在 C++ 中，基类将类型相关的函数与它希望其派生类继承而无需更改的函数区分开来。 基类将其期望其派生类为自己定义的那些函数定义为虚拟函数。 利用这些知识，我们可以开始编写 Quote 类：

在 C++ 中，当通过基类的引用（或指针）调用虚拟函数时，就会发生动态绑定。

**动态绑定（Dynamic Binding）**，也称为**运行时绑定（Runtime Binding）或晚绑定（Late Binding）**，是面向对象编程中的一个重要概念。它指的是在程序运行时（而不是编译时）确定调用哪个函数或方法。动态绑定通常与\*\*多态性（Polymorphism）**和**虚函数（Virtual Functions）\*\*密切相关。

. 基类通常应该定义一个虚拟析构函数。 虚拟析构函数是必要的，即使它们不起作用。

当我们通过指针或引用调用虚函数时，调用将被动态绑定。根据引用或指针绑定到的对象的类型，将执行基类或其派生类之一中的版本。

 virtual 关键字仅出现在类内部的声明中，不能用于出现在类主体外部的函数定义。 在基类中声明为虚函数的函数在派生类中也隐式为虚函数

新标准让派生类明确指出它打算使用成员函数来覆盖它继承的虚拟函数。 它通过在参数列表之后指定 override 来实现此目的，或者如果成员是 const (§7.1.2，第 258 页) 或引用 (第 13.6.3，第 546 页) 函数，则在 const 或引用限定符之后指定 override 。

* 派生类对象包含基类和派生类的成员。
* 派生类对象的概念结构是逻辑上的，基类部分和派生类部分不一定在内存中连续存储。
* 派生类对象可以当作基类对象使用，基类指针或引用可以绑定到派生类对象。
* 这种行为是实现多态性的基础。

派生对象包含其基类的子对象这一事实是继承如何工作的关键。

每个类控制其成员的初始化方式。

**初始化过程详解：**

1. **基类部分初始化**：
   * Bulk\_quote的构造函数通过初始化列表调用Quote的构造函数，传递book和p参数。
   * Quote的构造函数初始化bookNo和price成员。
2. **派生类部分初始化**：
   * 基类部分初始化完成后，Bulk\_quote的成员min\_qty和discount被初始化。
3. **构造函数体执行**：
   * 最后，Bulk\_quote构造函数的函数体（这里是空的）被执行。

首先初始化基类，然后按照在类中声明的顺序初始化派生类的成员。

**1. 每个类定义自己的接口**

* 类的接口包括其**公有成员函数**和**受保护成员函数**，这些函数定义了如何与类的对象进行交互。
* 即使派生类继承了基类的成员，它也应该通过基类的接口来访问和操作基类部分，而不是直接操作基类的成员变量。

**2. 派生类不能直接初始化基类的成员**

* 派生类的构造函数**不能直接初始化基类的成员变量**，因为这些成员变量是基类的一部分，属于基类的实现细节。
* 派生类必须通过调用基类的构造函数来初始化基类部分。这是通过派生类构造函数的**初始化列表**完成的。

**. 静态成员的特性**

* 静态成员（static成员）属于类本身，而不是类的某个特定对象。
* 静态成员在程序的生命周期内只有**一个实例**，无论创建了多少个类的对象。

**2. 继承链中的成员继承**

* 每个派生类会继承其**直接基类**的所有成员（包括数据成员和成员函数）。
* 直接基类的成员包括它从自己的基类继承的成员，依此类推，直到继承链的顶端。
* 最终，**最派生类**（即继承链最底层的类）会继承整个继承链中所有基类的成员。
* 可以将基类指针或引用绑定到派生类对象，这是C++多态性的基础。
* 静态类型是变量声明时的类型，动态类型是变量实际指向或绑定的对象的类型。
* 通过virtual函数，可以实现运行时多态性。
* 智能指针也支持派生类到基类的转换，可以用于管理动态分配的对象。

理解基类的指针或引用的静态类型可能与其动态类型不同是至关重要的。

当我们从派生类型的对象初始化或分配基类型的对象时，仅复制、移动或分配派生对象的基类部分。 对象的派生部分将被忽略。

* 当基类的print()函数**不是虚函数**时，ptr->print()的调用是基于**静态类型**（即Base\*）决定的。
* 编译器在编译时就知道ptr的类型是Base\*，因此会直接调用Base类的print()函数，而不会考虑ptr实际指向的对象类型（即动态类型）。
* 这种行为称为**静态绑定**（或早期绑定）。
* 当基类的print()函数是**虚函数**时，ptr->print()的调用是基于**动态类型**（即ptr实际指向的对象类型）决定的。
* 编译器会在运行时查找ptr实际指向的对象的虚函数表（vtable），并调用正确的函数。
* 这种行为称为**动态绑定**（或晚期绑定）。

**1. 派生类到基类的转换**

* **仅适用于指针或引用类型**：
  + 可以将派生类对象的指针或引用转换为基类类型的指针或引用。
  + 这种转换是隐式的，编译器会自动完成。

**2. 基类到派生类的转换**

* **没有隐式转换**：
  + 不能将基类对象或基类指针/引用隐式转换为派生类类型。
  + 这种转换需要显式的类型转换（如static\_cast或dynamic\_cast），并且只有在基类指针或引用实际指向派生类对象时才是安全的。

**3. 访问控制的影响**

* 派生类到基类的转换可能因为**访问控制**而不可访问。
* 如果基类的继承方式是private或protected，则派生类到基类的转换在类外部是不可用的。

**4. 派生类对象赋值给基类对象**

* 虽然不能直接将派生类对象转换为基类对象，但可以通过**拷贝、移动或赋值**操作将派生类对象的基类部分复制到基类对象中。
* 这种操作只会复制基类部分的成员，派生类特有的成员会被忽略。

当我们通过基类的引用或指针调用基类中定义的函数时，我们不知道执行该成员的对象的类型。 该对象可以是基类对象或派生类的对象。 如果该函数是虚拟函数，则要延迟到运行时才决定运行哪个函数。 运行的虚拟函数的版本是由引用绑定到的或指针指向的对象的类型定义的版本。

另一方面，对非虚函数的调用是在编译时绑定的。 类似地，对对象上的任何函数（虚拟或非虚拟）的调用也在编译时绑定。 对象的类型是固定不变的——我们无法使对象的动态类型与其静态类型不同。 因此，对对象进行的调用在编译时绑定到对象类型定义的版本。

仅当通过引用或指针进行调用时，虚拟对象才会在运行时解析。 只有在这些情况下，对象的动态类型才可能与其静态类型不同。

在基类中为虚函数的函数在其派生类中也隐式为虚函数。 当派生类重写虚拟类时，基类和派生类中的参数必须完全匹配。

* 虚函数的默认参数由**静态类型**决定，而不是动态类型。
* 如果基类和派生类的虚函数定义了不同的默认参数，可能会导致程序行为异常。
* **最佳实践**是基类和派生类的虚函数使用相同的默认参数值，或者避免使用默认参数。

通常，只有成员函数（或友元）内部的代码才需要使用作用域运算符来规避虚拟机制。

如果打算调用其基类版本的派生虚函数省略了作用域运算符，则该调用将在运行时解析为对派生版本本身的调用，从而导致无限递归。

**2. 纯虚函数的作用**

* **定义接口**：
  + 纯虚函数用于定义基类的接口，要求派生类必须实现该函数。
* **创建抽象类**：
  + 包含纯虚函数的类称为**抽象类**，不能实例化对象。
  + 抽象类只能作为基类，用于派生其他类。

我们不能创建抽象基类类型的对象。

**. 重构的优点**

* **重构是通过调整类层次结构来改进代码设计的过程。**
* **在面向对象编程中，重构通常涉及将操作或数据从一个类移动到另一个类。**
* **重构可以提高代码的复用性、可读性和可维护性。**
* **重构后，使用类的代码通常不需要修改，但需要重新编译。**
* **代码复用**：
  + 将共同逻辑提取到 Disc\_quote 中，避免了代码重复。
* **更好的设计**：
  + 通过引入中间类，类层次结构更加清晰，职责更加明确。
* **扩展性**：
  + 未来如果需要添加新的折扣类（如 Limited\_quote），可以直接继承 Disc\_quote，而不需要重复实现折扣逻辑。

**1. protected 访问控制的作用**

* protected 是一种介于 private 和 public 之间的访问控制级别。
* **对类的外部用户**：
  + protected 成员是**不可访问**的，类似于 private 成员。
* **对派生类**：
  + protected 成员是**可访问**的，类似于 public 成员。

**2. protected 的特性**

* **派生类可以访问基类的 protected 成员**：
  + 派生类的成员函数和友元可以访问基类的 protected 成员。
* **访问限制**：
  + 派生类只能通过**派生类对象**访问基类的 protected 成员，而不能通过**基类对象**访问。
* 继承访问说明符（public、protected、private）控制派生类及其用户对基类成员的访问权限。
* 基类成员的访问权限由基类中的访问说明符决定。
* public 继承保留基类成员的访问权限。
* private 继承将基类的 public 和 protected 成员变为 private。
* protected 继承将基类的 public 和 protected 成员变为 protected

**派生类到基类的转换**

* 派生类到基类的转换是指将派生类对象（或指针/引用）隐式转换为基类对象（或指针/引用）。
* 这种转换的**可访问性**取决于：
  1. **继承访问说明符**（public、protected、private）。
  2. **代码的上下文**（用户代码、派生类的成员函数或友元、派生类的派生类等）。

**3. 总结**

* **用户代码**：
  + 只有在 public 继承时才能使用派生类到基类的转换。
* **派生类的成员函数和友元**：
  + 无论继承方式如何，都可以使用派生类到基类的转换。
* **派生类的派生类的成员函数和友元**：
  + 只有在 public 或 protected 继承时才能使用派生类到基类的转换。

对于代码中的任何给定点，如果基类的公共成员是可访问的，那么派生到基类的转换也是可访问的，否则就不行。

正如友谊不可传递（§7.3.4，第 279 页）一样，友谊也不可继承。 基类的友元对其派生类的成员没有特殊访问权限，而派生类的友元对其基类也没有特殊访问权限

友谊不是遗传的； 每个类控制对其成员的访问。

派生类只能为允许访问的名称提供 using 声明。

私有派生类应显式指定私有而不是依赖于默认值。 明确表明私有继承是有意为之，而不是疏忽。

. 除了重写继承的虚函数之外，派生类通常不应重用其基类中定义的名称。

* **基类指针只能访问基类的成员**：
  + 即使基类指针指向派生类的派生类对象，它也只能访问基类中定义的成员（包括虚函数）。
  + 如果需要访问派生类的特有成员，必须将指针**向下转换**为派生类类型（如使用 dynamic\_cast 或 static\_cast）。

C++中**名字查找**（Name Lookup）和**继承**（Inheritance）的关系，特别是在调用成员函数时如何确定调用哪个函数。

**2. 名字查找的四个步骤**

**（1）确定静态类型**

* 首先确定 p 或 obj 的**静态类型**（即变量声明时的类型）。
* 静态类型必须是一个类类型（因为只有类类型才能有成员函数）。

**（2）在静态类型及其基类中查找成员**

* 在静态类型对应的类中查找 mem。
* 如果找不到，则沿着继承链向上查找，直到找到 mem 或查找到继承链的顶端。
* 如果在整个继承链中都找不到 mem，则编译失败。

**（3）类型检查**

* 找到 mem 后，进行**类型检查**，确保调用的参数和返回值类型与定义匹配。
* 如果类型不匹配，则编译失败。

**（4）生成代码**

* 如果调用合法，编译器生成代码：
  + 如果 mem 是**虚函数**，并且通过指针或引用调用，则生成**动态绑定**代码，根据对象的**动态类型**决定调用哪个版本。
  + 如果 mem 是**非虚函数**，或者通过对象（而不是指针或引用）调用，则生成**静态绑定**代码，直接调用找到的函数。
* 如果一个类可能被继承，并且可能通过基类指针删除派生类对象，那么基类的析构函数应该声明为虚函数。
* 即使基类没有其他虚函数，也应该将析构函数声明为虚函数，以避免潜在的问题。
* 如果基类的拷贝控制成员（如构造函数、析构函数、拷贝构造函数、拷贝赋值运算符等）是删除的或不可访问的，则派生类的相应成员也会被定义为删除的。
* 这是因为派生类的拷贝控制成员需要调用基类的相应成员来操作基类部分的对象。
* 如果基类的析构函数是删除的或不可访问的，则派生类的默认构造函数和拷贝构造函数也会被定义为删除的。
* 如果基类的移动操作是删除的或不可访问的，则派生类的相应移动操作也会被定义为删除的。

默认情况下，基类默认构造函数初始化派生对象的基类部分。 如果我们想要复制（或移动）基类部分，则必须在派生的构造函数初始值设定项列表中显式使用基类的复制（或移动）构造函数。

如果构造函数或析构函数调用 virtual，则运行的版本是与构造函数或析构函数本身的类型相对应的版本。

由于派生对象在分配给基类型对象时会被“切片”，因此通过继承相关的容器和类型不能很好地混合。

**3. 设计原则**

* **“Is A”关系**：
  + 使用**公开继承**（public）来表示。
  + 派生类应该能够替代基类使用。
* **“Has A”关系**：
  + 使用**成员变量**来表示。
  + 一个类包含另一个类的对象作为其组成部分。

继承让我们可以编写与其基类共享行为的新类，但可以根据需要重写或添加该行为。 动态绑定让我们可以在运行时根据对象的动态类型选择要运行的函数版本，从而忽略类型差异。 继承和动态绑定的结合使我们能够编写具有类型特定行为的独立于类型的程序。

在 C++ 中，动态绑定仅适用于声明为虚拟并通过引用或指针调用的函数。

派生类对象包含与其每个基类相对应的子对象。 因为每个派生对象都包含基类部分，所以我们可以将派生类类型的引用或指针转换为可访问基类的引用或指针。

继承的对象是通过在处理派生部分之前构造、复制、移动和分配对象的基础部分来构造、复制、移动和分配的。 析构函数以相反的顺序执行； 首先销毁派生类型，然后是基类子对象的析构函数。 基类通常应该定义一个虚拟析构函数，即使该类不需要析构函数。 如果指向基类的指针在实际寻址派生类对象时被删除，则析构函数必须是虚拟的。

派生类为其每个基类指定一个保护级别。 公共基类的成员是派生类的接口的一部分； 私人基地的成员无法访问； 从派生类派生的类可以访问受保护基类的成员，但派生类的用户不能访问。

**1. 对象的构造、拷贝、移动和赋值**

* **构造**：
  + 派生类对象的构造顺序是：**先构造基类部分，再构造派生类部分**。
  + 基类部分的构造通过派生类的构造函数初始化列表调用基类的构造函数完成。
* **拷贝、移动和赋值**：
  + 派生类对象的拷贝、移动和赋值操作也是**先处理基类部分，再处理派生类部分**。
  + 派生类的拷贝构造函数、移动构造函数、拷贝赋值运算符和移动赋值运算符需要显式调用基类的相应操作。

**3. 虚析构函数的重要性**

* **基类通常应该定义虚析构函数**：
  + 即使基类本身不需要析构函数，也应该定义虚析构函数。
  + 这是为了确保当通过基类指针删除派生类对象时，能够正确调用派生类的析构函数。
* **如果基类没有虚析构函数**：
  + 当通过基类指针删除派生类对象时，只会调用基类的析构函数，而不会调用派生类的析构函数。
  + 这会导致派生类部分的资源无法正确释放，从而引发内存泄漏或其他未定义行为。