，派生类版本可以返回派生类指针或引用

9.3 虚函数与多态性—动态绑定

```
class Base{
public:
    virtual Base* foo() { cout << "Base" << endl;
        return this; }
};

class Derived : public Base{
public:
    Derived* foo() { cout << "Derived" << endl;
        return this; }
};

void test() {
    Derived d;
    Base *p = &d;
    p->foo();
    d.foo();
}
```

例外

基类版本返回基类指针或引用，派生类版本可以返回派生类指针或引用

调用 test 函数输出

Derived
Derived

9.3 虚函数与多态性—动态绑定

```
class Base{
public:
    virtual void fun(int i=0) {
        cout << "Base:" << i << endl; }
};

class Derived : public Base{
public:
    void fun(int i=1) {
        cout << "Derived:" << i << endl; }
};

void test(){
    Derived d;
    Base *p = &d;
    p->fun();
    d.fun();
}
```

注意

如果参数具有默认值，
则各个版本中对应形参的默认值必须相同

9.3 虚函数与多态性—动态绑定

```
class Base{
public:
    virtual void fun(int i=0) {
        cout << "Base:" << i << endl; }
};

class Derived : public Base{
public:
    void fun(int i=1) {
        cout << "Derived:" << i << endl; }
};

void test(){
    Derived d;
    Base *p = &d;
    p->fun();
    d.fun();
}
```

注意

如果参数具有默认值，
则各个版本中对应形参的默认值必须相同

调用 test 函数输出

Derived:0

Derived:1

9.3 虚函数与多态性—动态绑定

final 和 override 说明符

C++11 引入了关键字 `override` 用来**显式**说明派生类的函数要覆盖基类的虚函数。类似的，可以使用关键字 `final` **阻止**派生类覆盖基类版本的虚函数。

```
struct B {  
    virtual void fun1(int) { }  
    virtual void fun2() { }  
    void fun3() { }  
};  
  
struct D1 : public B {  
    void fun1() override { }    //错误：基类没有不带参数的fun1函数  
    void fun2() final { }      //D1::fun2为最终版本  
    void fun3() override { }   //错误：基类没有可覆盖的函数  
};  
  
struct D2 : public D1 {  
    void fun2() { }            //错误：不允许覆盖基类D1中的fun2函数  
};
```

9.3 虚函数与多态性—抽象类

纯虚函数

上面定义的 Shape 类，实际上并不代表具体的几何形状类，因此它的成员函数 area 的定义是没有意义的，Shape 类只是几何形状的一个抽象，因此也不希望用户创建一个 Shape 类对象。C++ 允许将这样的虚函数声明为**纯虚函数**：

```
class Shape {  
public:  
    virtual double area() const = 0; //纯虚函数  
}  
Shape sh;                          //错误：不能创建抽象类的实例
```

9.3 虚函数与多态性—抽象类

纯虚函数

上面定义的 Shape 类，实际上并不代表具体的几何形状类，因此它的成员函数 area 的定义是没有意义的，Shape 类只是几何形状的一个抽象，因此也不希望用户创建一个 Shape 类对象。C++ 允许将这样的虚函数声明为**纯虚函数**：

```
class Shape {  
public:  
    virtual double area() const = 0; //纯虚函数  
}  
Shape sh;                          //错误：不能创建抽象类的实例
```

提示：公有继承方式下的基类成员函数的继承与覆盖

- 不要重新定义基类非虚函数
- 如果需要重新定义基类函数，则该函数应声明为虚函数
- 派生类继承基类**非虚函数的接口和实现、虚函数的接口和默认实现、纯虚函数的接口**

9.3 虚函数与多态性—继承与组合

Cat 类

```
class Cat {  
protected:  
    string m_name;  
public:  
    void meow() {    //喵喵叫  
        cout<<"meowing"<<endl;  
    }  
};
```

Dog 类

```
class Dog {  
protected:  
    string m_name;  
public:  
    void bark() {    //汪汪叫  
        cout<<"barking"<<endl;  
    }  
};
```

IS-A 设计

```
class Dog : public Cat {  
public:  
    void bark();  
};  
Dog dog;           //创建一个Dog类对象  
dog.bark();        //调用bark函数
```

虽然 `dog` 能汪汪叫，但是它也会喵喵叫，显然这是不符合事实的。`Dog` 不是一种 `Cat`，显然不是属于关系。

9.3 虚函数与多态性—继承与组合

IS-A 设计

```
class Dog : public Cat {  
public:  
    void bark();  
};  
Dog dog;           //创建一个Dog类对象  
dog.bark();        //调用bark函数
```

虽然 dog 能汪汪叫，但是它也会喵喵叫，显然这是不符合事实的。Dog 不是一种 Cat，显然不是属于关系。

HAS-A 设计

```
class Dog {  
    Cat m_cat;  
public:  
    void bark();  
};  
Dog dog;           //创建一个Dog类对象  
dog.bark();        //调用bark函数
```

虽然 dog 不能喵喵叫了，但不符合自然逻辑。Dog 和 Cat 类显然不是组合关系。

9.3 虚函数与多态性—继承与组合

抽象共有属性

将 Cat 和 Dog 共有的属性**抽象**，包括名字和发声行为，形成一个新的公共基类 Mammal

```
class Mammal {  
protected:  
    string m_name;  
public:  
    virtual void sounding() = 0;  
};
```

9.3 虚函数与多态性—继承与组合

抽象共有属性

将 Cat 和 Dog 共有的属性**抽象**，包括名字和发声行为，形成一个新的公共基类 Mammal

```
class Mammal {  
protected:  
    string m_name;  
public:  
    virtual void sounding() = 0;  
};
```

```
class Cat : public Mammal {  
protected:  
    void meow();  
public:  
    void sounding() override {  
        meow(); }  
};
```

9.3 虚函数与多态性—继承与组合

抽象共有属性

将 Cat 和 Dog 共有的属性**抽象**，包括名字和发声行为，形成一个新的公共基类 Mammal

```
class Mammal {  
protected:  
    string m_name;  
public:  
    virtual void sounding() = 0;  
};
```

```
class Cat : public Mammal {  
protected:  
    void meow();  
public:  
    void sounding() override {  
        meow(); }  
};
```

```
class Dog : public Mammal {  
protected:  
    void bark();  
public:  
    void sounding() override {  
        bark(); }  
};
```

9.3 虚函数与多态性—继承与组合

抽象共有属性

将 Cat 和 Dog 共有的属性**抽象**，包括名字和发声行为，形成一个新的公共基类 Mammal

```
class Mammal {  
protected:  
    string m_name;  
public:  
    virtual void sounding() = 0;  
};
```

```
class Cat : public Mammal {  
protected:  
    void meow();  
public:  
    void sounding() override {  
        meow(); }  
};
```

```
class Dog : public Mammal {  
protected:  
    void bark();  
public:  
    void sounding() override {  
        bark(); }  
};
```

```
Dog dog;  
Cat cat;  
dog.sounding(); //dog能正常的汪汪叫  
cat.sounding(); //cat能正常的喵喵叫
```

9.3 虚函数与多态性—继承与组合

抽象共有属性

将 Cat 和 Dog 共有的属性**抽象**，包括名字和发声行为，形成一个新的公共基类 Mammal

```
class Mammal {  
protected:  
    string m_name;  
public:  
    virtual void sounding() = 0;  
};
```

```
class Cat : public Mammal {  
protected:  
    void meow();  
public:  
    void sounding() override {  
        meow(); }  
};
```

```
class Dog : public Mammal {  
protected:  
    void bark();  
public:  
    void sounding() override {  
        bark(); }  
};
```

```
Dog dog;  
Cat cat;  
dog.sounding(); //dog能正常的汪汪叫  
cat.sounding(); //cat能正常的喵喵叫
```

- 既统一了接口，又实现了不同的行为。
- 符合事实和自然逻辑。

9.3 虚函数与多态性—再探计算器

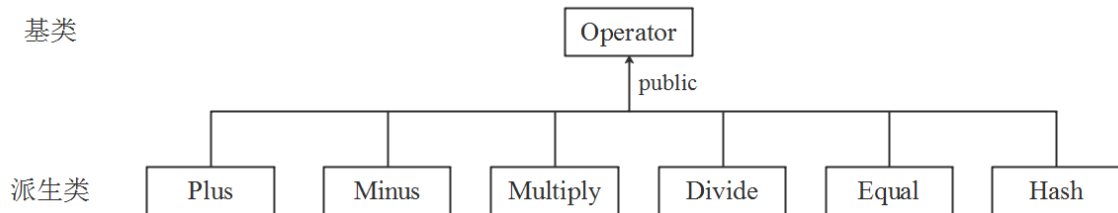
思考:

在前面章节，利用链栈实现了一个简单的计算器程序。经过学习本章节后，如何利用 OOP 思想重新设计与实现计算机程序？



定义运算符基类

把每一种运算符抽象成一个类，再把运算符的**共有属性抽象**出来，形成一个公共基类 Operator。运算符继承关系如下：



9.3 虚函数与多态性—再探计算器

定义运算符基类

```
class Operator{
public:
    Operator(char c, int num0prd, int pre) :m_symbol(c), m_num0prand(
        num0prd), m_precedence(pre){}
    char symbol() const { return m_symbol; }
    int num0prand() const { return m_num0prand; }
    int precedence() const { return m_precedence; };
    virtual double get(double a, double b) const = 0;
    virtual ~Operator() {}
protected:
    const char m_symbol;           //符号
    const int m_num0prand;         //目数
    const int m_precedence;        //优先级
};
```

9.3 虚函数与多态性—再探计算器

定义运算符类

```
class Plus : public Operator{           //运算符 +
public:
    Plus() :Operator('+', 2, 2) {}
    double get(double a, double b) const {
        return a + b; }
};

class Minus :public Operator{           //运算符 -
public:
    Minus() :Operator('-', 2, 2) {}
    double get(double a, double b) const {
        return a - b; }
};

class Multiply :public Operator{        //运算符 *
public:
    Multiply() :Operator('*', 2, 3) {}
    double get(double a, double b) const {
        return a * b; }
};
```

定义运算符类

```
class Divide :public Operator{          //运算符 /
public:
    Divide() :Operator('/', 2, 3) {}
    double get(double a, double b) const {
        return a / b; }
};

class Hash :public Operator{            //运算符 #
public:
    Hash() :Operator('#', 1, 1) {}      //无实际意义
    double get(double a, double b) const {
        return a; }
};

class Equal :public Operator{           //结束符 =
public:
    Equal() :Operator('=', 2, 0) {}    //无实际意义
    double get(double a, double b) const {
        return a; }
};
```

9.3 虚函数与多态性—再探计算器

定义计算器类-通过运算符栈保存基类指针实现动态绑定

```
class Calculator {  
private:  
    Stack<double> m_num;           //操作数栈  
    Stack<unique_ptr<Operator>> m_opr; //运算符数栈, 利用智能指针管理内存  
    void calculate();  
    //成员函数readNum和isNum与前面章节定义的相同  
public:  
    Calculator(){  
        m_opr.push(make_unique<Hash>()); //调用移动push函数  
    }  
    double doIt(const string &exp);  
};
```

9.3 虚函数与多态性—再探计算器

定义计算器类

```
void Calculator::calculate(){ //操作数出栈并进行相应计算
    double a[2] = {0};
    for (auto i = 0; i < m_opr.top()->numOprand(); ++i) {
        a[i] = m_num.top();
        m_num.pop();
    }
    m_num.push(m_opr.top()->get(a[1],a[0])); //触发动态绑定，并将计算结果压栈
    m_opr.pop();
}
```

9.3 虚函数与多态性—再探计算器

定义计算器类

```
double Calculator::dolt(const string &exp){
    for (auto it = exp.begin(); it != exp.end();){
        if (isNum(it))
            m_num.push(readNum(it));
        else{
            char o = *it++;
            unique_ptr<Operator> oo; //定义基类指针
            if (o == '+')
                oo = make_unique<Plus>();
            else if (o == '-')
                oo = make_unique<Minus>();
            else if (o == '*')
                oo = make_unique<Multiply>();
            else if (o == '/')
                oo = make_unique<Divide>();
```

```
            else if (o == '=')
                oo = make_unique<Equal>();
            while (oo->precedence()<=m_opr.top()->
                precedence()){
                if (m_opr.top()->symbol() == '#')
                    break;
                calculate();
            }
            if ( oo->symbol() != '=')
                m_opr.push(std::move(oo));
        }
    }
    double result = m_num.top();
    m_num.pop();
    return result;
}
```

增加移动语义

由于 `unique_ptr` 不支持复制操作，因此向前面章节定义的 `Node` 模板和 `Stack` 模板分别添加支持移动语义的构造函数和 `push` 函数：

```
template<typename T>                // 含右值形参的移动构造函数
Node<T>::Node(T &&val) :m_value(std::move(val)) { }
```

```
template<typename T>
void Stack<T>::push(T &&val) { //含右值形参的push函数
    Node<T> *node = new Node<T>(std::move(val));
    node->m_next = m_top;
    m_top = node;
}
```

本章结束

上机作业

- ① 实验指导书：第九章
- ② 检查日期：
- ③ 地点：与助教协商