@阿西拜-南昌

# 面向对象与程序设计

C++ Primer第五版　　第15章

1

# OOP：概述



面向对象程序设计（object-oriented programming）的核心思想：

- 数据抽象：接口与实现分离
- 继承：定义相似的类，并对其相似关系建模
- 动态绑定：忽略相似类的区别，以同一的方式使用它们

交通工具

通过继承（inheritance），联系在一起的类构成一种层次关系

- 基类（base class）：定义共同拥有的成员
- 派生类（derived class）：定义特有的成员
- 虚函数（virtual function）：基类希望派生类各自定义自己合适的版本

```
class Quote{
public:
	std::string isbn() const;
	virtual double net_price(std::size_t n) const;
};

//派生类必须通过使用类派生列表明确指出基类
class Bulk_quote:public Quote {
public:
	double net_price(std::size_t) const override;
};
```

动态绑定（dynamic binding），我们能用同一段代码分别处理派生类和基类

```
//计算并打印销售给定数量的某种书籍所得的费用
double print_total(ostream &os,const Quote &item, size_t n)
{
	//根据传入item形参的对象类型调用Quote::net_price
	//或者Bulk_quote::net_price
	double ret = item.net_price(n);
	os<<"ISBN:"<<item.isbn() //调用Quote::isbn
		<<"#sold: "<<n<<" total due: "<<ret<<endl;
	return ret;
}
//basic的类型是Quote;bulk的类型是Bulk_quote
print_total(cout,basic,20);
print_total(cout,bulk,20);
```

使用基类的引用（或指针）调用一个虚函数时将发生动态绑定（也叫运行时绑定：run-time binding）

# 定义基类和派生类



首先完成Quote类的定义

```
class Quote{
public:
    Quote()=default;
    Quote(const std::string &book, double sales_price):
        bookNo(book),price(sales_price){}
    std::string isbn() const { return bookNo; }
    //返回给定数量的书籍的销售总额
    //派生类负责改写并使用不同的泽口计算算法
    virtual double net_price(std::size_t n) const
        { return n*price; }
    virtual ~Quote()=default;
private:
    std::string bookNo;
protected:
    double price = 0.0; //代表普通状态下不打折的价格
}
```

基类通常都应该定义一个虚析构函数，即使该函数不执行任何实际操作也是如此

派生类需要访问的基类（受保护的）成员

定义Bulk_quote类

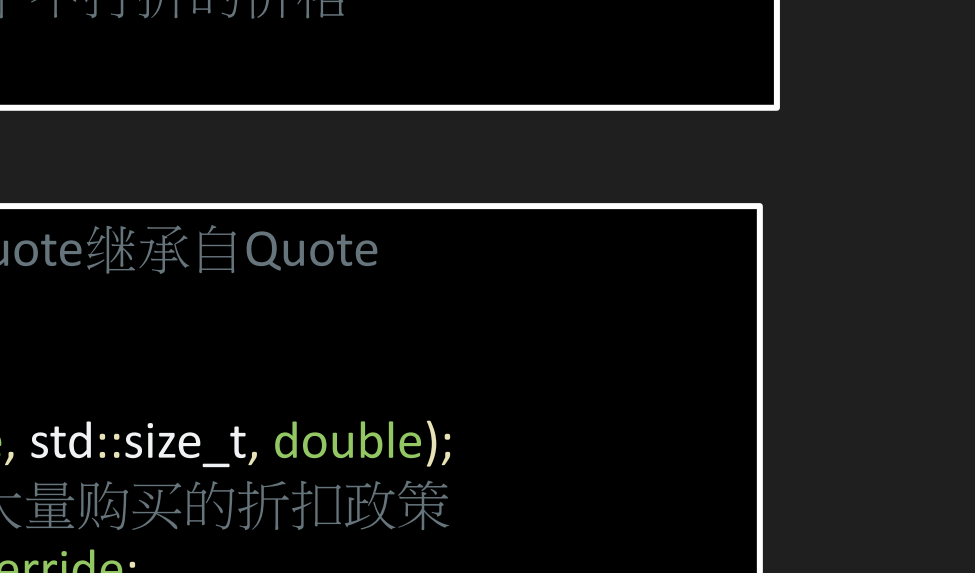
```
class Bulk_quote:public Quote { //Builk_quote继承自Quote
public:
    Bulk_quote()=default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    //覆盖基类的函数版本以实现基于大量购买的折扣政策
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0; //适用折扣政策的最低购买量
    double discount = 0.0; //以小数表示的折扣额
};
```

派生类经常（但不总是）覆盖它继承的虚函数

派生类对象及派生类向基类的类型转换

```
Quote item;         //基类对象
Bulk_quote bulk; //派生类对象
Quote *p = &item;
p = &bulk;          //指向bulk的Quote部分
Quote &r = bulk; //绑定到Quote部分
```

概念模型，而非物理模型




可以把派生类对象的指针用在需要基类指针的地方

派生类必须使用基类的构造函数来初始化继承来的成员

```
Bulk_quote(const std::string& book, double p,
                  std::size_t qty, double disc):
                  Quote(book, p), min_qty(qty), discount(disc){ }
```

首先初始化基类的部分，然后按声明的顺序依次初始化派生类的成员

派生类使用基类的成员：派生类可以访问基类的公有成员和受保护成员

```
//如果达到了购买书籍的某个最低限量值，就可以享受折扣了
double Bulk_quote::net_price(size_t cnt) const
{
      if(cnt>=min_qty)
            return cnt*(1-discount)*price;
      else
            return cnt*price;
}
```

继承与静态成员

```
class Base {
public:
      static void statmem();
};
class Derived:public Base{
      void f(const Derived&);
};

void Derived::f(const Derived &derived_obj)
{
      Base::statmem();
      Derived::statmem();
      derived_obj.statmem();
      statmem();  //通过this对象访问
}
```

派生类的声明

```
class Bulk_quote:public Quote;      //错误
class Bulk_quote;                   //正确
```

一个类的基类，同时也可以是一个派生类



```
class Base { /* ... */ };
class D1:public Base { /* ... */ };
class D2:public D1 { /* ... */ };
```

防止继承发生

```
class NoDerived final { /* */}; //不能作为基类
class Base { /* */};
//Last是final的；我们不能继承Last
class Last final:Base { /* */};
class Bad:Noderived { /* */};          //错误
class Bad2:Last { /* */};              //错误
```

使用基类的引用（或指针）时，实际上我们并不清楚所绑定对象的真实类型

静态类型（static type）：编译时已知

动态类型（dynamic type）：运行时才可知

```
//计算并打印销售给定数量的某种书籍所得的费用
double print_total(ostream &os,const Quote &item, size_t n)
{
    //根据传入item形参的对象类型调用Quote::net_price
    //或者Bulk_quote::net_price
    double ret = item.net_price(n);
    os<<"ISBN:"<<item.isbn() //调用Quote::isbn
        <<"#sold: "<<n<<" total due: "<<ret<<endl;
    return ret;
}
//basic的类型是Quote;bulk的类型是Bulk_quote
print_total(cout,basic,20);
print_total(cout,bulk,20);
```

静态类型为Quote，动态类型依赖于实参


不存在从基类向派生类的隐式类型转换… …

```
Quote base;
Bulk_quote* bulkP = &base; //错误
Bulk_quote& bulkRef = base; //错误

Bulk_quote bulk;
Quote *itemP = &bulk;      //正确
Bulk_quote *bulkP =  itemp;//错误：编译器只能通过检验静态类型来推断
```

可以通过dynamic_cast或static_cast进行转换

在对象间不存在类型转换

```
Bulk_quote bulk;            //派生类对象
Quote item(bulk);           //使用Quote::Quote(const Quote&)构造函数
item = bulk;                //代用Quote::operator=(const Quote&)
```

## 虚函数



虚函数的调用可能在运行时才被解析

```
Quote base("0-201-1",50);
print_total(cout,base,10);
Bulk_quote derived("0-201-1",50,5,.19);
print_total(cout,derived,10);

base = derived;                 //把derived的quote部分拷贝给base
base.net_price(20);             //调用Quote::net_price
```

基类中的虚函数在派生类中隐含地也是一个虚函数。
该函数在基类中的形参必须与派生类中的形严格匹配

final和override说明符

```
struct B{
        virtual void f1(int) const;
        virtual void f2();
        void f3()
};
struct D1:B{
        void f1(int) const override;        //正确
        void f2(int) override;              //错误
        void f3() override;                 //错误
        void f4() override;                 //错误
};

struct D2:B{
        //从B继承f2()和f3(),覆盖f1(int)
        void f1(int) const final; //不允许后续的其他类覆盖f1(int)
};
struct D3:D2{
        void f2();                  //正确
        void f1(int) const;         //错误，在D2中以声明为final
};
```

如果虚函数使用默认实参，基类和派生类中定义的默认实参最好一致

回避虚函数的机制

```
//强制调用基类中定义的函数版本而不管baseP的动态类型到底是什么
double undiscounted = baseP->Quote::net_price(42);
//该调用将在编译时完成解析
```

## 抽象基类



在声明语句的分号之前书写=0，可以定义为纯虚函数

```
//用于保存折扣值和购买量的类
class Disc_quote:public Quote {
public:
	Disc_quote() = default;
	Disc_quote( const std::string& book, double price,
		std::size_t qty, double disc):
		Quote(book,price),quantity(qty),discount(disc){ }
	double net_price(std::size_t) const = 0;
protected:
	std::size_t quantity = 0; //折扣适用的购买量
	double discount = 0.0; //折扣
};
```

可以为纯虚函数提供定义，但必须定义在类的外部

含有（或未经覆盖直接继承）纯虚函数的类是抽象基类

```
//Disc_quote声明了纯虚函数，而Bulk_quote将覆盖该函数
Disc_quote discounted; //错误
Bulk_quote bulk; //正确
```

不能创建抽象基类的对象

派生类构造函数只初始化它的直接基类

```
class Bulk_quote:public Disc_quote{
public:
	Bulk_quote() = default;
	Bulk_quote(const std::string& book, double price,
		std::size_t qty, double disc):
		Disc_quote(book,price,qty,disc) { }
	//覆盖基类中的函数版本以现实一种新的折扣策略
	double net_price(std::size_t) const override;
};
```

## 访问控制与继承



控制其成员对于派生类来说是否可以访问

```
class Base {
protected:
        int prot_mem; //protected成员
};
class Sneaky:public Base {
        friend void clobber(Sneaky&);  //能访问Sneaky::prot_mem
        friend void clobber(Base&);    //不能访问Base::prot_mem
        int j;
};
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }       //正确
void clobber(Base &b) { b.prot_mem = 0; }               //错误
```

派生类友元对于一个基类对象中的受保护成员没有任何访问特权

某个类对其继承来的成员的访问权限受到两个因素的影响：

- 在基类中该成员的访问说明符
- 在派生类的派生列表中的访问说明符

```
class Base {
public:
        void pub_mem();
protected:
        int prot_mem;
private:
        char priv_mem;
};
struct Pub_Derv:public Base{
        int f() { return prot_mem; } //正确
        int g() { return priv_mem; } //错误
};
struct Priv_Derv:private Base {//private不影响派生类的访问权限
        int f1() const { return prot_mem; }
}
//派生访问说明符的目的是控制派生类用户对于基类成语的访问权限
Pub_Derv d1;
Priv_Derv d2;
d1.pub_mem();
d2.pub_mem();//错误：pub_mem在派生类中是private的
//派生访问说明符还可以控制继承自派生类的新类的访问权限
struct Derived_from_Public:public Pub_Derv{
        int use_base() {return prot_mem;}
};
struct Derived_from_Private:public Priv_Derv{
        int use_base() {return prot_mem; } //错误
}
```

仍然是受保护的

## 友元与继承：友元关系不能继承

```
class Base {
        //添加friend声明，其他成员与之前的版本一致
        friend class Pal; //Pal在访问Base的派生类时不具有特殊性
};
class Pal{
public:
        int f(Base b) { return b.prot_mem:} //正确
        int f2(Sneaky S) { return s.j; }; //错误：Paul不是Sneaky的友元
        //对基类的访问权限由基类本身控制
        //即使对于派生类的基类部分也是如此
        int f3(Sneaky s) { return s.prot_mem; } //正确：虽然看上去有些奇怪
};
```

Pal能够访问Base的成员，这种访问包括了Base对象内嵌在其派生类对象中的情况

```
//D2对Base的protected和private成员不具有特殊的访问能力
class D2:public Pal {
public:
        int mem(Base b)
                { return b.prot_mem; } //错误:友元关系不能继承
};
```

通过使用using改变个别成员的可访问性

```
class Base{
public:
        std::size_t size() const { return n; }
protected:
        std::size_t n;
};
class Derived:private Base { //注意：private继承
public:
        //保持对象尺寸相关的成员的访问级别
        using Base::size;
protected:
        using Base::n;
};
```

派生类只能为那些它可以访问的名字提供using声明

默认的继承保护级别

```
class Base { /* ... */ };
struct D1 : Base { /* ... */ };          //默认public继承
class D2 : Base { /* ... */ };           //默认private继承
```

## 继承中的类作用域



如果一个名字在派生类的作用域内无法解析，则编译器将继续在外层的基类作用域中寻找该名字的定义

```
Bulk_quote bulk;
cout << bulk.isbn();
```

1. 在Bulk_quote中查找isbn失败
2. 在Disc_quote中查找失败
3. 在Quote中查找成功

静态类型决定了哪些成员是可见的

```
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
        { return { quantity, discount }; }
    //其他成员与之前版本一致
};

Bulk_quote bulk;
Bulk_quote *bulkP = &bulk;          //静态类型与动态类型一致
Quote *itemp = &bulk;               //静态类型与动态类型不一致
bulkP->discount_policy();           //正确
itemP->discount_policy();           //错误：itemP的类型是Quote*
```

内层作用域（派生类）的名字隐藏定义在外层作用域（基类）的名字

```
struct Base {
    Base():mem(0) { }
protected:
    int mem;
};

struct Derived:Base {
    Derived(int i):mem(i) { }       //用i初始化Derived::mem
                                    //Base::mem进行默认初始化
    int get_mem() { return mem; } //返回Derived::mem
protected:
    int mem; //隐藏基类中的mem
};
```

派生类的成员将隐藏同名的基类成员

不会重载，只会隐藏

```
struct Base {
        int memfcn();
};
struct Derived:Base {
        int memfcn(int); //隐藏基类的memfcn
};
Derived d; Base b;
b.memfcn();         //调用Base::memfcn
d.memfcn(10);       //调用Derived::memfcn
d.memfcn();         //错误：被隐藏了无法调用
d.Base::memfcn(); //正确
```

虚函数的作用域

虚函数不会被隐藏

```
class Base {
public:
        virtual int fcn();
};
class D1:public Base{
public:
        //隐藏基类的fnc，这个fcn不是虚函数
        //D1继承了Baes::fnc()的定义
        int fcn(int);                   //形参列表与Base中的fcn不一致
        virtual void f2();              //是一个新的虚函数，在Base中不存在
};
class D2:public D1{
        int fcn(int);   //是一个非虚函数，隐藏了D1::fcn(int)
        int fnc();      //覆盖了Base的虚函数fcn
        void  f2();     //覆盖了D1的虚函数f2
};
Base bobj; D1 d1obj; D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn(); //虚调用，将在运行时调用Base::fcn
bp2->fcn(); //虚调用，将在运行时调用Base::fcn
bp3->fcn(); //虚调用，将在运行时调用D2::fcn

D1 *d1p = &d1obj; D2 *d2p = &d2obj;
bp2->f2(); //错误：Base没有名为f2的成员
d1p->f2(); //虚调用，将在运行时调用D1::f2()
d2p->f2(); //虚调用，将在运行时调用D2::f2()

Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42); //错误：Base中没有一个接受一个int的fcn
p2->fcn(42); //静态绑定，调用D1::fcn(int)
p3->fcn(42); //静态绑定，调用D2::fcn(int)
```

## 构造函数与拷贝控制



通过在基类中将析构函数定义成虚函数以确保执行正确的析构函数版本：

```
class Quote {
public:
    //删除一个指向派生类对象的基类指针，则需要虚构函数
    virtual ~Quote() = default; //动态绑定析构函数
};

//基类的析构函数是虚函数，delete基类指针将运行正确的析构函数
Quote *itemP = new Quote;          //静态类型与动态类型一致
delete itemP;                      //调用Quote的析构函数
itemP = new Bulk_quote;            //静态类型与动态类型不一致
delete itemP;                      //调用Bulk_quote的析构函数
```

析构函数的属性会被继承，Quote的派生类的析构函数都将是虚函数

合成拷贝控制与继承

- 合成的Bulk_quote默认构造函数运行Disc_quote的默认构造函数，后者又运行Quote的默认构造函数。
- Quote的构造函数完成后，继续执行Disc_quote的构造函数。
- Disc_quote的构造函数完成后，继续执行Bulk_quote的构造函数。

派生类中删除的拷贝控制与基类的关系

```
class B{
public:
    B();
    B(const B&) = delete;
    //其他成员，不含有移动构造函数
};
class D : public B {
    //没有声明任何构造函数
};
D d;                //正确：D的合成默认构造函数使用B的默认构造函数
D d2(d);            //错误：D的合成拷贝构造函数是被删除的
D d3(std::move(d)); //错误：隐式地使用D的被删除的拷贝构造函数
```

移动操作与继承

如果定义了一个移动构造函数/或一个移动赋值运算符，则该类的合成拷贝构造函数和拷贝赋值运算符被定义为删除的

```
class Quote{
public:
    Quote() = default;                          //对成员依次进行默认初始化
    Quote(const Quote&) = default;              //对成员依次拷贝
    Quote(Quote&&) = default;                   //对成员依次拷贝
    Quote& operator=(const Quote&) = default; //拷贝赋值
    Quote& operator=(Quote&&) = default; //移动赋值
    virtual ~Quote() = default;
    //其他成员与之前的版本一致
};
```

派生类的拷贝控制成员

```
class Base { /* ... */};
class D:public Base {
public:
    //默认情况下，基类的默认构造函数初始化对象的基类部分
    //要想使用拷贝或移动构造函数，必须在构造函数初始值列表中
    //显式调用该构造函数
    D(const D& d): Base(d) //拷贝基类成员
    /*D的成语的初始值*/{/*...*/}
    D(D&& d):Base(std::move(d))//移动基类成语
    /*D的成语的初始值*/{/*...*/}
};
```

派生类赋值运算符

```
//Base::operator=(const Base&)不会自动被调用
D &D::operator=(const D &rhs){
    Base::operator=(rhs); //为基类部分赋值
    //按照过去的方式为派生类的成员赋值
    //酌情处理自赋值及释放已有资源等情况
    return *this;
}
```

派生类析构函数

```
class D:public Base{
public:
    //Base::~Base被自动调用执行
    ~D(){  /*该处由用户定义清除派生类成员的操作*/}
};
```

构造函数或析构函数调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对于的虚函数版本。

当基类构造函数调用虚函数的派生类版本时，会发生什么情况？

- 这个虚函数可能会访问派生类的成员。
- 然而，当执行基类型构造函数时，派生类成员尚未初始化。

继承的构造函数

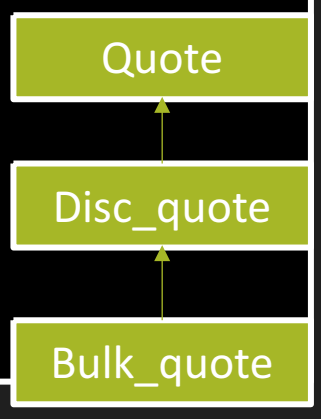
```
class Bulk_quote:public Disc_quote{
public:
    using Disc_quote::Disc_quote; //继承Disc_quote的构造函数
    double net+price(std::size_t) const;
};
//等价于
Bulk_quote(const std::string& book, double price,
    std::size_t qty, double disc):
Disc_quote(book, price, qty, disc) { }
```

## 容器与继承



使用容器存放继承体系中的对象时，通常采用间接存储的方式

```
vector<Quote> basket;
basket.push_back(Quote("0-201-1",50);
//正确：但是只能把对象的Qutoe部分拷贝给basket
basket.push_back(Bulk_quote("0-201-8",50,10,.25));
//调用Quote定义的版本，打印750，即15*$50
cout<<basket.back().net_price(15)<<endl;
```




在容器中放置（智能）指针而非对象

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-1",50));
basket.push_back(make_shared<Bulk_quote>("0-201-8",50,10,.25));
//调用Bulk_quote定义的版本；打印562.5
cout<<basket.back()->net_price(15)<<endl;
```

编写Basket类：表示购物篮的类

```
class Basket {
public:
    //Basket使用合成的默认构造函数和拷贝控制成员
    void add_item(const std::shared_ptr<Quote> &sale)
        { items.insert(sale); }
    //打印每本书的总价和购物篮中所有书的总价
    double total_receipt(std::ostream&) const;
private:
    //该函数用于比较shared_ptr,multiset成员会用到它
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
        { return lhs->isbn() < rhs->isbn(); }
    //multiset保存多个报价，按照compare成员排序
    std::multiset<std::shared_ptr<Quote>,decltype(compare)*>
            items{compare};
};
```

## 定义Basket的成员

```
double Basket::total_receipt(ostream &os) const
{
	double sum = 0.0; //保存实时计算出的总价格
	//iter指向ISBN相同的一批元素中的第一个
	//upper_bound返回一个迭代器，该迭代器指向这批元素的尾后
	for(auto iter = items.cbegin(); iter != items.cend();
				iter = items.upper_bound(*iter)) {
		//我们知道当前的Basket中至少有一个该关键字的元素
		//打印该书籍对于的项目
		sum+=print_total(os,**iter,items.count(*iter));
	}
	os <<"Total Sale:"<<sum<<endl; //打印最终的总价格
	return sum;
}
```

## 隐藏指针

```
Basket bsk;
bsk.add_item(make_shared<Quote>("123",45));
bsk.add_item(make_shared<Bulk_quote>("345",45,3,.15));

//为了实现上面的功能，需要给Quote类添加一个虚函数
class Quote{
public:
	//该虚函数返回当前对象的一份动态分配的拷贝
	virtual Quote* clone() const & { return new Quote(*this);}
	virtual Quote* clone() && { return new Quote(std::move(*this));}
	//其他成员与之前的版本一致
};
class Bulk_quote:public Quote{
	Bulk_quote* clone() const & { return new Bulk_quote(*this); }
	Bulk_quote* clone() && { return new Bulk_quote(std::move(*this)); }
	//其他成员与之前的版本一致
};

class Basket {
public:
	void add_item(const Quote& sale) //拷贝给定的对象
		{ item.insert(std::shared_ptr<Quote>(sale.clone())); }
	void add_item(Quote&& sale) //移动给定的对象
		{ items.insert(std::shared_ptr<Quote>(std::move(sale).clone())); }
	//其他成员与之前的版本一致
};
```

```
//重新定义add_item，使得它接受一个Quote对象而非指针
void add_item(const Quote& sale); //拷贝给定的对象
void add_item(Quote&& sale); //移动给定的对象
```

# 文本查询程序再探



```
1  Alice Emma has long flowing red hair.
2  Her Daddy says when the wind blows
3  through her hair, it looks almost alive,
4  like a fiery bird in flight.
5
6  A beautiful fiery bird, he tells her,
7  magical but untamed.
8  "Daddy, shush, there is no such thing,"
9  she tells him, at the same time wanting
10 him to tell her more.
   Shyly, she asks, "I mean, Daddy, is there?"
```

我们的系统将支持如下查询形式：

- 单词查询，用于得到匹配某个给定 string 的所有行：

```
Executing Query for: Daddy
Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"
```

- 逻辑非查询，使用~运算符得到不匹配查询条件的所有行：

```
Executing Query for: ~(Alice)
~(Alice) occurs 9 times
...
```

- 逻辑或查询，使用 | 运算符返回匹配两个条件中任意一个的行：

```
Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,
```

- 逻辑与查询，使用&运算符返回匹配全部两个条件的行：

```
Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.
```

并且支持混用：

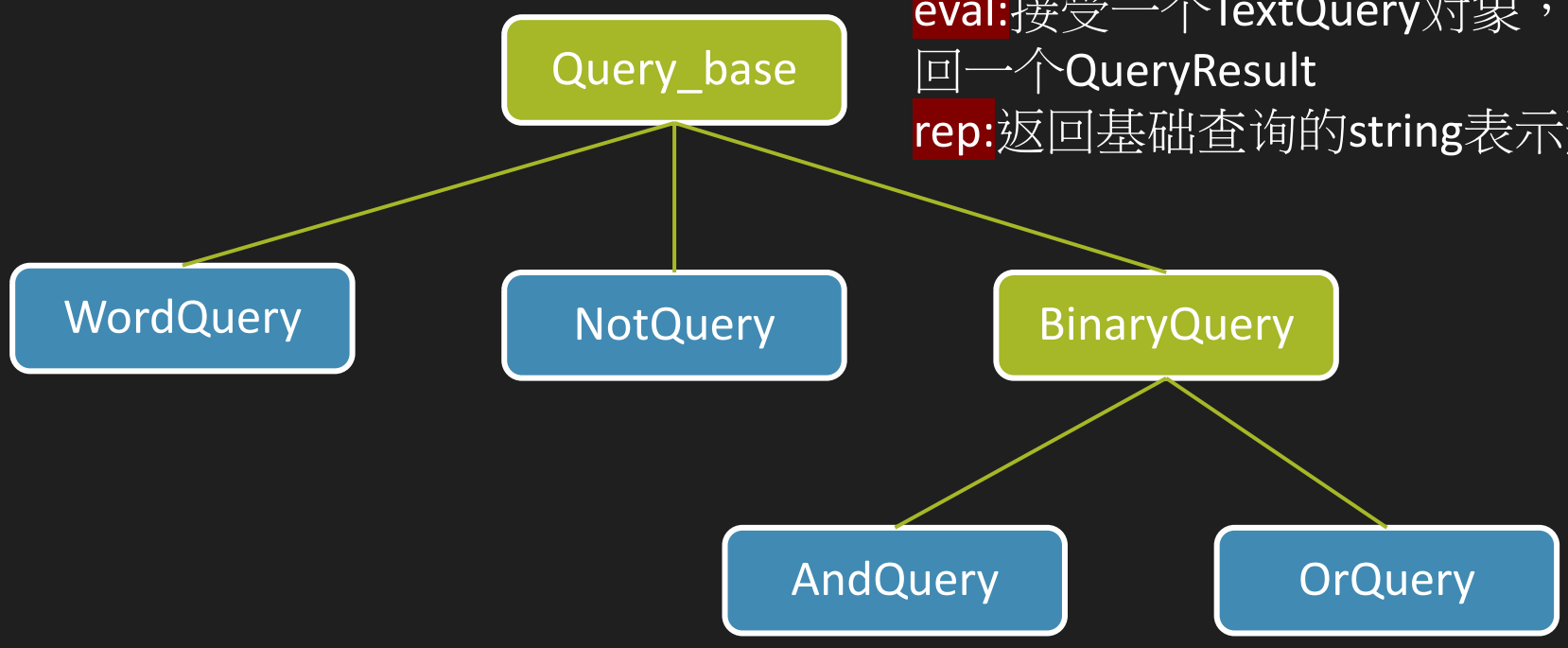
```
Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,
```

## 数据结构：

| | |
|---|---|
| TextQuery | • 包含一个vecotr<string>：保存输入文本<br>• 包含一个map<string,set<ine_no>>：关联单词和行号 |
| QueryResult | 保存查询结果、print函数 |

## 在类直接共享数据：

| | |
|---|---|
| TextQuery | 两个类共享了数据，使用shared_ptr来反映数据结构中的这种共享关系 |
| QueryResult | |

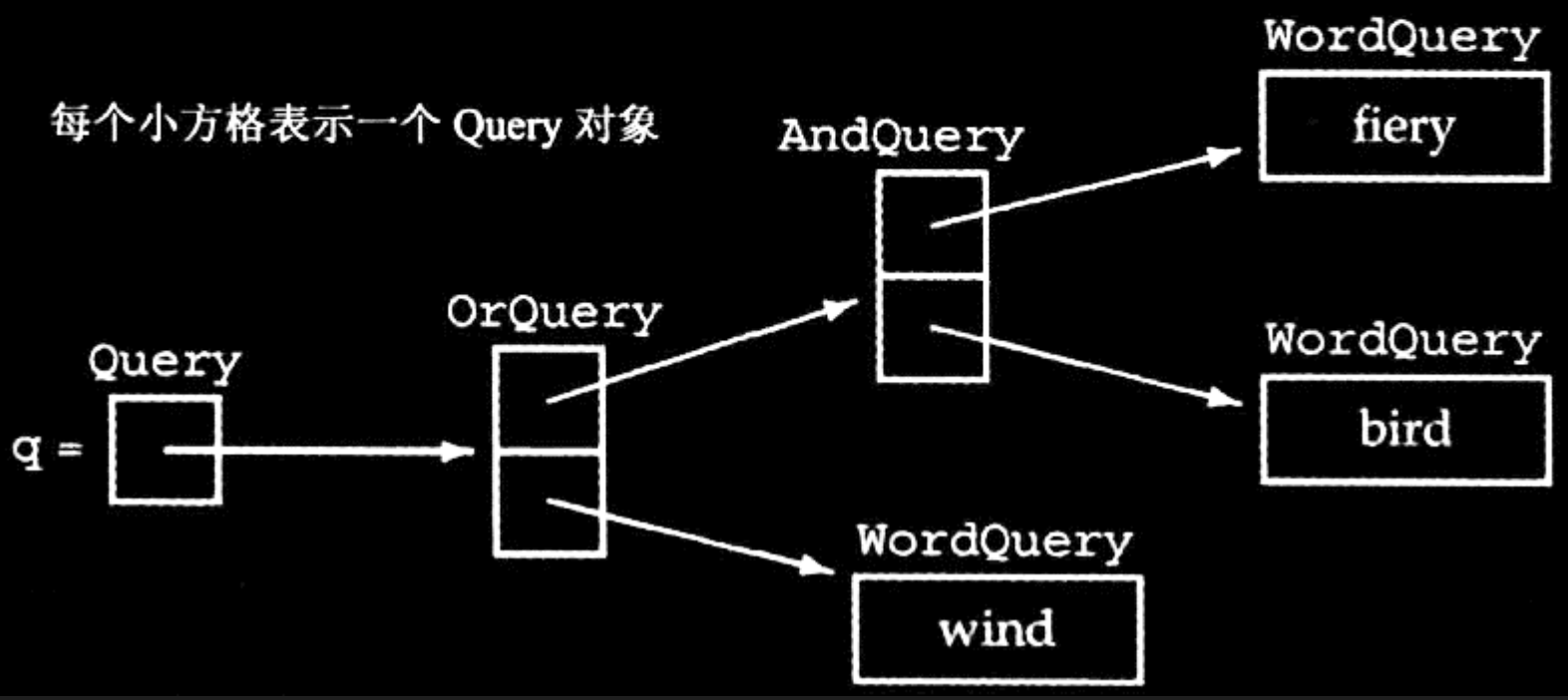



eval:接受一个TextQuery对象，并返回一个QueryResult
rep:返回基础查询的string表示形式

将层次关系隐藏于接口类中

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

- &运算符生成一个绑定到新的 AndQuery 对象上的 Query 对象；
- |运算符生成一个绑定到新的 OrQuery 对象上的 Query 对象；
- ~运算符生成一个绑定到新的 NotQuery 对象上的 Query 对象；
- 接受 string 参数的 Query 构造函数生成一个新的 WordQuery 对象。



## 概述：Query程序设计

| Query 程序接口类和操作 | |
|---|---|
| TextQuery | 该类读入给定的文件并构建一个查找图。这个类包含一个 query 操作，它接受一个 string 实参，返回一个 QueryResult 对象；该 QueryResult 对象表示 string 出现的行 |
| QueryResult | 该类保存一个 query 操作的结果 |
| Query | 是一个接口类，指向 Query_base 派生类的对象 |
| Query q(s) | 将 Query 对象 q 绑定到一个存放着 string s 的新 WordQuery 对象上 |
| q1 & q2 | 返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 AndQuery 对象上 |
| q1 \| q2 | 返回一个 Query 对象，该 Query 绑定到一个存放 q1 和 q2 的新 OrQuery 对象上 |
| ~q | 返回一个 Query 对象，该 Query 绑定到一个存放 q 的新 NotQuery 对象上 |
| **Query 程序实现类** | |
| Query_base | 查询类的抽象基类 |
| WordQuery | Query_base 的派生类，用于查找一个给定的单词 |
| NotQuery | Query_base 的派生类，查询结果是 Query 运算对象没有出现的行的集合 |
| BinaryQuery | Query_base 派生出来的另一个抽象基类，表示有两个运算对象的查询 |
| OrQuery | BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的并集 |
| AndQuery | BinaryQuery 的派生类，返回它的两个运算对象分别出现的行的交集 |

Query_base类和Query类

@阿西拜-南昌

```
//抽象基类，具体的查询类型从中派生，所有成员都是private的
class Query_base{
	friend class Query;
protected:
	using line_no = TestQuery::line_no; //用于eval函数
	virtual ~Query_base() = default;
private:
	//eval返回与当前Query匹配的QueryResult
	virtual QueryResult eval(const TextQuery&) const = 0;
	//rep是表示查询的一个string
	virtual std::string rep() const = 0;
};

//这是一个管理Query_base继承体系的接口类
class Query {
	//这些运算符需要访问接受shared_ptr的构造函数，而该函数是私有的
	friend Query operator~(const Query &);
	friend Query operator|(const Query&, const Query&);
	friend Query operator&(const Query&, const Query&);
public:
	Query(const std::string&); //构建一个新的WordQuery
	//接口函数：调用对应的Query_base操作
	QueryResult eval(const TextQuery &t) const
		{ return q->eval(t); }
	std::string rep() const { return q->rep(); }
private:
	Query(std::shared_ptr<Query_base> query):q(query){ }
	std::shared_ptr<Query_base> q;
};
```

Query的输出运算符

```
std::ostream & operator<<(std::ostream &os, const Query &query)
{
	//Query::rep通过它的Query_base指针对rep()进行了虚调用
	return os<<query.rep();
}

//当我们使用时，代码如下
Query andq = Query(sought1) & Query(sought2);
cout<< andq << endl;
```

WordQuery类：查找给定的string

```
class WordQuery:public Query_base {
	friend class Query; //Query使用WordQuery构造函数
	WordQuery(const std::string &s):query_word(s) { }
	//具体的类：WordQuery将定义所有继承而来的纯虚函数
	QueryResult eval(const TextQuery &t) const
		{return t.query(query_word); }
	std::string rep() const { return query_word; }
	std::string query_word; //要查找的单词
};

//定义了WordQuery类之后，就可以完成Query接受string的构造函数了
inline Query::Query(const std::string &s):q(new WordQuery(s)) { }
```

NotQuery类及~运算符：~运算符生成一个NotQuery（保存着一个需要取反的Query）

```
class NotQuery:public Query_base {
	friend Query operator~(const Query &);
	NotQuery(cosnt Query &q):query(q) { }
	//具体的类：NotQuery将定义所有继承而来的纯虚函数
	std::string rep() const { return "~("+query.rep()+")";}
	QueryResult eval(const TextQuery&) const;
	Query query;
};
inline Query operator~(const Query &operand)
{
	return std::shared_ptr<Query_base>(new NotQuery(operand));
}
```

BinaryQuery类：抽象基类，保存操作两个运算对象的查询类型所需的数据

```
class BinaryQuery:public Query_base{
protected:
	BinaryQuery(const Query &l, const Query &r, std::string s):
	 lhs(l),rhs(r),opSys(s) { }
	//抽象类型：BinaryQuery不定义eval
	std::string rep() const { return "(" + lhs.rep() + " "
					+opSym + " "
					+ rhs.rep() + ")"; }
	Query lhs, rhs; //左侧和右侧运算对象
	std::string opSym; //运算符的名字
};
```

AndQuery类、OrQuery类及相应的运算符

```
class AndQuery:public BinaryQuery {
      friend Query operator&(const Query&, const Query&);
      AndQuery( const Query &left, const Query &right):
                                          BinaryQuery(left,right,"&"){ }
      //具体的类：AndQuery继承了rep并且定义了其他纯虚函数
      QueryResult eval(const TextQuery&) const;
};
inline Query operator&(const Query &lhs, cosnt Query &rhs)
{
      return std::shared_ptr<Query_base>(new AndQuery(lhs,rhs));
}

class OrQuery:public BinaryQuery{
      friend Query operator|(const Query&,const Query&);
      OrQuery(const Query &left, const Query &right):
                                      BinaryQuery(left,right,"|") { }
      QueryResult eval(const TextQuery&) const;
};
inline Query operator|(const Query &lhs, const Query &rhs)
{
      return std::shared_ptr<Query_base>(new OrQuery(lhs,rhs));
}
```

OrQuery::eval

```
//返回运算对象查询结果set的并集
QueryResult OrQuery::eval(const TextQuery& text) const
{
      //通过Query成员lhs和rhs进行的虚调用
      //调用eval返回每个运算对象的QueryResult
      auto right = rhs.eval(text), left = lhs.eval(text);
      //将左侧运算对象的行号拷贝到结果set中
      auto ret_lines = make_shared<set<line_no>>(left,begin(),left.end());
      //插入右侧运算对象所得的行号
      ret_lines->insert(right.begin(),right,end());
      //返回一个新的QueryResult，它表示lhs和rhs的并集
      return QueryResult(rep(),ret_lines,left.get_file());
}
```

AndQuery::eval

```
//返回运算对象查询结果set的交集
QueryResult AndQuery::eval(const TextQuery& text) const
{
	//通过Query运算对象进行的虚调用，获得运算对象的查询结果set
	auto left = lhs.eval(text), right = rhs.eval(text);
	//保存left和right交集的set
	auto ret_lines = make_shared<set<line_no>>();
	//将两个范围的交集写入一个目的迭代器中
	//本次调用的目的迭代器向ret添加元素
	set_intesection(left.begin(),left.end(),
				right.begin(),right.end(),
				inserter(*ret_lines,ret_lines->begin()));
	return QueryResult(rep(),ret_lines,left.get_fine());
};
```

NotQuery::eval

```
//返回运算对象的结果set中不存在的行
QueryResult NotQuery::eval(const TextQuery& text) const
{
	//通过Query运算对象对eval进行虚调用
	auto result = query.eval(text);
	//开始时结果set为空
	auto ret_lines = make_shared<seet<line_n0>>();
	//我们必须在运算对象出现的所有行中进行迭代
	auto beg = result.begin(), end = result.end();
	//对于输入文件的每一行，如果不在result当中，将其添加到ret_lines
	auto sz = result.get_file()->size();
	for(size_t n = 0; n!=sz; ++n){
		//如果我们还没有处理完result的所有行
		//检查当前行是否存在
		if(beg==end || *beg != n)
			ret_lines->insert(n); //如果不在result当中，添加这一行
		else if(beg != end)
			++beg;  //否则继续获取result的下一行（如果有的话）
	}
	return QueryResult(rep(),ret_lines,result.get_file());
}
```