1，OOP：概述

面向对象程序设计的核心思想是数据抽象，继承和动态绑定。

对于某些函数，基类希望它的派生类各自定义适合自身的版本，此时基类就将这些函数声明为虚函数。

virtual double net\_price(std::size\_t n) const;

派生类必须在其内部对所有重载定义的虚函数进行声明。

派生类可以在这样的函数之前加上virtual关键字，但是并不是非得这么做。

C++11新标准允许派生类显示地注明它将使用哪个成员函数改写基类的虚函数，使用ovveride关键字。

在C++语言中，当我们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。

2，定义基类和派生类

2.1 定义基类

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。

任何构造函数之外的非静态函数都可以是虚函数。

关键字virtual只能出现在类内部的声明语句之前而不能用于类外部的函数定义。

如果基类把一个函数声明成虚函数，则该函数在派生类中隐式地也是虚函数。

成员函数如果没有被声明为虚函数，则其解析过程发生在编译时而非运行时。

基类希望它的派生类有权访问该成员，同时禁止其他用户访问，使用protected访问运算符来说明。

2.2 定义派生类

派生类必须使用类派生列表明确指出它是从哪个（哪些）基类继承而来的。

每个基类前面可以有以下三种访问说明符中的一个：public，protected或者private。

派生类必须将其继承而来的成员函数中需要覆盖的那些重新声明。

派生类经常（但不总是）覆盖它继承的虚函数。如果派生类没有覆盖其基类中的某个虚函数，则派生类会直接继承其在基类中的版本。

每个类控制它自己的成员初始化过程。

可以利用基类的构造函数初始化子类：

Bulk\_quote(const string& book, double p, size\_t qty, double disc):

Quote(book, p),min\_qty(qty),discount(disc){}

首先初始化基类的部分，然后按照声明的顺序依次初始化派生类的成员。

派生类可以访问基类的公有成员和受保护成员。

如果基类定义了一个静态成员，则在整个继承体系中存在该成员的唯一定义。无论从基类中派生出来多少个派生类，对于每个静态成员来说都只存在一个实例。

一条声明语句的目的是令程序知晓某个名字的存在以及该名字表示一个什么样的实体，如一个类，一个函数或一个变量等。

派生类的声明，不包含它的派生列表。

如果我们想要将某个类用作基类，则该类必须已经定义而非仅仅声明。

直接基类出现在派生列表中，而间接基类由派生类通过其直接基类继承而来。

在类名后面跟一个关键字final可以防止继承的发生。

class NoDerived final{};

2.3 类型转换与继承

我们可以将基类的指针或引用绑定到派生类对象上。

和内置指针一样，智能指针类也支持派生类类向基类的类型转换。

表达式的静态类型在编译时总是已知的，动态类型则是运行时才知道的。

如果表达式既不是引用也不是指针，则它的动态类型永远与静态类型一致。

不存在从基类向派生类的隐式转换。

在对象之间不存在类型转换。

派生类向基类的自动类型转换只对指针或引用类型有效，在派生类类型和基类类型之间不存在这样的转换。

3，虚函数

所有的虚函数都必须有定义。

动态绑定只有当我们通过指针或引用调用虚函数时才会发生。

在派生类中覆盖某个虚函数时，可以再一次使用virtual关键字指出该函数的性质。然而这么做并非必须。

派生类中虚函数的返回类型也必须与基函数匹配。该规则存在一个例外，当类的虚函数返回类型是类本身的指针或引用时，上述规则无效。

我们可以把某个函数指定为final，则之后的任何常识覆盖该函数的操作都将引发错误。

如果虚函数使用默认实参，则基类和派生类中定义的默认实参最好一致。

某些情况下，我们希望对虚函数的调用不要进行动态绑定，而是强迫其执行虚函数的某个版本。

double undiscounted = baseP->Quote::net\_price(42);

4，抽象基类

和普通的虚函数不一样，一个纯虚函数无须定义。

我们通过在函数体的位置书写=0就可以将一个虚函数说明为纯虚函数。其中=0只能出现在类内部的虚函数说明语句处。

double net\_price(std::size\_t) const = 0;

我们也可以为纯虚函数提供定义，不过函数体必须定义在类的外部。不能在类的内部为一个=0的函数提供函数体。

含有纯虚函数的类是抽象基类。

我们不能创建抽象基类的对象。

每个类各自控制其对象的初始化过程，即使Bulk\_quote没有自己的数据成员，它也人仍然需要像原来一样提供一个接受四个参数的构造函数。

5，访问控制与继承

某个类对其继承而来的成员的访问权限受到两个因素影响：一是在基类中该成员的访问说明符，二是在派生类的派生列表中的访问说明符。

派生访问说明符的目的是控制派生类用户对于基类成员的访问权限。

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。

就像友元关系不能传递一样，友元关系也不能继承。

不能继承友元关系；每个类负责控制各自成员的访问权限。

有时我们需要改变派生类继承的某个名字的访问级别，通过使用using声明。

public:

using Base::size;

派生类只能为那些它可以访问的名字提供using声明。

struct和class关键字定义的类唯一的差别就是默认成员访问说明符及默认派生访问说明符。

6，继承中的类作用域

派生类的成员将隐藏同名的基类成员。

通过作用域运算符来使用隐藏的成员。return Base::mem;

如果派生类的成员与基类的某个成员同名，则派生类将在其作用域内隐藏该基类成员。即使派生类成员和基类成员的形参列表不一致。

基类与派生类中的虚函数必须有相同的形参列表。

假如基类与派生类的虚函数接受的实参不同，则我们就无法通过基类的引用或指针调用派生类的虚函数了。

和其他函数一样，成员函数无论是否是虚函数都能被重载。派生类可以覆盖重载函数的0个或多个实例。

如果派生类希望所有的重载版本对于它来说都是可见的，那么它就需要覆盖所有的版本，或者一个都不覆盖。

一条基类成员函数的using声明语句就可以把函数的所有重载实例添加到派生类作用域中。

7，构造函数与拷贝控制

7.1 虚析构函数

基类通常应该定义一个虚析构函数。

如果基类的析构函数不是虚函数，则delete一个指向派生类对象的基类指针将产生未定义行为。

一个基类总是需要析构函数，而且它将析构函数设定为虚函数。

如果一个类定义了析构函数，即使它通过=default的形式使用了合成的版本，编译器也不会为这个类合成移动操作。

7.2 合成拷贝控制与继承

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数，赋值运算符或析构函数类似，它们对类本身的成员一次进行初始化，赋值或销毁操作。

此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化，赋值或销毁操作。

如果基类的默认构造函数，拷贝构造函数，拷贝控制运算符或析构函数是被删除的函数或者不可访问，则派生类中对应的成员将是被删除的。

原因是编译器不能使用基类成员来执行派生类对象基类部分的构造，赋值或销毁操作。

如果在基类中有一个不可访问或删除掉的析构函数，则派生类中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分。

7.3 派生类的拷贝控制成员

派生类的拷贝和移动构造函数在拷贝和移动自有成员的同时，也要拷贝和移动基类部分的成员。

派生类赋值运算符也必须为其基类部分的成员赋值。

当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。

class D : public Base{

public:

D(const D& d):Base(d)

D(D&& d):Base(std::move(d))

};

在默认情况下，基类默认构造函数初始化派生类对象的基类部分。如果我们想拷贝（或移动）基类部分，则必须在派生类的构造函数初始值列表中显示地使用基类的拷贝（或移动）构造函数。

与拷贝和移动构造函数一样，派生类的赋值运算符也必须显示地为其基类部分赋值。

无论基类的构造函数和赋值运算符是自定义的版本还是合成的版本，派生类的对应操作都能使用它们。

和构造函数及赋值运算符不同的是，派生类析构函数只负责销毁由派生类自己分配的资源。

如果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应的虚函数版本。

7.4 继承的构造函数

一个类只初始化它的直接基类，出于同样的原因，一个类也只能继承其直接基类的构造函数。

类不能继承默认，拷贝和移动构造函数。

派生类继承其基类构造函数的方式是提供一条注明了直接基类名的using声明语句。

using Disc\_quote::Disc\_quote//继承Disc\_quote的构造函数

通常情况下，using声明语句只是令某个命根子在当前作用域内可见。而当作用于构造函数时，using声明语句将令编译器产生代码。

对于基类的某个构造函数，编译器都生成一个与之对应的派生类构造函数。

这些编译器生成的构造函数形如：derived(parms):base(args){}

例子：Bulk\_quote(const string& book,double price,size\_t qty,double disc):Disc\_quote(book, price, qty, disc){}

和普通成员的using声明不一样，一个构造函数的using声明不会改变该构造函数的访问级别。

8，容器与继承

当我们使用容器存放继承体系中的对象时，通常必须采取间接存储的方式。

因为不允许在容器中保存不同类型的元素，所以我们不能把具有继承关系的多种类型的对象直接存放在容器当中。

当派生类对象被赋值给基类对象时，其中的派生类部分将被切掉，因此容器和存在继承关系的类型无法兼容。

在容器中放置（智能）指针而非对象。

当我们希望在容器中存放具有继承关系的对象时，我们实际上存放的通常是基类的指针。（更好的选择是智能指针）

vector<shared\_ptr<Quote>> basket;

basket.push\_back(make\_shared<Quote>("",50))

basket.push\_back(make\_shared<Bulk\_quote>("",50,10,.5))

编程：编写Basket类

9，文本查询程序再探