面向对象程序设计的核心思想：数据抽象、继承、动态绑定。

继承和动态绑定的影响：

1.可以更容易定义与其他类相似但不完全相同的新类。

2.使用彼此相似的类编写程序时，可以在一定程度上忽略掉他们的差别。

类和继承

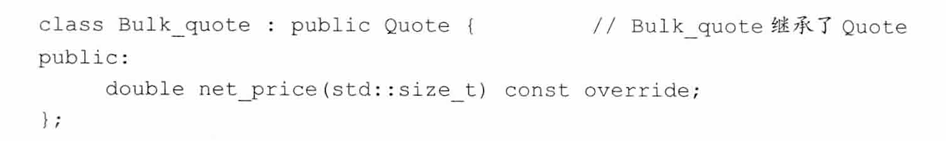
基类位于层次关系的根部，其他类直接或间接地从基类继承而来，继承得到的类称为派生类。基类负责定义所有类共有的成员，派生类定义各自特有的成员。

对于某些成员函数，基类希望它的派生类各自定义适合自身的版本，此时就将这些函数声明为虚函数。（运行时多态）

派生类通过类派生列表来指出它从哪个或哪些基类继承而来。

类派生列表：首先是一个冒号，然后是以逗号分隔的基类列表，每个基类前面可以有访问说明符。

就像下面这样，定义一个基类的子类：



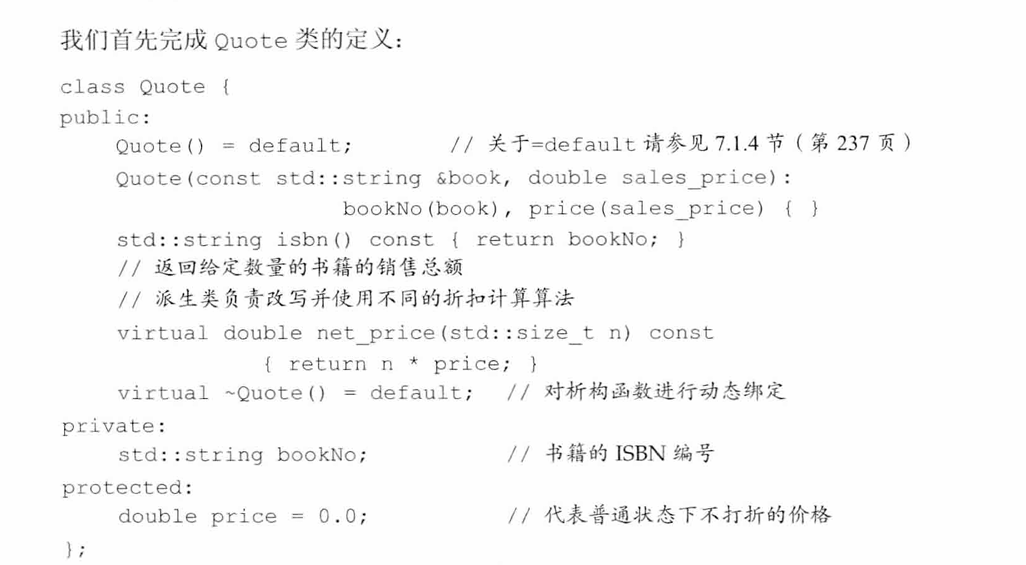
ps：派生类必须在内部对所有重新定义的虚函数进行声明，声明时可以在前面加上 virtual，也可以不加。C++11 允许使用 override 关键字显式地指明重新定义的虚函数，把 override 放到形参列表后面。

动态绑定

当使用基类的引用或指针来调用一个虚函数时将发生动态绑定。

动态绑定的实现机制：动态绑定根据传入的参数类型来选择函数版本（可能是基类中的该函数或派生类中的该函数），它发生在运行时，又称运行时绑定。依赖于虚函数表。

定义一个基类



成员函数与继承

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操作也是如此。

基类通过虚函数区分两种成员函数：

1.基类希望派生类进行覆盖的函数，将其定义为虚函数。构造函数与静态函数都不能定义成虚函数，任何构造函数之外的非静态函数都可以定义为虚函数。

2.基类希望派生类直接继承不要改变的函数。

使用指针或引用调用虚函数时，该调用将被动态绑定。

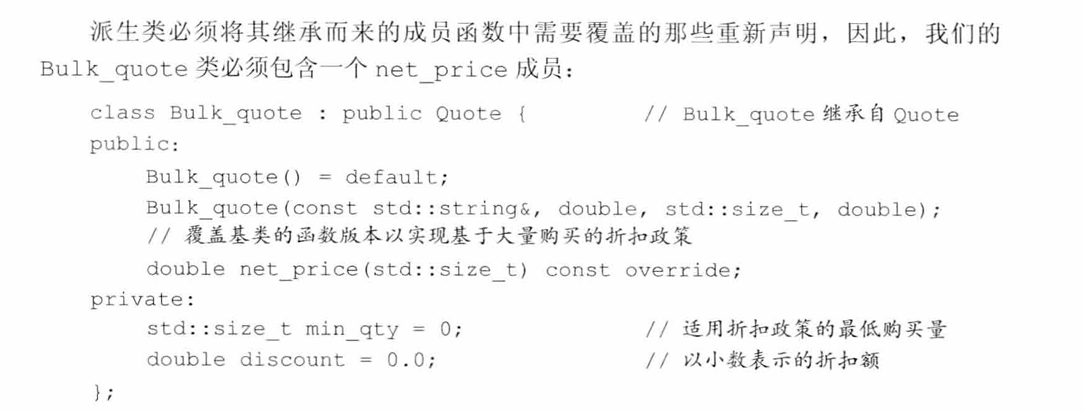
基类通过在其成员函数声明语句前加上virtual关键字使得该函数执行动态绑定/如果基类把一个函数声明为虚函数，该函数在派生类中隐式地也是虚函数。（基类里虚函数必须使用virtual关键字，而派生类了可以不用加virtual关键字）

虚函数的解析过程发生在运行时，普通函数的解析过程发生在编译时。

访问控制与继承

派生类可以继承定义在基类中的成员，但是派生类的成员函数不一定有权访问从基类继承而来的成员——派生类能访问基类的公有成员和受保护成员，不能访问私有成员。

定义一个上面例子基类的派生类



关于派生类中的虚函数

派生类经常覆盖它继承的虚函数。如果没有覆盖，派生类会直接继承其在基类中的版本。

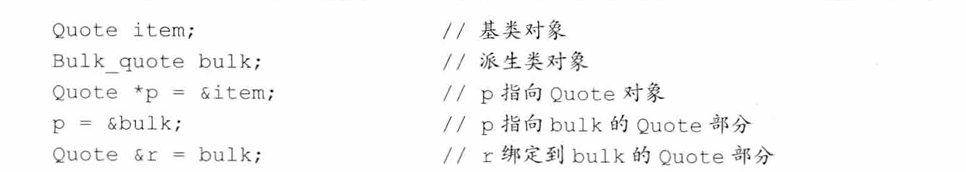
C++允许 override 关键字显式地指明重新定义的虚函数，把 override 放到形参列表后面、或 const 成员函数的 const 关键字后面、或引用成员函数的引用限定符后面。当覆盖一个虚函数时，override实际上可以省略，但是建议仍然加上override关键字，因为使用 override 关键字可以让编译器在编译时进行检查，确保派生类中的成员函数确实重写了基类中的虚函数（即基类中确实有这个虚函数）。如果函数签名不匹配，编译器会给出错误，从而避免运行时抛出异常。

为什么派生类对象可以向基类的类型转换

一个派生类对象有多个组成部分：一个含有派生类自己定义的成员的子对象，一个与该派生类继承的基类对应的子对象。下面是一个示意图：

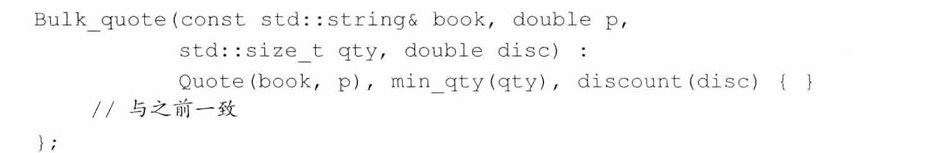


因为派生类对象中含有与基类对应的组成部分，所以可以把派生类的对象当成基类对象来使用，也能把基类的指针或引用绑定到派生类对象中的基类部分上。类似于下面书上的例子：



关于派生类的构造函数

派生类不能直接初始化从基类继承来的成员，而是使用基类的构造函数来初始化它的基类部分。即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。派生类构造函数如下，使用初始化列表：



先调用了父类的构造函数，然后再初始化类内的成员变量。

ps：从通过初始化列表的方式调用父类的构造函数可以看出，派生类构造函数通过构造函数初始化列表将实参传递给基类构造函数。每个类控制自己的成员初始化过程。

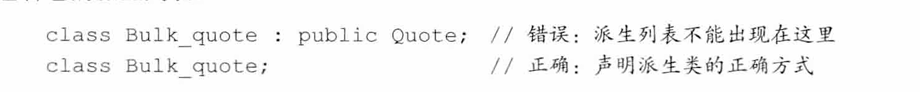
继承关系与类型的静态成员

基类的静态成员只存在唯一一个实例，不论有多少派生类。

如果静态成员是可访问的（public、protected），派生类也能使用它。

派生类的声明

派生类的声明不包含派生列表（即从哪些类和接口派生）



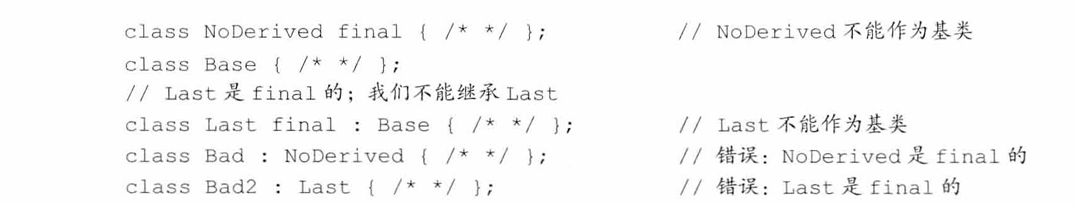
关于被继承的基类

必须完整定义某个类后，该类才能作为基类被其他类继承。不能仅仅是一个声明。

继承可以多重继承，最终的派生类将包含它的直接基类的子对象和每一个间接基类的子对象。

使得一个类不能被继承

使用final关键字，在类名后使用final关键字，下面是书上的例子：



类型转换与继承

可以将基类的指针或引用绑定到派生类对象上有一层重要含义：当使用基类的引用或指针时，实际上我们并不清楚它所绑定对象的真实类型，可能是基类对象也可能是派生类对象。比如内置指针、智能指针类也支持派生类向基类的转换。

派生类向基类的类型转换也可能因为访问受限而不可行。

静态类型指编译时已知对象类型，动态类型指需要运行时确定对象类型。如果表达式既不是指针也不是引用，那么它的动态类型和静态类型永远一致。如果表达式是指针或是引用，静态类型通常是类类型，而动态类型通常是指向类类型、内置类型的指针类型；反之都是类类型或者内置类型。

派生类可以向基类转换是因为派生类对象中包含基类部分，而基类的引用或指针可以绑定到该基类部分上，反过来是不行的。（联想到C#拆箱必须显式转换）

ps：在对象之间不存在类型转换，只有指针和引用可以自动从派生类转化至基类，而非引用和非指针类型不可以，从定义合成拷贝函数、定义合成拷贝运算符等操作可以佐证在对象之间不存在类型转换。

关于虚函数

虚函数必须提供定义，即使没有被用到。因为编译器不知道哪个虚函数在运行时会被使用到。

只有当通过指针或引用调用虚函数时才会发生动态绑定，正常调用虚函数时不会发生动态绑定。

派生类中的虚函数

当在派生类中覆盖了某个虚函数时，可以用 virtual 指明也可以不用。虚函数在所有的派生类中都是虚函数。

如果派生类的函数覆盖了继承而来的虚函数，它的形参类型必须与被覆盖的基类函数完全一致。返回类型也必须相匹配**。**

关于final 和 override 说明符

再次强调两者分别作用：C++11 中可以使用 override 来指明派生类中的虚函数。这时如果该函数没有覆盖基类中的虚函数，编译器就会报错。

可以把某个函数指定为 final，这样该函数就不能被派生类所覆盖。

final 和 override 都写到形参列表和尾置返回类型之后。

虚函数也可以有默认实参，实参值由调用的静态类型决定。即如果通过基类的指针或引用调用虚函数，则使用基类中定义的默认实参。

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。

回避虚函数的默认机制

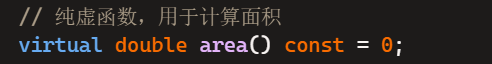
有时希望对虚函数的调用不进行动态绑定，而是强迫执行虚函数的某个特定版本，可以通过作用域运算符来实现。下面是例子：



ps：通常只有在成员函数或友元中的代码才需要使用作用域运算符来回避虚函数的机制。或者当一个派生类的虚函数需要调用它的基类版本时。

抽象基类

可以将一个没有实际意义的虚函数定义为纯虚函数，只需当在类内对它进行声明时最后加一个 =0 即可，无需额外定义。



含有纯虚函数的类是抽象基类。不能直接创建一个抽象基类的对象。抽象基类不能实例化，但是可以有其他的成员方法和变量。

可以创建派生类的对象，前提是派生类覆盖了原本的纯虚函数，否则该派生类也是抽象基类。

访问控制与继承

每个类分别控制自己的成员初始化进程，也控制着自己的成员对于派生类来说是否可访问。

protected受保护的成员

一个类使用 protected 关键字来声明希望对派生类可见但对其他用户不可见的成员。

派生类的成员和友元可以通过派生类对象访问基类的受保护成员，但是不能直接通过基类对象来访问。

公有、私有和受保护继承

一个类对继承的基类成员的访问权限受两方面影响：

1.基类中该成员的访问说明符

2.派生类的派生列表中的访问说明符

派生列表中的访问说明符不会影响派生类自身的成员和友元对基类的访问权限，对直接基类的访问权限只与基类中的访问说明符有关。它影响的是派生类的用户（包括派生类的对象、派生类的派生类）对基类成员的访问权限。

访问说明符在继承中的作用：

1.如果继承是公有的，则在派生类中，基类的成员将遵循原有的访问说明符

2.如果继承是受保护的，则基类的所有公有成员在派生类中都是受保护的。

3.如果继承是私有的，则基类的所有公有和受保护成员在派生类中都是私有的。

派生类向基类转换的可访问性

1.只有当 D 公有地继承 B 时，用户代码才能使用派生类向基类的转换。

2.无论 D 以什么方式继承 B，D 的成员函数和友元都能使用派生类向基类的转换。

3.如果 D 以受保护的方式继承 B，则 D 的派生类的成员和友元也可以使用 D 向 B 的类型转换。

友元与继承

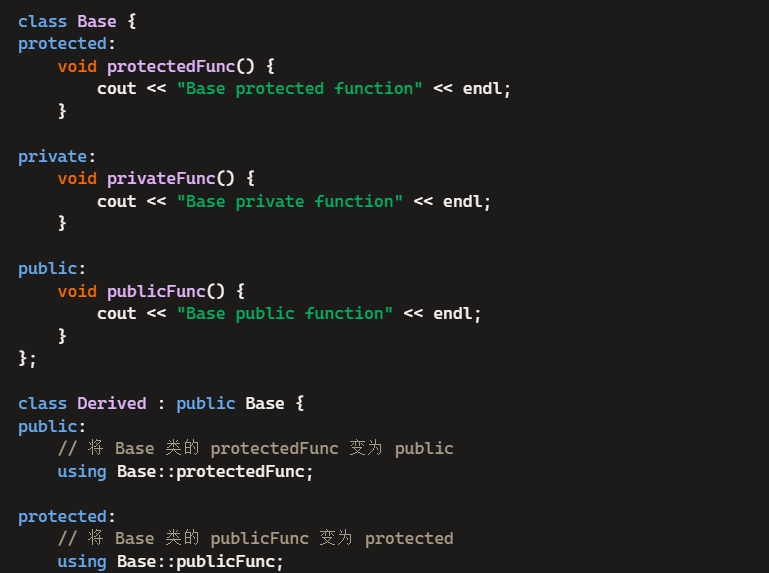
友元关系不能传递也不能继承。如果类 A 是基类 B 的友元，那么 A 可以访问 B 对象的成员和 B 的派生类对象中属于 B 部分的成员。

改变个别成员的可访问性

可以使用 using 声明改变派生类继承的某个名字的访问级别。

using 声明位于 public 部分就是公有成员，位于 private 部分就是私有成员，位于 protected 部分就是受保护成员。

下面例子中， protectedFunc使用using变为了public成员，publicFunc变成了protected成员。



继承的默认保护级别

和直接定义相似，当继承时不使用访问说明符，使用 class 关键字定义的派生类默认是私有继承的，使用 struct 关键字定义的派生类默认是公有继承的。

继承中的类作用域

每个类定义自己的作用域，在这个作用域内定义类的成员。当存在继承时，派生类的作用域嵌套在基类的作用域内。举个例子，如果使用派生类的实例查找一个成员变量，会先在派生类中查找，若派生类中没找到会在其基类中寻找。

在编译时进行名字查找

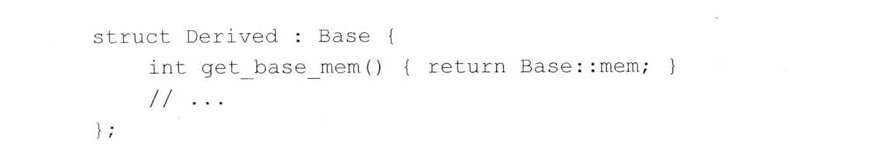
一个对象、引用或指针的静态类型决定了该对象的哪些成员是可见的，即使静态类型和动态类型可能不一致。

如果一个基类的成员在基类中是私有的，即使在派生类中将其访问级别改变为公有，仍然不能通过基类对象来访问该成员。这是因为基类对象的静态类型是基类，而在基类中，该成员是私有的。

名字冲突与继承

派生类能重用定义在其基类中的名字，定义在派生类的函数不会重载基类中的同名成员，重用后派生类的成员将隐藏同名的基类成员。因此除了覆盖虚函数外，派生类最好不要重用基类成员的名字。

通过作用域运算符来使用隐藏的基类成员。下面例子中，使用作用域运算符才可以调用基类的mem函数

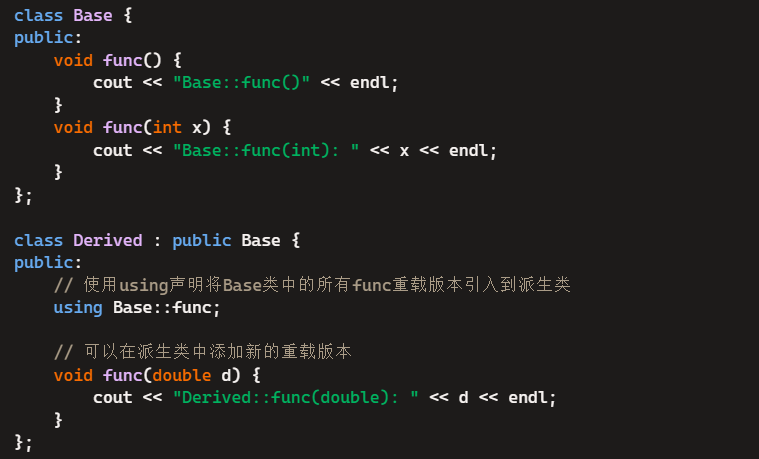


虚函数与作用域

派生类在重定义从基类继承的虚函数时，形参列表必须与基类的虚函数相同。如果不同，派生类定义的将是一个新函数，该新函数不是虚函数，并且会隐藏掉从基类继承的同名虚函数。

对于对于重载的虚函数，可以在派生类中覆盖所需的重载函数，但是必须一一覆盖，除非使用类内的using语句

使用using将重载函数添加到派生类



构造函数与拷贝控制

虚析构函数

因为继承关系的影响，基类通常应该定义一个虚析构函数。下面是原因：

因为经常通过动态分配来生成类的对象，使用完后要 delete 相应的类指针，这时需要调用类的析构函数来销毁对象。但是类指针的静态类型可能是基类的指针，却动态绑定到了一个继承类，这时需要保证 delete 执行的是继承类的析构函数。所以要将析构函数定义为虚析构函数，这样可以通过动态绑定执行正确的版本。此时只要基类的析构函数是虚函数，就可以保证 delete 基类指针时运行正确的析构函数版本。

ps：定义虚析构函数的额外影响——虚析构函数将会阻止编译器合成移动操作。

合成拷贝控制（编译器合成的拷贝控制成员，=default版本）与继承

类的对象分为基类部分和当前的子类部分：

基类部分：派生类的合成拷贝控制成员（构造函数、析构函数、赋值运算符等）在对其中的基类部分进行相关拷贝、销毁等操作时都是通过调用基类的对应成员完成的。

派生类部分：通过派生类定义的对应成员相关操作。

派生类中删除的拷贝控制与基类的关系：

1.如果基类的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是删除的，那么派生类中对应的成员也会是删除的（因为派生类中的拷贝控制成员需要调用基类的对应成员来完成操作）。

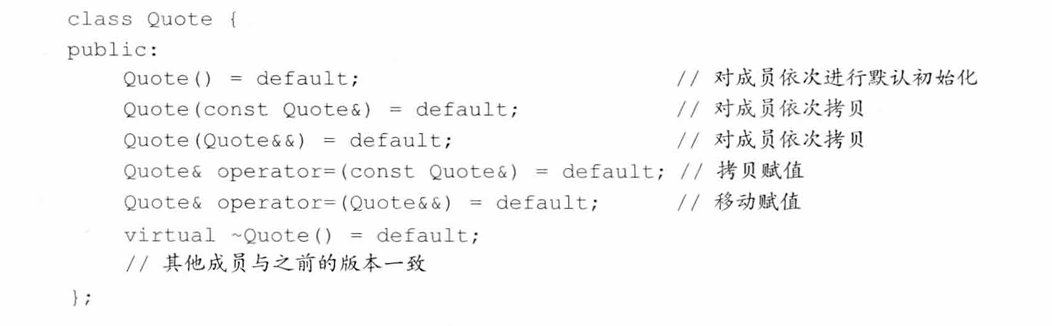
2.如果基类有一个不可访问或删除的析构函数，则派生类中合成的默认和拷贝构造函数也会是删除的，因为编译器无法销毁派生类对象的基类部分。（我不理解，这种情况应该无法出现才对）

3.如果基类的移动操作是删除的，那么派生类中的对应函数也是删除的，因为派生类的基类部分不可移动。

移动操作与继承

因为大多数基类都会定义一个虚析构函数，所以默认情况下基类通常没有合成的移动操作。如果需要执行移动操作，首先要在基类中定义。之后派生类会自动合成移动操作。

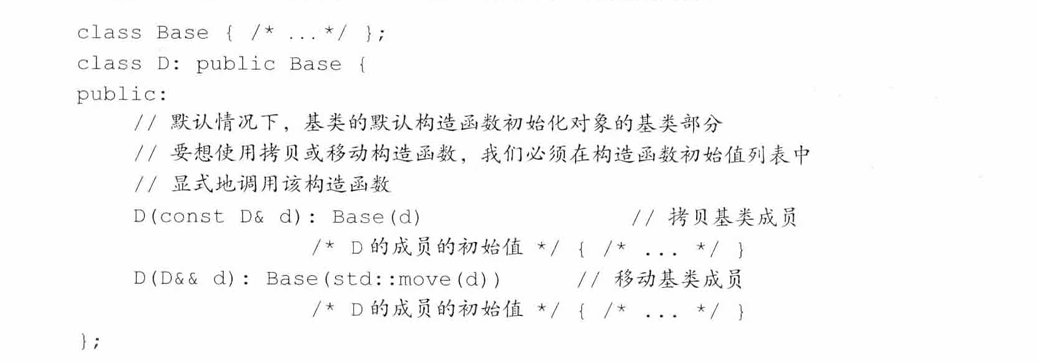
合成移动操作需要显式定义在基类中，就像书上的例子一样，此时在子类中就会自动获得移动合成操作。



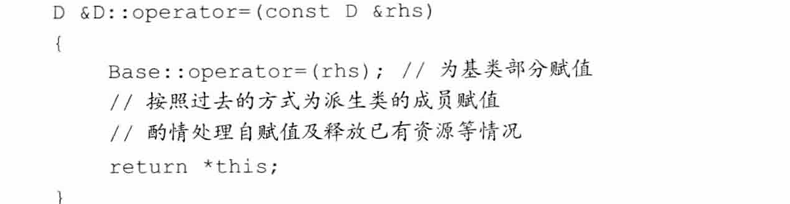
派生类的拷贝控制成员

当派生类定义了拷贝或移动操作（包括对应的构造函数和赋值运算符），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（通过调用基类的对应成员），无论基类的构造函数和赋值运算符是自定义的还是合成的，派生类的对应操作都可以使用它们。

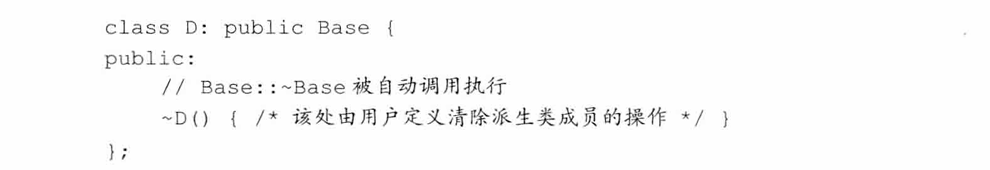
定义派生类的拷贝或移动构造函数，在拷贝或移动对象的自有部分前，需要先拷贝或移动基类部分：



派生类的赋值运算符重载，在赋值运算符重载函数中必须显式地为基类部分赋值：



析构函数不同，因为析构部分是隐式销毁的，基类部分也是自动销毁的，不需要派生类来负责：



在构造函数和析构函数中调用虚函数

在构造函数中调用虚函数时，虚函数的行为像非虚函数。这意味着它不会调用派生类的覆盖版本，而是调用基类版本。这是因为在构造函数期间，派生类的部分还没有完全构造，所以虚函数指向的是基类版本。在析构函数中调用虚函数也会类似地表现为调用基类版本。这是因为在析构过程中，派生类的部分已经被销毁，只剩下基类的部分。

继承的构造函数

C++11新增派生类能够重用其直接基类定义的构造函数。这些构造函数并非以常规的方式继承而来。这种非常规的“继承”指——一个类只初始化它的直接基类，一个类也只继承其直接基类的构造函数（类不能继承默认、拷贝、移动构造函数，这三种构造函数如果没有定义类会自己合成）。

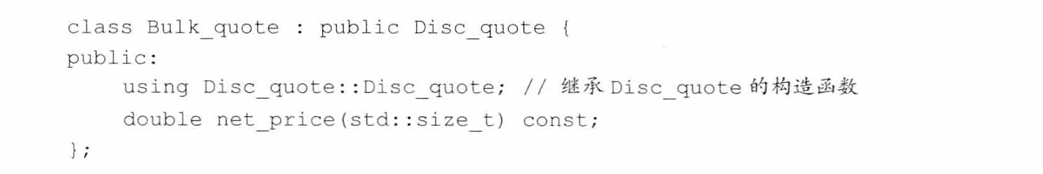
继承构造函数的作用：

1.减少重复代码

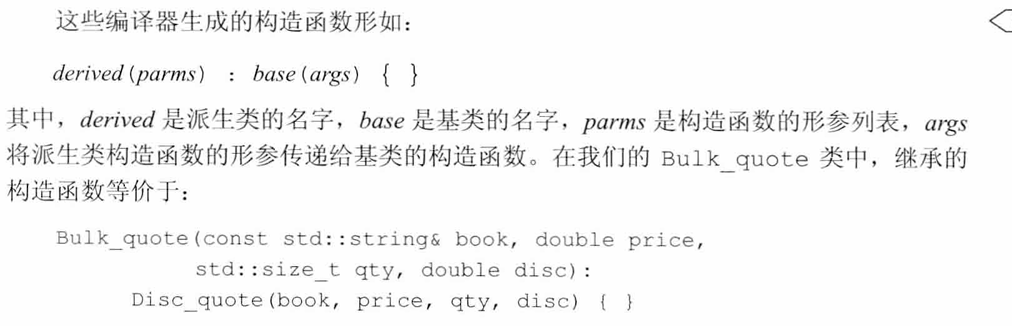
2.避免冗余初始化代码.在没有继承构造函数之前，派生类必须手动调用基类构造函数，重复初始化逻辑。通过继承构造函数，可以直接利用基类的初始化逻辑，避免冗余

3.当基类有多个构造函数时，继承构造函数可以使派生类更灵活地使用基类的不同构造方式

派生类通过一个 using 声明语句来继承基类的构造函数：



ps：通常 using 声明语句只是令某个名字在当前作用域内可见。但是当作用域构造函数时，using 声明语句将令编译器产生代码。



继承的构造函数的特点

和普通成员的 using 声明不一样，构造函数的 using 声明不会改变该构造函数的访问级别（即不管 using 声明在哪，派生类的构造函数的访问级别和基类中对应的构造函数的访问级别都是一样的）。

如果基类的构造函数是 explicit 或 constexpr 的，则继承的构造函数也有相同的属性。

当一个基类构造函数含有默认实参，这些实参并不会被直接继承，派生类会获得多个继承的构造函数，每个构造函数分别省略掉一个含有默认实参的形参。

如果基类含有几个构造函数，大多数时候派生类会继承所有这些构造函数。除了两个例外情况：

1.如果基类的某个构造函数和派生类自己定义的构造函数具有相同的参数列表，则该构造函数不会被继承。

2.默认、拷贝、移动构造函数不会被继承。

容器和继承

容器和继承可以结合使用来创建更加复杂和灵活的数据结构。例如：

1.你可以创建一个基类，定义一组共享的接口和行为，然后使用容器存储基类指针。

2.子类可以实现和扩展基类的功能，容器可以存储不同子类对象的指针，并以统一的方式进行管理。

当要在容器中存放继承体系中的对象时，使用基类指针作为容器的元素类型，最好放智能指针。