

# 第 15 章

# 面向对象程序设计

## 内容

---

15.1 OOP: 概述 .....	526
15.2 定义基类和派生类 .....	527
15.3 虚函数 .....	536
15.4 抽象基类 .....	540
15.5 访问控制与继承 .....	542
15.6 继承中的类作用域 .....	547
15.7 构造函数与拷贝控制 .....	551
15.8 容器与继承 .....	558
15.9 文本查询程序再探 .....	562
小结 .....	575
术语表 .....	575

面向对象程序设计基于三个基本概念：数据抽象、继承和动态绑定。第 7 章已经介绍了数据抽象的知识，本章将介绍继承和动态绑定。

继承和动态绑定对程序的编写有两方面的影响：一是我们可以更容易地定义与其他类相似但不完全相同的新类；二是在使用这些彼此相似的类编写程序时，我们可以在一定程度上忽略掉它们的区别。

592

在很多程序中都存在着一些相互关联但是有细微差别的概念。例如，书店中不同书籍的定价策略可能不同：有的书籍按原价销售，有的则打折销售。有时，我们给那些购买书籍超过一定数量的顾客打折；另一些时候，则只对前多少本销售的书籍打折，之后就调回原价，等等。面向对象的程序设计（OOP）适用于这类应用。



## 15.1 OOP：概述

面向对象程序设计（object-oriented programming）的核心思想是数据抽象、继承和动态绑定。通过使用数据抽象，我们可以将类的接口与实现分离（见第 7 章）；使用继承，可以定义相似的类型并对其相似关系建模；使用动态绑定，可以在一定程度上忽略相似类型的区别，而以统一的方式使用它们的对象。

### 继承

通过继承（inheritance）联系在一起的类构成一种层次关系。通常在层次关系的根部有一个基类（base class），其他类则直接或间接地从基类继承而来，这些继承得到的类称为派生类（derived class）。基类负责定义在层次关系中所有类共同拥有的成员，而每个派生类定义各自特有的成员。

为了对之前提到的不同定价策略建模，我们首先定义一个名为 `Quote` 的类，并将它作为层次关系中的基类。`Quote` 的对象表示按原价销售的书籍。`Quote` 派生出另一个名为 `Bulk_quote` 的类，它表示可以打折销售的书籍。

这些类将包含下面的两个成员函数：

- `isbn()`，返回书籍的 ISBN 编号。该操作不涉及派生类的特殊性，因此只定义在 `Quote` 类中。
- `net_price(size_t)`，返回书籍的实际销售价格，前提是用户购买该书的数量达到一定标准。这个操作显然是类型相关的，`Quote` 和 `Bulk_quote` 都应该包含该函数。

在 C++ 语言中，基类将类型相关的函数与派生类不做改变直接继承的函数区分对待。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成虚函数（virtual function）。因此，我们可以将 `Quote` 类编写成：

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

593

派生类必须通过使用类派生列表（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中每个基类前面可以有访问说明符：

```
class Bulk_quote : public Quote {           // Bulk_quote 继承了 Quote
public:
    double net_price(std::size_t) const override;
};
```

因为 `Bulk_quote` 在它的派生列表中使用了 `public` 关键字，因此我们完全可以把

`Bulk_quote` 的对象当成 `Quote` 的对象来使用。

派生类必须在其内部对所有重新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。出于 15.3 节（第 538 页）将要解释的原因，C++11 新标准允许派生类显式地注明它将使用哪个成员函数改写基类的虚函数，具体措施是在该函数的形参列表之后增加一个 `override` 关键字。

## 动态绑定

通过使用 **动态绑定**（dynamic binding），我们能用同一段代码分别处理 `Quote` 和 `Bulk_quote` 的对象。例如，当要购买的书籍和购买的数量都已知时，下面的函数负责打印总的费用：

```
// 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;
}
```

该函数非常简单：它返回调用 `net_price()` 的结果，并将该结果连同调用 `isbn()` 的结果一起打印出来。

关于上面的函数有两个有意思的结论：因为函数 `print_total` 的 `item` 形参是基类 `Quote` 的一个引用，所以出于 15.2.3 节（第 534 页）将要解释的原因，我们既能使用基类 `Quote` 的对象调用该函数，也能使用派生类 `Bulk_quote` 的对象调用它；又因为 `print_total` 是使用引用类型调用 `net_price` 函数的，所以出于 15.2.1 节（第 528 页）将要解释的原因，实际传入 `print_total` 的对象类型将决定到底执行 `net_price` 的哪个版本：

```
// basic 的类型是 Quote; bulk 的类型是 Bulk_quote
print_total(cout, basic, 20);           // 调用 Quote 的 net_price
print_total(cout, bulk, 20);            // 调用 Bulk_quote 的 net_price
```

第一条调用句将 `Quote` 对象传入 `print_total`，因此当 `print_total` 调用 `net_price` 时，执行的是 `Quote` 的版本；在第二条调用语句中，实参的类型是 `Bulk_quote`，因此执行的是 `Bulk_quote` 的版本（计算打折信息）。因为在上述过程中函数的运行版本由实参决定，即在运行时选择函数的版本，所以动态绑定有时又被称为运行时绑定（run-time binding）。

&lt; 594



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。

## 15.2 定义基类和派生类

定义基类和派生类的方式在很多方面都与我们已知的定义其他类的方式类似，但是也有一些不同之处。本节将介绍在定义有继承关系的类时可能用到的基本特性。



## 15.2.1 定义基类

我们首先完成 `Quote` 类的定义：

```
class Quote {
public:
    Quote() = default;           // 关于=default 请参见 7.1.4 节（第 237 页）
    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折扣计算算法
    virtual double net_price(std::size_t n) const
    { return n * price; }
    virtual ~Quote() = default;   // 对析构函数进行动态绑定
private:
    std::string bookNo;          // 书籍的 ISBN 编号
protected:
    double price = 0.0;          // 代表普通状态下不打折的价格
};
```

对于上面这个类来说，新增的部分是在 `net_price` 函数和析构函数之前增加的 `virtual` 关键字以及最后的 `protected` 访问说明符。我们将在 15.7.1 节（第 552 页）详细介绍虚析构函数的知识，现在只需记住作为继承关系中根节点的类通常都会定义一个虚析构函数。



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。

## 成员函数与继承

派生类可以继承其基类的成员，然而当遇到如 `net_price` 这样与类型相关的操作时，

**595** 派生类必须对其重新定义。换句话说，派生类需要对这些操作提供自己的新定义以覆盖（`override`）从基类继承而来的旧定义。

在 C++ 语言中，基类必须将它的两种成员函数区分开来：一种是基类希望其派生类进行覆盖的函数；另一种是基类希望派生类直接继承而不要改变的函数。对于前者，基类通常将其定义为虚函数（`virtual`）。当我们使用指针或引用调用虚函数时，该调用将被动态绑定。根据引用或指针所绑定的对象类型不同，该调用可能执行基类的版本，也可能执行某个派生类的版本。

基类通过在其成员函数的声明语句之前加上关键字 `virtual` 使得该函数执行动态绑定。任何构造函数之外的非静态函数（参见 7.6 节，第 268 页）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。我们将在 15.3 节（第 536 页）介绍更多关于虚函数的知识。

成员函数如果没被声明为虚函数，则其解析过程发生在编译时而非运行时。对于 `isbn` 成员来说这正是我们希望看到的结果。`isbn` 函数的执行与派生类的细节无关，不管作用于 `Quote` 对象还是 `Bulk_quote` 对象，`isbn` 函数的行为都一样。在我们的继承层次关系中只有一个 `isbn` 函数，因此也就不存在调用 `isbn()` 时到底执行哪个版本的疑问。

## 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。和其他使用基类的代码一样，派生类能访问公有成员，而不能访问私有成员。不过在某些时候基类中还有这样一种成员，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。我们用受保护的（protected）访问运算符说明这样的成员。

我们的 Quote 类希望它的派生类定义各自的 net\_price 函数，因此派生类需要访问 Quote 的 price 成员。此时我们将 price 定义成受保护的。与之相反，派生类访问 bookNo 成员的方式与其他用户是一样的，都是通过调用 isbn 函数，因此 bookNo 被定义成私有的，即使是 Quote 派生出来的类也不能直接访问它。我们将在 15.5 节（第 542 页）介绍更多关于受保护成员的知识。

### 15.2.1 节练习

练习 15.1：什么是虚成员？

练习 15.2：protected 访问说明符与 private 有何区别？

练习 15.3：定义你自己的 Quote 类和 print\_total 函数。

## 15.2.2 定义派生类

596

派生类必须通过使用类派生列表（class derivation list）明确指出它是从哪个（哪些）基类继承而来的。类派生列表的形式是：首先是一个冒号，后面紧跟以逗号分隔的基类列表，其中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，因此，我们的 Bulk\_quote 类必须包含一个 net\_price 成员：

```
class Bulk_quote : public Quote {           // Bulk_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
};
```

我们的 Bulk\_quote 类从它的基类 Quote 那里继承了 isbn 函数和 bookNo、price 等数据成员。此外，它还定义了 net\_price 的新版本，同时拥有两个新增加的数据成员 min\_qty 和 discount。这两个成员分别用于说明享受折扣所需购买的最低数量以及一旦该数量达到之后具体的折扣信息。

我们将在 15.5 节（第 543 页）详细介绍派生列表中用到的访问说明符。现在，我们只需知道访问说明符的作用是控制派生类从基类继承而来的成员是否对派生类的用户可见。

如果一个派生是公有的，则基类的公有成员也是派生类接口的组成部分。此外，我们能将公有派生类型的对象绑定到基类的引用或指针上。因为我们在派生列表中使用了

`public`, 所以 `Bulk_quote` 的接口隐式地包含 `isbn` 函数, 同时在任何需要 `Quote` 的引用或指针的地方我们都能使用 `Bulk_quote` 的对象。

大多数类都只继承自一个类, 这种形式的继承被称作“单继承”, 它构成了本章的主题。关于派生列表中含有多个基类的情况将在 18.3 节(第 710 页)中介绍。

### 派生类中的虚函数

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, 则该虚函数的行为类似于其他的普通成员, 派生类会直接继承其在基类中的版本。

**C++ 11** 派生类可以在它覆盖的函数前使用 `virtual` 关键字, 但不是非得这么做。我们将在 15.3 节(第 538 页)介绍其原因, C++11 新标准允许派生类显式地注明它使用某个成员函数覆盖了它继承的虚函数。具体做法是在形参列表后面、或者在 `const` 成员函数(参见 7.1.2 节, 第 231 页)的 `const` 关键字后面、或者在引用成员函数(参见 13.6.3 节, 第 483 页)的引用限定符后面添加一个关键字 `override`。

### 597 派生类对象及派生类向基类的类型转换

一个派生类对象包含多个组成部分: 一个含有派生类自己定义的(非静态)成员的子对象, 以及一个与该派生类继承的基类对应的子对象, 如果有多个基类, 那么这样的子对象也有多个。因此, 一个 `Bulk_quote` 对象将包含四个数据元素: 它从 `Quote` 继承而来的 `bookNo` 和 `price` 数据成员, 以及 `Bulk_quote` 自己定义的 `min_qty` 和 `discount` 成员。

C++ 标准并没有明确规定派生类的对象在内存中如何分布, 但是我们可以认为 `Bulk_quote` 的对象包含如图 15.1 所示的两部分。



在一个对象中, 继承自基类的部分和派生类自定义的部分不一定是连续存储的。图 15.1 只是表示类工作机理的概念模型, 而非物理模型。

图 15.1: Bulk\_quote 对象的概念结构

因为在派生类对象中含有与其基类对应的组成部分, 所以我们能把派生类的对象当成基类对象来使用, 而且我们也能将基类的指针或引用绑定到派生类对象中的基类部分上。

```

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

```

这种转换通常称为派生类到基类的(derived-to-base)类型转换。和其他类型转换一样, 编译器会隐式地执行派生类到基类的转换(参见 4.11 节, 第 141 页)。

这种隐式特性意味着我们可以把派生类对象或者派生类对象的引用用在需要基类引

用的地方；同样的，我们也可以把派生类对象的指针用在需要基类指针的地方。



在派生类对象中含有与其基类对应的组成部分，这一事实是继承的关键所在。

## 派生类构造函数

&lt;598

尽管在派生类对象中含有从基类继承而来的成员，但是派生类并不能直接初始化这些成员。和其他创建了基类对象的代码一样，派生类也必须使用基类的构造函数来初始化它的基类部分。



每个类控制它自己的成员初始化过程。

派生类对象的基类部分与派生类对象自己的数据成员都是在构造函数的初始化阶段（参见 7.5.1 节，第 258 页）执行初始化操作的。类似于我们初始化成员的过程，派生类构造函数同样是通过构造函数初始化列表来将实参传递给基类构造函数的。例如，接受四个参数的 Bulk\_quote 构造函数如下所示：

```
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
    Quote(book, p), min_qty(qty), discount(disc) {}
// 与之前一致
};
```

该函数将它的前两个参数（分别表示 ISBN 和价格）传递给 Quote 的构造函数，由 Quote 的构造函数负责初始化 Bulk\_quote 的基类部分（即 bookNo 成员和 price 成员）。当（空的）Quote 构造函数体结束后，我们构建的对象的基类部分也就完成初始化了。接下来初始化由派生类直接定义的 min\_qty 成员和 discount 成员。最后运行 Bulk\_quote 构造函数的（空的）函数体。

除非我们特别指出，否则派生类对象的基类部分会像数据成员一样执行默认初始化。如果想使用其他的基类构造函数，我们需要以类名加圆括号内的实参列表的形式为构造函数提供初始值。这些实参将帮助编译器决定到底应该选用哪个构造函数来初始化派生类对象的基类部分。



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，就可以享受折扣价格了
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

该函数产生一个打折后的价格：如果给定的数量超过了 min\_qty，则将 discount （一 <599

个小于 1 大于 0 的数) 作用于 price。

我们将在 15.6 节 (第 547 页) 进一步讨论作用域, 目前只需要了解派生类的作用域嵌套在基类的作用域之内。因此, 对于派生类的一个成员来说, 它使用派生类成员 (例如 min\_qty 和 discount) 的方式与使用基类成员 (例如 price) 的方式没什么不同。

### 关键概念: 遵循基类的接口

必须明确一点: 每个类负责定义各自的接口。要想与类的对象交互必须使用该类的接口, 即使这个对象是派生类的基类部分也是如此。

因此, 派生类对象不能直接初始化基类的成员。尽管从语法上来说我们可以在派生类构造函数体内给它的公有或受保护的基类成员赋值, 但是最好不要这么做。和使用基类的其他场合一样, 派生类应该遵循基类的接口, 并且通过调用基类的构造函数来初始化那些从基类中继承而来的成员。

## 继承与静态成员

如果基类定义了一个静态成员 (参见 7.6 节, 第 268 页), 则在整个继承体系中只存在该成员的唯一定义。不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唯一的实例。

```
class Base {
public:
    static void statmem();
};

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

静态成员遵循通用的访问控制规则, 如果基类中的成员是 private 的, 则派生类无权访问它。假设某静态成员是可访问的, 则我们既能通过基类使用它也能通过派生类使用它:

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();           // 正确: Base 定义了 statmem
    Derived::statmem();        // 正确: Derived 继承了 statmem
    // 正确: 派生类的对象能访问基类的静态成员
    derived_obj.statmem();    // 通过 Derived 对象访问
    statmem();                // 通过 this 对象访问
}
```

## 600> 派生类的声明

派生类的声明与其他类差别不大 (参见 7.3.3 节, 第 250 页), 声明中包含类名但是不包含它的派生列表:

```
class Bulk_quote : public Quote; // 错误: 派生列表不能出现在这里
class Bulk_quote;             // 正确: 声明派生类的正确方式
```

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。派生列表以及与定义有关的其他细节必须与类的主体一起出现。

## 被用作基类的类

如果我们想将某个类用作基类，则该类必须已经定义而非仅仅声明：

```
class Quote; // 声明但未定义
// 错误: Quote 必须被定义
class Bulk_quote : public Quote { ... };
```

这一规定的原因显而易见：派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类当然要知道它们是什么。因此该规定还有一层隐含的意思，即一个类不能派生它本身。

一个类是基类，同时它也可以是一个派生类：

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

在这个继承关系中，Base 是 D1 的直接基类 (direct base)，同时是 D2 的间接基类 (indirect base)。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。

每个类都会继承直接基类的所有成员。对于一个最终的派生类来说，它会继承其直接基类的成员；该直接基类的成员又含有其基类的成员；依此类推直至继承链的顶端。因此，最终的派生类将包含它的直接基类的子对象以及每个间接基类的子对象。

## 防止继承的发生

有时我们会定义这样一种类，我们不希望其他类继承它，或者不想考虑它是否适合作为一个基类。为了实现这一目的，C++11 新标准提供了一种防止继承发生的方法，即在类名后跟一个关键字 final：

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

### 15.2.2 节练习

C++  
11

601

**练习 15.4：**下面哪条声明语句是不正确的？请解释原因。

- class Base { ... };
- (a) class Derived : public Derived { ... };
- (b) class Derived : private Base { ... };
- (c) class Derived : public Base;

**练习 15.5：**定义你自己的 Bulk\_quote 类。

**练习 15.6：**将 Quote 和 Bulk\_quote 的对象传给 15.2.1 节（第 529 页）练习中的 print\_total 函数，检查该函数是否正确。

**练习 15.7：**定义一个类使其实现一种数量受限的折扣策略，具体策略是：当购买书籍的数量不超过一个给定的限量时享受折扣，如果购买量一旦超过了限量，则超出的部分将以原价销售。



### 15.2.3 类型转换与继承



理解基类和派生类之间的类型转换是理解 C++ 语言面向对象编程的关键所在。

通常情况下，如果我们想把引用或指针绑定到一个对象上，则引用或指针的类型应与对象的类型一致（参见 2.3.1 节，第 46 页和 2.3.2 节，第 47 页），或者对象的类型含有一个可接受的 `const` 类型转换规则（参见 4.11.2 节，第 144 页）。存在继承关系的类是一个重要的例外：我们可以将基类的指针或引用绑定到派生类对象上。例如，我们可以用 `Quote&` 指向一个 `Bulk_quote` 对象，也可以把一个 `Bulk_quote` 对象的地址赋给一个 `Quote*`。

可以将基类的指针或引用绑定到派生类对象上有一层极为重要的含义：当使用基类的引用（或指针）时，实际上我们并不清楚该引用（或指针）所绑定对象的真实类型。该对象可能是基类的对象，也可能是派生类的对象。



和内置指针一样，智能指针类（参见 12.1 节，第 400 页）也支持派生类向基类的类型转换，这意味着我们可以将一个派生类对象的指针存储在一个基类的智能指针内。



### 静态类型与动态类型

当我们使用存在继承关系的类型时，必须将一个变量或其他表达式的 **静态类型**（static type）与该表达式表示对象的 **动态类型**（dynamic type）区分开来。表达式的静态类型在编译时总是已知的，它是变量声明时的类型或表达式生成的类型；动态类型则是变量或表达式表示的内存中的对象的类型。动态类型直到运行时才可知。

602 &gt;

例如，当 `print_total` 调用 `net_price` 时（参见 15.1 节，第 527 页）：

```
double ret = item.net_price(n);
```

我们知道 `item` 的静态类型是 `Quote&`，它的动态类型则依赖于 `item` 绑定的实参，动态类型直到在运行时调用该函数时才会知道。如果我们传递一个 `Bulk_quote` 对象给 `print_total`，则 `item` 的静态类型将与它的动态类型不一致。如前所述，`item` 的静态类型是 `Quote&`，而在此例中它的动态类型则是 `Bulk_quote`。

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。例如，`Quote` 类型的变量永远是一个 `Quote` 对象，我们无论如何都不能改变该变量对应的对象的类型。



基类的指针或引用的静态类型可能与其动态类型不一致，读者一定要理解其中的原因。

### 不存在从基类向派生类的隐式类型转换……

之所以存在派生类向基类的类型转换是因为每个派生类对象都包含一个基类部分，而基类的引用或指针可以绑定到该基类部分上。一个基类的对象既可以以独立的形式存在，也可以作为派生类对象的一部分存在。如果基类对象不是派生类对象的一部分，则它只含有基类定义的成员，而不含有派生类定义的成员。

因为一个基类的对象可能是派生类对象的一部分，也可能不是，所以不存在从基类向派生类的自动类型转换：

```
Quote base;
Bulk_quote* bulkP = &base;           // 错误：不能将基类转换成派生类
Bulk_quote& bulkRef = base;         // 错误：不能将基类转换成派生类
```

如果上述赋值是合法的，则我们有可能会使用 bulkP 或 bulkRef 访问 base 中本不存在的成员。

除此之外还有一种情况显得有点特别，即使一个基类指针或引用绑定在一个派生类对象上，我们也不能执行从基类向派生类的转换：

```
Bulk_quote bulk;
Quote *itemP = &bulk;                // 正确：动态类型是 Bulk_quote
Bulk_quote *bulkP = itemP;           // 错误：不能将基类转换成派生类
```

编译器在编译时无法确定某个特定的转换在运行时是否安全，这是因为编译器只能通过检查指针或引用的静态类型来推断该转换是否合法。如果在基类中含有一个或多个虚函数，我们可以使用 `dynamic_cast`（参见 19.2.1 节，第 730 页）请求一个类型转换，该转换的安全检查将在运行时执行。同样，如果我们已知某个基类向派生类的转换是安全的，则我们可以使用 `static_cast`（参见 4.11.3 节，第 144 页）来强制覆盖掉编译器的检查工作。

### ……在对象之间不存在类型转换



603 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。很多时候，我们确实希望将派生类对象转换成它的基类类型，但是这种转换的实际发生过程往往与我们期望的有所差别。

请注意，当我们初始化或赋值一个类类型的对象时，实际上是在调用某个函数。当执行初始化时，我们调用构造函数（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页）；而当执行赋值操作时，我们调用赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）。这些成员通常都包含一个参数，该参数的类型是类类型的 `const` 版本的引用。

因为这些成员接受引用作为参数，所以派生类向基类的转换允许我们给基类的拷贝/移动操作传递一个派生类的对象。这些操作不是虚函数。当我们给基类的构造函数传递一个派生类对象时，实际运行的构造函数是基类中定义的那个，显然该构造函数只能处理基类自己的成员。类似的，如果我们将一个派生类对象赋值给一个基类对象，则实际运行的赋值运算符也是基类中定义的那个，该运算符同样只能处理基类自己的成员。

例如，我们的书店类使用了合成版本的拷贝和赋值操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。关于拷贝控制与继承的知识将在 15.7.2 节（第 552 页）做更详细的介绍，现在我们只需要知道合成版本会像其他类一样逐成员地执行拷贝或赋值操作：

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

当构造 `item` 时，运行 `Quote` 的拷贝构造函数。该函数只能处理 `bookNo` 和 `price` 两个成员，它负责拷贝 `bulk` 中 `Quote` 部分的成员，同时忽略掉 `bulk` 中 `Bulk_quote` 部分的成员。类似的，对于将 `bulk` 赋值给 `item` 的操作来说，只有 `bulk` 中 `Quote` 部分的成员被赋值给 `item`。

因为在上述过程中会忽略 `Bulk_quote` 部分，所以我们可以说 `bulk` 的 `Bulk_quote` 部分被切掉（sliced down）了。



当我们用一个派生类对象为一个基类对象初始化或赋值时，只有该派生类对象中的基类部分会被拷贝、移动或赋值，它的派生类部分将被忽略掉。

### 15.2.3 节练习

**练习 15.8：**给出静态类型和动态类型的定义。

**练习 15.9：**在什么情况下表达式的静态类型可能与动态类型不同？请给出三个静态类型与动态类型不同的例子。

**练习 15.10：**回忆我们在 8.1 节（第 279 页）进行的讨论，解释第 284 页中将 ifstream 传递给 Sales\_data 的 read 函数的程序是如何工作的。

#### 关键概念：存在继承关系的类型之间的转换规则

要想理解在具有继承关系的类之间发生的类型转换，有三点非常重要：

- 从派生类向基类的类型转换只对指针或引用类型有效。
- 基类向派生类不存在隐式类型转换。
- 和任何其他成员一样，派生类向基类的类型转换也可能会由于访问受限而变得不可行。我们将在 15.5 节（第 544 页）详细介绍可访问性的问题。

尽管自动类型转换只对指针或引用类型有效，但是继承体系中的大多数类仍然（显式或隐式地）定义了拷贝控制成员（参见第 13 章）。因此，我们通常能够将一个派生类对象拷贝、移动或赋值给一个基类对象。不过需要注意的是，这种操作只处理派生类对象的基类部分。



## 15.3 虚函数

如前所述，在 C++ 语言中，当我们使用基类的引用或指针调用一个虚成员函数时会执行动态绑定（参见 15.1 节，第 527 页）。因为我们直到运行时才能知道到底调用了哪个版本的虚函数，所以所有虚函数都必须有定义。通常情况下，如果我们不使用某个函数，则无须为该函数提供定义（参见 6.1.2 节，第 186 页）。但是我们必须为每一个虚函数都提供定义，而不管它是否被用到了，这是因为连编译器也无法确定到底会使用哪个虚函数。

#### 对虚函数的调用可能在运行时才被解析

当某个虚函数通过指针或引用调用时，编译器产生的代码直到运行时才能确定应该调用哪个版本的函数。被调用的函数是与绑定到指针或引用上的对象的动态类型相匹配的那个。

举个例子，考虑 15.1 节（第 527 页）的 print\_total 函数，该函数通过其名为 item 的参数来进一步调用 net\_price，其中 item 的类型是 Quote&。因为 item 是引用而且 net\_price 是虚函数，所以我们到底调用 net\_price 的哪个版本完全依赖于运行时绑定到 item 的实参的实际（动态）类型：

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);           // 调用 Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
```

```
print_total(cout, derived, 10);           // 调用 Bulk_quote::net_price
```

在第一条调用语句中，item 绑定到 Quote 类型的对象上，因此当 print\_total 调用 net\_price 时，运行在 Quote 中定义的版本。在第二条调用语句中，item 绑定到 Bulk\_quote 类型的对象上，因此 print\_total 调用 Bulk\_quote 定义的 net\_price。605

必须要搞清楚的一点是，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。

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

当我们通过一个具有普通类型（非引用非指针）的表达式调用虚函数时，在编译时就会将调用的版本确定下来。例如，如果我们使用 base 调用 net\_price，则应该运行 net\_price 的哪个版本是显而易见的。我们可以改变 base 表示的对象的值（即内容），但是不会改变该对象的类型。因此，在编译时该调用就会被解析成 Quote 的 net\_price。

### 关键概念：C++的多态性

OOP 的核心思想是多态性（polymorphism）。多态性这个词源自希腊语，其含义是“多种形式”。我们把具有继承关系的多个类型称为多态类型，因为我们能使用这些类型的“多种形式”而无须在意它们的差异。引用或指针的静态类型与动态类型不同这一事实正是 C++ 语言支持多态性的根本所在。

当我们使用基类的引用或指针调用基类中定义的一个函数时，我们并不知道该函数真正作用的对象是什么类型，因为它可能是一个基类的对象也可能是一个派生类的对象。如果该函数是虚函数，则直到运行时才会决定到底执行哪个版本，判断的依据是引用或指针所绑定的对象的真实类型。

另一方面，对非虚函数的调用在编译时进行绑定。类似的，通过对象进行的函数（虚函数或非虚函数）调用也在编译时绑定。对象的类型是确定不变的，我们无论如何都不可能令对象的动态类型与静态类型不一致。因此，通过对象进行的函数调用将在编译时绑定到该对象所属类中的函数版本上。



当且仅当对通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型才有可能与静态类型不同。

### 派生类中的虚函数

当我们在派生类中覆盖了某个虚函数时，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，因为一旦某个函数被声明成虚函数，则在所有派生类中它都是虚函数。

一个派生类的函数如果覆盖了某个继承而来的虚函数，则它的形参类型必须与被它覆盖的基类函数完全一致。

同样，派生类中虚函数的返回类型也必须与基类函数匹配。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。也就是说，如果 D 由 B 派生得到，则基类的虚函数可以返回 B\* 而派生类的对应函数可以返回 D\*，只不过这样的返回类型要求从 D 到 B 的类型转换是可访问的。15.5 节（第 544 页）将介绍如何确定一个基类的可访问性，在 15.8.1 节（第 561 页）中我们将看到这种虚函数的一个实际例子。606



基类中的虚函数在派生类中隐含地也是一个虚函数。当派生类覆盖了某个虚函数时，该函数在基类中的形参必须与派生类中的形参严格匹配。

### final 和 override 说明符

如我们将要在 15.6 节（第 550 页）介绍的，派生类如果定义了一个函数与基类中虚函数的名字相同但是形参列表不同，这仍然是合法的行为。编译器将认为新定义的这个函数与基类中原有的函数是相互独立的。这时，派生类的函数并没有覆盖掉基类中的版本。就实际的编程习惯而言，这种声明往往意味着发生了错误，因为我们可能原本希望派生类能覆盖掉基类中的虚函数，但是一不小心把形参列表弄错了。

C++  
11

要想调试并发现这样的错误显然非常困难。在 C++11 新标准中我们可以使用 `override` 关键字来说明派生类中的虚函数。这么做的好处是在使得程序员的意图更加清晰的同时让编译器可以为我们发现一些错误，后者在编程实践中显得更加重要。如果我们使用 `override` 标记了某个函数，但该函数并没有覆盖已存在的虚函数，此时编译器将报错：

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};

struct D1 : B {
    void f1(int) const override;           // 正确: f1 与基类中的 f1 匹配
    void f2(int) override;                 // 错误: B 没有形如 f2(int) 的函数
    void f3() override;                  // 错误: f3 不是虚函数
    void f4() override;                  // 错误: B 没有名为 f4 的函数
};
```

在 D1 中，`f1` 的 `override` 说明符是正确的，因为基类和派生类中的 `f1` 都是 `const` 成员，并且它们都接受一个 `int` 返回 `void`，所以 D1 中的 `f1` 正确地覆盖了它从 B 中继承而来的虚函数。

D1 中 `f2` 的声明与 B 中 `f2` 的声明不匹配，显然 B 中定义的 `f2` 不接受任何参数而 D1 的 `f2` 接受一个 `int`。因为这两个声明不匹配，所以 D1 的 `f2` 不能覆盖 B 的 `f2`，它是一个新函数，仅仅是名字恰好与原来的函数一样而已。因为我们使用 `override` 所表达的意思是我们希望能覆盖基类中的虚函数而实际上并未做到，所以编译器会报错。

607

因为只有虚函数才能被覆盖，所以编译器会拒绝 D1 的 `f3`。该函数不是 B 中的虚函数，因此它不能被覆盖。类似的，`f4` 的声明也会发生错误，因为 B 中根本就没有名为 `f4` 的函数。

我们还能把某个函数指定为 `final`，如果我们已经把函数定义成 `final` 了，则之后任何尝试覆盖该函数的操作都将引发错误：

```
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：覆盖从间接基类 B 继承而来的 f2
    void f1(int) const;            // 错误：D2 已经将 f2 声明成 final
};
```

`final` 和 `override` 说明符出现在形参列表（包括任何 `const` 或引用修饰符）以及尾置返回类型（参见 6.3.3 节，第 206 页）之后。

### 虚函数与默认实参

和其他函数一样，虚函数也可以拥有默认实参（参见 6.5.1 节，第 211 页）。如果某次函数调用使用了默认实参，则该实参值由本次调用的静态类型决定。

换句话说，如果我们通过基类的引用或指针调用函数，则使用基类中定义的默认实参，即使实际运行的是派生类中的函数版本也是如此。此时，传入派生类函数的将是基类函数定义的默认实参。如果派生类函数依赖不同的实参，则程序结果将与我们的预期不符。



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。

### 回避虚函数的机制

在某些情况下，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特定版本。使用作用域运算符可以实现这一目的，例如下面的代码：

```
// 强行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);
```

该代码强行调用 `Quote` 的 `net_price` 函数，而不管 `baseP` 实际指向的对象类型到底是什么。该调用将在编译时完成解析。



通常情况下，只有成员函数（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。

什么时候我们需要回避虚函数的默认机制呢？通常是当一个派生类的虚函数调用它覆盖的基类的虚函数版本时。在此情况下，基类的版本通常完成继承层次中所有类型都要做的共同任务，而派生类中定义的版本需要执行一些与派生类本身密切相关的操作。



如果一个派生类虚函数需要调用它的基类版本，但是没有使用作用域运算符，则在运行时该调用将被解析为对派生类版本自身的调用，从而导致无限递归。

608

## 15.3 节练习

**练习 15.11：**为你的 `Quote` 类体系添加一个名为 `debug` 的虚函数，令其分别显示每个类的数据成员。

**练习 15.12：**有必要将一个成员函数同时声明成 `override` 和 `final` 吗？为什么？

**练习 15.13：**给定下面的类，解释每个 `print` 函数的机理：

```
class base {  
public:  
    string name() { return basename; }  
    virtual void print(ostream &os) { os << basename; }  
private:
```

```

        string basename;
    };
    class derived : public base {
public:
    void print(ostream &os) { print(os); os << " " << i; }
private:
    int i;
};

```

在上述代码中存在问题吗？如果有，你该如何修改它？

**练习 15.14：**给定上一题中的类以及下面这些对象，说明在运行时调用哪个函数：

base bobj;	base *bp1 = &bobj;	base &br1 = bobj;
derived dobj;	base *bp2 = &dobj;	base &br2 = dobj;
(a) bobj.print();	(b) dobj.print();	(c) bp1->name();
(d) bp2->name();	(e) br1.print();	(f) br2.print();

## 15.4 抽象基类

假设我们希望扩展书店程序并令其支持几种不同的折扣策略。除了购买量超过一定数量享受折扣外，我们也可能提供另外一种策略，即购买量不超过某个限额时可以享受折扣，但是一旦超过限额就要按原价支付。或者折扣策略还可能是购买量超过一定数量后购买的全部书籍都享受折扣，否则全都不打折。

上面的每个策略都要求一个购买量的值和一个折扣值。我们可以定义一个新的名为 Disc\_quote 的类来支持不同的折扣策略，其中 Disc\_quote 负责保存购买量的值和折扣值。其他的表示某种特定策略的类（如 Bulk\_quote）将分别继承自 Disc\_quote，每个派生类通过定义自己的 net\_price 函数来实现各自的折扣策略。

在定义 Disc\_quote 类之前，首先要确定它的 net\_price 函数完成什么工作。显然我们的 Disc\_quote 类与任何特定的折扣策略都无关，因此 Disc\_quote 类中的 net\_price 函数是没有实际含义的。

我们可以在 Disc\_quote 类中不定义新的 net\_price，此时，Disc\_quote 将继承 Quote 中的 net\_price 函数。

然而，这样的设计可能导致用户编写出一些无意义的代码。用户可能会创建一个 Disc\_quote 对象并为其提供购买量和折扣值，如果将该对象传给一个像 print\_total 这样的函数，则程序将调用 Quote 版本的 net\_price。显然，最终计算出的销售价格并没有考虑我们在创建对象时提供的折扣值，因此上述操作毫无意义。

### 纯虚函数

认真思考上面描述的情形我们可以发现，关键问题不仅仅是不知道应该如何定义 net\_price，而是我们根本就不希望用户创建一个 Disc\_quote 对象。Disc\_quote 类表示的是一本打折书籍的通用概念，而非某种具体的折扣策略。

我们可以将 net\_price 定义成纯虚（pure virtual）函数从而令程序实现我们的设计意图，这样做可以清晰明了地告诉用户当前这个 net\_price 函数是没有实际意义的。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（即在声明语句

的分号之前) 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声明语句处:

```
// 用于保存折扣值和购买量的类, 派生类使用这些数据可以实现不同的价格策略
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;            // 表示折扣的小数值
};
```

和我们之前定义的 Bulk\_quote 类一样, Disc\_quote 也分别定义了一个默认构造函数和一个接受四个参数的构造函数。尽管我们不能直接定义这个类的对象, 但是 Disc\_quote 的派生类构造函数将会使用 Disc\_quote 的构造函数来构建各个派生类对象的 Disc\_quote 部分。其中, 接受四个参数的构造函数将前两个参数传递给 Quote 的构造函数, 然后直接初始化自己的成员 discount 和 quantity。默认构造函数则对这些成员进行默认初始化。

值得注意的是, 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。◀ 610也就是说, 我们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

含有(或者未经覆盖直接继承)纯虚函数的类是**抽象基类**(abstract base class)。抽象基类负责定义接口, 而后续的其他类可以覆盖该接口。我们不能(直接)创建一个抽象基类的对象。因为 Disc\_quote 将 net\_price 定义成了纯虚函数, 所以我们不能定义 Disc\_quote 的对象。我们可以定义 Disc\_quote 的派生类的对象, 前提是这些类覆盖了 net\_price 函数:

```
// Disc_quote 声明了纯虚函数, 而 Bulk_quote 将覆盖该函数
Disc_quote discounted;           // 错误: 不能定义 Disc_quote 的对象
Bulk_quote bulk;                 // 正确: Bulk_quote 中没有纯虚函数
```

Disc\_quote 的派生类必须给出自己的 net\_price 定义, 否则它们仍将是抽象基类。



我们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

接下来可以重新实现 Bulk\_quote 了, 这一次我们让它继承 Disc\_quote 而非直接继承 Quote:

```
// 当同一书籍的销售量超过某个值时启用折扣
// 折扣的值是一个小于 1 的正的小数值, 以此来降低正常销售价格
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实现一种新的折扣策略
double net_price(std::size_t) const override;
};

}

```

这个版本的 `Bulk_quote` 的直接基类是 `Disc_quote`, 间接基类是 `Quote`。每个 `Bulk_quote` 对象包含三个子对象: 一个(空的)`Bulk_quote`部分、一个`Disc_quote`子对象和一个`Quote`子对象。

如前所述, 每个类各自控制其对象的初始化过程。因此, 即使 `Bulk_quote` 没有自己的数据成员, 它也仍然需要像原来一样提供一个接受四个参数的构造函数。该构造函数将它的实参传递给 `Disc_quote` 的构造函数, 随后 `Disc_quote` 的构造函数继续调用 `Quote` 的构造函数。`Quote` 的构造函数首先初始化 `bulk` 的 `bookNo` 和 `price` 成员, 当 `Quote` 的构造函数结束后, 开始运行 `Disc_quote` 的构造函数并初始化 `quantity` 和 `discount` 成员, 最后运行 `Bulk_quote` 的构造函数, 该函数无须执行实际的初始化或其他工作。

611 &gt;

### 关键概念: 重构

在 `Quote` 的继承体系中增加 `Disc_quote` 类是重构 (refactoring) 的一个典型示例。重构负责重新设计类的体系以便将操作和/或数据从一个类移动到另一个类中。对于面向对象的应用程序来说, 重构是一种很普遍的现象。

值得注意的是, 即使我们改变了整个继承体系, 那些使用了 `Bulk_quote` 或 `Quote` 的代码也无须进行任何改动。不过一旦类被重构 (或以其他方式被改变), 就意味着我们必须重新编译含有这些类的代码了。

## 15.4 节练习

**练习 15.15:** 定义你自己的 `Disc_quote` 和 `Bulk_quote`。

**练习 15.16:** 改写你在 15.2.2 节 (第 533 页) 练习中编写的数量受限的折扣策略, 令其继承 `Disc_quote`。

**练习 15.17:** 尝试定义一个 `Disc_quote` 的对象, 看看编译器给出的错误信息是什么?



## 15.5 访问控制与继承

每个类分别控制自己的成员初始化过程 (参见 15.2.2 节, 第 531 页), 与之类似, 每个类还分别控制着其成员对于派生类来说是否可访问 (accessible)。

### 受保护的成员

如前所述, 一个类使用 `protected` 关键字来声明那些它希望与派生类分享但是不想被其他公共访问使用的成员。`protected` 说明符可以看做是 `public` 和 `private` 中和后的产物:

- 和私有成员类似, 受保护的成员对于类的用户来说是不可访问的。

- 和公有成员类似，受保护的成员对于派生类的成员和友元来说是可访问的。

此外，`protected` 还有另外一条重要的性质。

- 派生类的成员或友元只能通过派生类对象来访问基类的受保护成员。派生类对于一个基类对象中的受保护成员没有任何访问特权。

为了理解最后一条规则，请考虑如下的例子：

612

```
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; // j 默认是 private
};

// 正确：clobber 能访问 Sneaky 对象的 private 和 protected 成员
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }
// 错误：clobber 不能访问 Base 的 protected 成员
void clobber(Base &b) { b.prot_mem = 0; }
```

如果派生类（及其友元）能访问基类对象的受保护成员，则上面的第二个 `clobber`（接受一个 `Base&`）将是合法的。该函数不是 `Base` 的友元，但是它仍然能够改变一个 `Base` 对象的内容。如果按照这样的思路，则我们只要定义一个形如 `Sneaky` 的新类就能非常简单地规避掉 `protected` 提供的访问保护了。

要想阻止以上的用法，我们就要做出如下规定，即派生类的成员和友元只能访问派生类对象中的基类部分的受保护成员；对于普通的基类对象中的成员不具有特殊的访问权限。

## 公有、私有和受保护继承

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。举个例子，考虑如下的继承关系：

```
class Base {
public:
    void pub_mem(); // public 成员
protected:
    int prot_mem; // protected 成员
private:
    char priv_mem; // private 成员
};

struct Pub_Derv : public Base {
    // 正确：派生类能访问 protected 成员
    int f() { return prot_mem; }
    // 错误：private 成员对于派生类来说是不可访问的
    char g() { return priv_mem; }
};

struct Priv_Derv : private Base {
    // private 不影响派生类的访问权限
    int f1() const { return prot_mem; }
};
```

派生访问说明符对于派生类的成员（及友元）能否访问其直接基类的成员没什么影响。对 613 基类成员的访问权限只与基类中的访问说明符有关。Pub\_Derv 和 Priv\_Derv 都能访问受保护的成员 prot\_mem，同时它们都不能访问私有成员 priv\_mem。

派生访问说明符的目的是控制派生类用户（包括派生类的派生类在内）对于基类成员的访问权限：

```
Pub_Derv d1;           // 继承自 Base 的成员是 public 的
Priv_Derv d2;          // 继承自 Base 的成员是 private 的
d1.pub_mem();          // 正确：pub_mem 在派生类中是 public 的
d2.pub_mem();          // 错误：pub_mem 在派生类中是 private 的
```

Pub\_Derv 和 Priv\_Derv 都继承了 pub\_mem 函数。如果继承是公有的，则成员将遵循其原有的访问说明符，此时 d1 可以调用 pub\_mem。在 Priv\_Derv 中，Base 的成员是私有的，因此类的用户不能调用 pub\_mem。

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 {
    // 正确：Base::prot_mem 在 Pub_Derv 中仍然是 protected 的
    int use_base() { return prot_mem; }
};

struct Derived_from_Private : public Priv_Derv {
    // 错误：Base::prot_mem 在 Priv_Derv 中是 private 的
    int use_base() { return prot_mem; }
};
```

Pub\_Derv 的派生类之所以能访问 Base 的 prot\_mem 成员是因为该成员在 Pub\_Derv 中仍然是受保护的。相反，Priv\_Derv 的派生类无法执行类的访问，对于它们来说，Priv\_Derv 继承自 Base 的所有成员都是私有的。

假设我们之前还定义了一个名为 Prot\_Derv 的类，它采用受保护继承，则 Base 的所有公有成员在新定义的类中都是受保护的。Prot\_Derv 的用户不能访问 pub\_mem，但是 Prot\_Derv 的成员和友元可以访问那些继承而来的成员。



## 派生类向基类转换的可访问性

派生类向基类的转换（参见 15.2.2 节，第 530 页）是否可访问由使用该转换的代码决定，同时派生类的派生访问说明符也会有影响。假定 D 继承自 B：

- 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；如果 D 继承 B 的方式是受保护的或者私有的，则用户代码不能使用该转换。
- 不论 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；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的。
- 如果 D 继承 B 的方式是公有的或者受保护的，则 D 的派生类的成员和友元可以使用 D 向 B 的类型转换；反之，如果 D 继承 B 的方式是私有的，则不能使用。



对于代码中的某个给定节点来说，如果基类的公有成员是可访问的，则派生类向基类的类型转换也是可访问的；反之则不行。

## 关键概念：类的设计与受保护的成员

不考虑继承的话，我们可以认为一个类有两种不同的用户：普通用户和类的实现者。

其中，普通用户编写的代码使用类的对象，这部分代码只能访问类的公有（接口）成员；实现者则负责编写类的成员和友元的代码，成员和友元既能访问类的公有部分，也能访问类的私有（实现）部分。

如果进一步考虑继承的话就会出现第三种用户，即派生类。基类把它希望派生类能够使用的部分声明成受保护的。普通用户不能访问受保护的成员，而派生类及其友元仍旧不能访问私有成员。

和其他类一样，基类应该将其接口成员声明为公有的；同时将属于其实现的部分分成两组：一组可供派生类访问，另一组只能由基类及基类的友元访问。对于前者应该声明为受保护的，这样派生类就能在实现自己的功能时使用基类的这些操作和数据；对于后者应该声明为私有的。

## 友元与继承

就像友元关系不能传递一样（参见 7.3.4 节，第 250 页），友元关系同样也不能继承。基类的友元在访问派生类成员时不具有特殊性，类似的，派生类的友元也不能随意访问基类的成员：

```
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：Pal 是 Base 的友元
    int f2(Sneaky s) { return s.j; }     // 错误：Pal 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：Pal 是 Base 的友元
};
```

如前所述，每个类负责控制自己的成员的访问权限，因此尽管看起来有点儿奇怪，但 f3 确实是正确的。Pal 是 Base 的友元，所以 Pal 能够访问 Base 对象的成员，这种可访问性包括了 Base 对象内嵌在其派生类对象中的情况。◀615

当一个类将另一个类声明为友元时，这种友元关系只对做出声明的类有效。对于原来那个类来说，其友元的基类或者派生类不具有特殊的访问能力：

```
// 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
};
```



不能继承友元关系；每个类负责控制各自成员的访问权限。

## 改变个别成员的可访问性

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（参见 3.1 节，第 74 页）可以达到这一目的：

```
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;
};

```

因为 `Derived` 使用了私有继承，所以继承而来的成员 `size` 和 `n`（在默认情况下）是 `Derived` 的私有成员。然而，我们使用 `using` 声明语句改变了这些成员的可访问性。改变之后，`Derived` 的用户将可以使用 `size` 成员，而 `Derived` 的派生类将能使用 `n`。

通过在类的内部使用 `using` 声明语句，我们可以将该类的直接或间接基类中的任何可访问成员（例如，非私有成员）标记出来。`using` 声明语句中名字的访问权限由该 `using` 声明语句之前的访问说明符来决定。也就是说，如果一条 `using` 声明语句出现在类的 `private` 部分，则该名字只能被类的成员和友元访问；如果 `using` 声明语句位于 `public` 部分，则类的所有用户都能访问它；如果 `using` 声明语句位于 `protected` 部分，则该名字对于成员、友元和派生类是可访问的。



派生类只能为那些它可以访问的名字提供 `using` 声明。

616 >

### 默认的继承保护级别

在 7.2 节（第 240 页）中我们曾经介绍过使用 `struct` 和 `class` 关键字定义的类具有不同的默认访问说明符。类似的，默认派生运算符也由定义派生类所用的关键字来决定。默认情况下，使用 `class` 关键字定义的派生类是私有继承的；而使用 `struct` 关键字定义的派生类是公有继承的：

```

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

```

人们常常有一种错觉，认为在使用 `struct` 关键字和 `class` 关键字定义的类之间还有更深层次的差别。事实上，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；除此之外，再无其他不同之处。



一个私有派生的类最好显式地将 `private` 声明出来，而不要仅仅依赖于默认的设置。显式声明的好处是可以令私有继承关系清晰明了，不至于产生误会。

## 15.5 节练习

**练习 15.18：**假设给定了第 543 页和第 544 页的类，同时已知每个对象的类型如注释所示，判断下面的哪些赋值语句是合法的。解释那些不合法的语句为什么不允许：

<code>Base *p = &amp;d1;</code>	<code>// d1 的类型是 Pub_Derv</code>
<code>p = &amp;d2;</code>	<code>// d2 的类型是 Priv_Derv</code>

```

p = &d3;           // d3 的类型是 Prot_Derv
p = &dd1;          // dd1 的类型是 Derived_from_Public
p = &dd2;          // dd2 的类型是 Derived_from_Private
p = &dd3;          // dd3 的类型是 Derived_from_Protected

```

**练习 15.19:** 假设 543 页和 544 页的每个类都有如下形式的成员函数：

```
void memfcn(Base &b) { b = *this; }
```

对于每个类，分别判断上面的函数是否合法。

**练习 15.20:** 编写代码检验你对前面两题的回答是否正确。

**练习 15.21:** 从下面这些一般性抽象概念中任选一个（或者选一个你自己的），将其对应的一组类型组织成一个继承体系：

- (a) 图形文件格式（如 gif、tiff、jpeg、bmp）
- (b) 图形基元（如方格、圆、球、圆锥）
- (c) C++语言中的类型（如类、函数、成员函数）

**练习 15.22:** 对于你在上一题中选择的类，为其添加合适的虚函数及公有成员和受保护的成员。

## 15.6 继承中的类作用域



每个类定义自己的作用域（参见 7.4 节，第 253 页），在这个作用域内我们定义类的成员。当存在继承关系时，派生类的作用域嵌套（参见 2.2.4 节，第 43 页）在其基类的作用域之内。如果一个名字在派生类的作用域内无法正确解析，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。

&lt; 617

派生类的作用域位于基类作用域之内这一事实可能有点儿出人意料，毕竟在我们的程序文本中派生类和基类的定义是相互分离开来的。不过也恰恰因为类作用域有这种继承嵌套的关系，所以派生类才能像使用自己的成员一样使用基类的成员。例如，当我们编写下面的代码时：

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

名字 isbn 的解析将按照下述过程所示：

- 因为我们是通过 Bulk\_quote 的对象调用 isbn 的，所以首先在 Bulk\_quote 中查找，这一步没有找到名字 isbn。
- 因为 Bulk\_quote 是 Disc\_quote 的派生类，所以接下来在 Disc\_quote 中查找，仍然找不到。
- 因为 Disc\_quote 是 Quote 的派生类，所以接着查找 Quote；此时找到了名字 isbn，所以我们使用的 isbn 最终被解析为 Quote 中的 isbn。

### 在编译时进行名字查找

一个对象、引用或指针的静态类型（参见 15.2.3 节，第 532 页）决定了该对象的哪些成员是可见的。即使静态类型与动态类型可能不一致（当使用基类的引用或指针时会发生

这种情况), 但是我们能使用哪些成员仍然是由静态类型决定的。举个例子, 我们可以给 Disc\_quote 添加一个新成员, 该成员返回一个存有最小(或最大)数量及折扣价格的 pair (参见 11.2.3 节, 第 379 页):

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

我们只能通过 Disc\_quote 及其派生类的对象、引用或指针使用 discount\_policy:

```
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: bulkP 的类型是 Bulk_quote*
itemP->discount_policy();         // 错误: itemP 的类型是 Quote*
```

618 尽管在 bulk 中确实含有一个名为 discount\_policy 的成员, 但是该成员对于 itemP 却是不可见的。itemP 的类型是 Quote 的指针, 意味着对 discount\_policy 的搜索将从 Quote 开始。显然 Quote 不包含名为 discount\_policy 的成员, 所以我们无法通过 Quote 的对象、引用或指针调用 discount\_policy。

### 名字冲突与继承

和其他作用域一样, 派生类也能重用定义在其直接基类或间接基类中的名字, 此时定义在内层作用域(即派生类)的名字将隐藏定义在外层作用域(即基类)的名字(参见 2.2.4 节, 第 43 页):

```
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
};
```

get\_mem 中 mem 引用的解析结果是定义在 Derived 中的名字, 下面的代码

```
Derived d(42);
cout << d.get_mem() << endl;          // 打印 42
```

的输出结果将是 42。



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

我们可以通过作用域运算符来使用一个被隐藏的基类成员:

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
    // ...
};
```

作用域运算符将覆盖掉原有的查找规则，并指示编译器从 `Base` 类的作用域开始查找 `mem`。如果使用最新的 `Derived` 版本运行上面的代码，则 `d.get_mem()` 的输出结果将是 0。

**Best Practices**

除了覆盖继承而来的虚函数之外，派生类最好不要重用其他定义在基类中的名字。

### 关键概念：名字查找与继承

619

理解函数调用的解析过程对于理解 C++ 的继承至关重要，假定我们调用 `p->mem()`（或者 `obj.mem()`），则依次执行以下 4 个步骤：

- 首先确定 `p`（或 `obj`）的静态类型。因为我们调用的是一个成员，所以该类型必然是类类型。
- 在 `p`（或 `obj`）的静态类型对应的类中查找 `mem`。如果找不到，则依次在直接基类中不断查找直至到达继承链的顶端。如果找遍了该类及其基类仍然找不到，则编译器将报错。
- 一旦找到了 `mem`，就进行常规的类型检查（参见 6.1 节，第 183 页）以确认对于当前找到的 `mem`，本次调用是否合法。
- 假设调用合法，则编译器将根据调用的是否是虚函数而产生不同的代码：
  - 如果 `mem` 是虚函数且我们是通过引用或指针进行的调用，则编译器产生的代码将在运行时确定到底运行该虚函数的哪个版本，依据是对象的动态类型。
  - 反之，如果 `mem` 不是虚函数或者我们是通过对象（而非引用或指针）进行的调用，则编译器将产生一个常规函数调用。

### 一如既往，名字查找先于类型检查

如前所述，声明在内层作用域的函数并不会重载声明在外层作用域的函数（参见 6.4.1 节，第 210 页）。因此，定义派生类中的函数也不会重载其基类中的成员。和其他作用域一样，如果派生类（即内层作用域）的成员与基类（即外层作用域）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，基类成员也仍然会被隐藏掉：

```
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：参数列表为空的 memfcn 被隐藏了
d.Base::memfcn();            // 正确：调用 Base::memfcn
```

Derived 中的 memfcn 声明隐藏了 Base 中的 memfcn 声明。在上面的代码中前两条调用语句容易理解，第一个通过 Base 对象 b 进行的调用执行基类的版本；类似的，第二个通过 d 进行的调用执行 Derived 的版本；第三条调用语句有点特殊，d.memfcn() 是非法的。

为了解析这条调用语句，编译器首先在 Derived 中查找名字 memfcn；因为 Derived 确实定义了一个名为 memfcn 的成员，所以查找过程终止。一旦名字找到，编译器就不再继续查找了。Derived 中的 memfcn 版本需要一个 int 实参，而当前的调用语句无法提供任何实参，所以该调用语句是错误的。

## 虚函数与作用域

我们现在可以理解为什么基类与派生类中的虚函数必须有相同的形参列表了（参见 15.3 节，第 537 页）。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。例如：

```
class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // 隐藏基类的 fcn，这个 fcn 不是虚函数
    // D1 继承了 Base::fcn() 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 {
public:
    int fcn(int);           // 是一个非虚函数，隐藏了 D1::fcn(int)
    int fcn();               // 覆盖了 Base 的虚函数 fcn
    void f2();               // 覆盖了 D1 的虚函数 f2
};
```

D1 的 fcn 函数并没有覆盖 Base 的虚函数 fcn，原因是它们的形参列表不同。实际上，D1 的 fcn 将隐藏 Base 的 fcn。此时拥有了两个名为 fcn 的函数：一个是 D1 从 Base 继承而来的虚函数 fcn；另一个是 D1 自己定义的接受一个 int 参数的非虚函数 fcn。

## 通过基类调用隐藏的虚函数

给定上面定义的这些类后，我们来看几种使用其函数的方法：

```
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()
```

前三条调用语句是通过基类的指针进行的，因为 `fcn` 是虚函数，所以编译器产生的代码将在运行时确定使用虚函数的那个版本。判断的依据是该指针所绑定对象的真实类型。在 `bp2` 的例子中，实际绑定的对象是 `D1` 类型，而 `D1` 并没有覆盖那个不接受实参的 `fcn`，所以通过 `bp2` 进行的调用将在运行时解析为 `Base` 定义的版本。

接下来的三条调用语句是通过不同类型的指针进行的，每个指针分别指向继承体系中的一个类型。因为 `Base` 类中没有 `fcn()`，所以第一条语句是非法的，即使当前的指针碰巧指向了一个派生类对象也无济于事。

为了完整地阐明上述问题，我们不妨再观察一些对于非虚函数 `fcn(int)` 的调用语句：

```
Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 = &d2obj;
p1->fcn(42);           // 错误: Base 中没有接受一个 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)
```

在上面的每条调用语句中，指针都指向了 `D2` 类型的对象，但是由于我们调用的是非虚函数，所以不会发生动态绑定。实际调用的函数版本由指针的静态类型决定。

## 覆盖重载的函数

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，或者一个也不覆盖。

有时一个类仅需覆盖重载集合中的一些而非全部函数，此时，如果我们不得不覆盖基类中的每一个版本的话，显然操作将极其烦琐。

一种好的解决方案是为重载的成员提供一条 `using` 声明语句（参见 15.5 节，第 546 页），这样我们就无须覆盖基类中的每一个重载版本了。`using` 声明语句指定一个名字而不指定形参列表，所以一条基类成员函数的 `using` 声明语句就可以把该函数的所有重载实例添加到派生类作用域中。此时，派生类只需要定义其特有的函数就可以了，而无须为继承而来的其他函数重新定义。

类内 `using` 声明的一般规则同样适用于重载函数的名字（参见 15.5 节，第 546 页）；基类函数的每个实例在派生类中都必须是可访问的。对派生类没有重新定义的重载版本的访问实际上是对 `using` 声明点的访问。

## 15.6 节练习

**练习 15.23:** 假设第 550 页的 `D1` 类需要覆盖它继承而来的 `fcn` 函数，你应该如何对其进行修改？如果你修改之后 `fcn` 匹配了 `Base` 中的定义，则该节的那些调用语句将如何解析？

## 15.7 构造函数与拷贝控制

和其他类一样，位于继承体系中的类也需要控制当其对象执行一系列操作时发生什么样的行为，这些操作包括创建、拷贝、移动、赋值和销毁。如果一个类（基类或派生类）没有定义拷贝控制操作，则编译器将为它合成一个版本。当然，这个合成的版本也可以定义成被删除的函数。



## 15.7.1 虚析构函数

继承关系对基类拷贝控制最直接的影响是基类通常应该定义一个虚析构函数（参见 15.2.1 节，第 528 页），这样我们就能动态分配继承体系中的对象了。

如前所述，当我们 `delete` 一个动态分配的对象的指针时将执行析构函数（参见 13.1.3 节，第 445 页）。如果该指针指向继承体系中的某个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符的情况（参见 15.2.2 节，第 530 页）。例如，如果我们 `delete` 一个 `Quote*` 类型的指针，则该指针有可能实际指向了一个 `Bulk_quote` 类型的对象。如果这样的话，编译器就必须清楚它应该执行的是 `Bulk_quote` 的析构函数。和其他函数一样，我们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:
    // 如果我们删除的是一个指向派生类对象的基类指针，则需要虚析构函数
    virtual ~Quote() = default;           // 动态绑定析构函数
};
```

和其他虚函数一样，析构函数的虚属性也会被继承。因此，无论 `Quote` 的派生类使用合成的析构函数还是定义自己的析构函数，都将是虚析构函数。只要基类的析构函数是虚函数，就能确保当我们 `delete` 基类指针时将运行正确的析构函数版本：

```
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
```



**WARNING** 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的行为。

之前我们曾介绍过一条经验准则，即如果一个类需要析构函数，那么它也同样需要拷贝和赋值操作（参见 13.1.4 节，第 447 页）。基类的析构函数并不遵循上述准则，它是一个重要的例外。一个基类总是需要析构函数，而且它能将析构函数设定为虚函数。此时，该析构函数为了成为虚函数而令内容为空，我们显然无法由此推断该基类还需要赋值运算符或拷贝构造函数。

### 623 虚析构函数将阻止合成移动操作

基类需要一个虚析构函数这一事实还会对基类和派生类的定义产生另外一个间接的影响：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（参见 13.6.2 节，第 475 页）。

## 15.7.1 节练习

练习 15.24：哪种类需要虚析构函数？虚析构函数必须执行什么样的操作？



## 15.7.2 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依次进行初始化、赋值或销毁的操作。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

毁的操作。例如，

- 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 的默认构造函数将 bookNo 成员默认初始化为空字符串，同时使用类内初始值将 price 初始化为 0。
- Quote 的构造函数完成后，继续执行 Disc\_quote 的构造函数，它使用类内初始值初始化 qty 和 discount。
- Disc\_quote 的构造函数完成后，继续执行 Bulk\_quote 的构造函数，但是它什么具体工作也不做。

类似的，合成的 Bulk\_quote 拷贝构造函数使用（合成的） Disc\_quote 拷贝构造函数，后者又使用（合成的） Quote 拷贝构造函数。其中， Quote 拷贝构造函数拷贝 bookNo 和 price 成员； Disc\_quote 拷贝构造函数拷贝 qty 和 discount 成员。

值得注意的是，无论基类成员是合成的版本（如 Quote 继承体系的例子）还是自定义的版本都没有太大影响。唯一的要求是相应的成员应该可访问（参见 15.5 节，第 542 页）并且不是一个被删除的函数。

在我们的 Quote 继承体系中，所有类都使用合成的析构函数。其中，派生类隐式地使用而基类通过将其虚析构函数定义成`=default`而显式地使用。一如既往，合成的析构函数体是空的，其隐式的析构部分负责销毁类的成员（参见 13.1.3 节，第 444 页）。对于派生类的析构函数来说，它除了销毁派生类自己的成员外，还负责销毁派生类的直接基类；该直接基类又销毁它自己的直接基类，以此类推直至继承链的顶端。

如前所述，Quote 因为定义了析构函数而不能拥有合成的移动操作，因此当我们移动 Quote 对象时实际使用的是合成的拷贝操作（参见 13.6.2 节，第 477 页）。如我们即将看到的那样，Quote 没有移动操作意味着它的派生类也没有。

### 派生类中删除的拷贝控制与基类的关系

就像其他任何类的情况一样，基类或派生类也能出于同样的原因将其合成的默认构造函数或者任何一个拷贝控制成员定义成被删除的函数（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。此外，某些定义基类的方式也可能导致有的派生类成员成为被删除的函数：

- 如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的函数或者不可访问（参见 15.5 节，第 543 页），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。
- 和过去一样，编译器将不会合成一个删除掉的移动操作。当我们使用`=default`请求一个移动操作时，如果基类中的对应操作是删除的或不可访问的，那么派生类中该函数将是被删除的，原因是派生类对象的基类部分不可移动。同样，如果基类的析构函数是删除的或不可访问的，则派生类的移动构造函数也将是被删除的。

举个例子，对于下面的基类 B 来说：

```
class B {
public:
    B();
```

C++  
11

624

C++  
11

```

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

```

基类 B 含有一个可访问的默认构造函数和一个显式删除的拷贝构造函数。因为我们定义了拷贝构造函数，所以编译器将不会为 B 合成一个移动构造函数(参见 13.6.2 节，第 475 页)。因此，我们既不能移动也不能拷贝 B 的对象。如果 B 的派生类希望它自己的对象能被移动和拷贝，则派生类需要自定义相应版本的构造函数。当然，在这一过程中派生类还必须考虑如何移动或拷贝其基类部分的成员。在实际编程过程中，如果在基类中没有默认、拷贝或移动构造函数，则一般情况下派生类也不会定义相应的操作。

### 625 移动操作与继承

如前所述，大多数基类都会定义一个虚析构函数。因此在默认情况下，基类通常不含有合成的移动操作，而且在它的派生类中也没有合成的移动操作。

因为基类缺少移动操作会阻止派生类拥有自己的合成移动操作，所以当我们确实需要执行移动操作时应该首先在基类中进行定义。我们的 `Quote` 可以使用合成的版本，不过前提是 `Quote` 必须显式地定义这些成员。一旦 `Quote` 定义了自己的移动操作，那么它必须同时显式地定义拷贝操作(参见 13.6.2 节，第 476 页)：

```

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
};

```

通过上面的定义，我们就能对 `Quote` 的对象逐成员地分别进行拷贝、移动、赋值和销毁操作了。而且除非 `Quote` 的派生类中含有排斥移动的成员，否则它将自动获得合成的移动操作。

#### 15.7.2 节练习

**练习 15.25:** 我们为什么为 `Disc_quote` 定义一个默认构造函数？如果去除掉该构造函数的话会对 `Bulk_quote` 的行为产生什么影响？



#### 15.7.3 派生类的拷贝控制成员

如我们在 15.2.2 节(第 531 页)介绍过的，派生类构造函数在其初始化阶段中不但要初始化派生类自己的成员，还负责初始化派生类对象的基类部分。因此，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。类似的，派生类赋值运算符也必须为其基类部分的成员赋值。

和构造函数及赋值运算符不同的是，析构函数只负责销毁派生类自己分配的资源。如前所述，对象的成员是被隐式销毁的（参见 13.1.3 节，第 445 页）；类似的，派生类对象的基类部分也是自动销毁的。



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。

&lt;626

## 定义派生类的拷贝或移动构造函数



当为派生类定义拷贝或移动构造函数时（参见 13.1.1 节，第 440 页和 13.6.2 节，第 473 页），我们通常使用对应的基类构造函数初始化对象的基类部分：

```
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，我们必须在构造函数初始值列表中
    // 显式地调用该构造函数
    D(const D& d): Base(d)           // 拷贝基类成员
        /* D 的成员的初始值 */ { /* ... */ }
    D(D&& d): Base(std::move(d))      // 移动基类成员
        /* D 的成员的初始值 */ { /* ... */ }
};
```

初始值 `Base(d)` 将一个 `D` 对象传递给基类构造函数。尽管从道理上来说，`Base` 可以包含一个参数类型为 `D` 的构造函数，但是在实际编程过程中通常不会这么做。相反，`Base(d)` 一般会匹配 `Base` 的拷贝构造函数。`D` 类型的对象 `d` 将被绑定到该构造函数的 `Base&` 形参上。`Base` 的拷贝构造函数负责将 `d` 的基类部分拷贝给要创建的对象。假如我们没有提供基类的初始值的话：

```
// D 的这个拷贝构造函数很可能是不正确的定义
// 基类部分被默认初始化，而非拷贝
D(const D& d) /* 成员初始值，但是没有提供基类初始值 */
{ /* ... */ }
```

在上面的例子中，`Base` 的默认构造函数将被用来初始化 `D` 对象的基类部分。假定 `D` 的构造函数从 `d` 中拷贝了派生类成员，则这个新构建的对象的配置将非常奇怪：它的 `Base` 成员被赋予了默认值，而 `D` 成员的值则是从其他对象拷贝得来的。



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式地使用基类的拷贝（或移动）构造函数。

&lt;627

## 派生类赋值运算符

与拷贝和移动构造函数一样，派生类的赋值运算符（参见 13.1.2 节，第 443 页和 13.6.2 节，第 474 页）也必须显式地为其基类部分赋值：

```
// Base::operator=(const Base&) 不会被自动调用
```

&lt;627

```
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;
}
```

上面的运算符首先显式地调用基类赋值运算符，令其为派生类对象的基类部分赋值。基类的运算符（应该可以）正确地处理自赋值的情况，如果赋值命令是正确的，则基类运算符将释放掉其左侧运算对象的基类部分的旧值，然后利用 `rhs` 为其赋一个新值。随后，我们继续进行其他为派生类成员赋值的工作。

值得注意的是，无论基类的构造函数或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。例如，对于 `Base::operator=` 的调用语句将执行 `Base` 的拷贝赋值运算符，至于该运算符是由 `Base` 显式定义的还是由编译器合成的无关紧要。

### 派生类析构函数

如前所述，在析构函数体执行完成后，对象的成员会被隐式销毁（参见 13.1.3 节，第 445 页）。类似的，对象的基类部分也是隐式销毁的。因此，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：

```
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 */ }
};
```

对象销毁的顺序正好与其创建的顺序相反：派生类析构函数首先执行，然后是基类的析构函数，以此类推，沿着继承体系的反方向直至最后。

### 在构造函数和析构函数中调用虚函数

如我们所知，派生类对象的基类部分将首先被构建。当执行基类的构造函数时，该对象的派生类部分是未被初始化的状态。类似的，销毁派生类对象的次序正好相反，因此当执行基类的析构函数时，派生类部分已经被销毁掉了。由此可知，当我们执行上述基类成员的时候，该对象处于未完成的状态。

为了能够正确地处理这种未完成状态，编译器认为对象的类型在构造或析构的过程中仿佛发生了改变一样。也就是说，当我们构建一个对象时，需要把对象的类和构造函数的类看作是同一个；对虚函数的调用绑定正好符合这种把对象的类和构造函数的类看成同一个的要求；对于析构函数也是同样的道理。上述的绑定不但对直接调用虚函数有效，对间接调用也是有效的，这里的间接调用是指通过构造函数（或析构函数）调用另一个函数。  
628>

为了理解上述行为，不妨考虑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，毕竟，如果它不需要访问派生类成员的话，则派生类直接使用基类的虚函数版本就可以了。然而，当执行基类构造函数时，它要用到的派生类成员尚未初始化，如果我们允许这样的访问，则程序很可能会崩溃。



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。

### 15.7.3 节练习

**练习 15.26:** 定义 `Quote` 和 `Bulk_quote` 的拷贝控制成员，令其与合成的版本行为一致。为这些成员以及其他构造函数添加打印状态的语句，使得我们能够知道正在运行哪个程序。使用这些类编写程序，预测程序将创建和销毁哪些对象。重复实验，不断比较你的预测和实际输出结果是否相同，直到预测完全准确再结束。

### 15.7.4 继承的构造函数

在 C++11 新标准中，派生类能够重用其直接基类定义的构造函数。尽管如我们所知，这些构造函数并非以常规的方式继承而来，但是为了方便，我们不妨姑且称其为“继承”的。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。如果派生类没有直接定义这些构造函数，则编译器将为派生类合成它们。

派生类继承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。举个例子，我们可以重新定义 `Bulk_quote` 类（参见 15.4 节，第 541 页），令其继承 `Disc_quote` 类的构造函数：

```
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;
};
```

通常情况下，`using` 声明语句只是令某个名字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。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。换句话说，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。

这些编译器生成的构造函数形如：

```
derived(parms) : base(args) { }
```

其中，`derived` 是派生类的名字，`base` 是基类的名字，`parms` 是构造函数的形参列表，`args` 将派生类构造函数的形参传递给基类的构造函数。在我们的 `Bulk_quote` 类中，继承的构造函数等价于：

```
Bulk_quote(const std::string& book, double price,
           std::size_t qty, double disc):
    Disc_quote(book, price, qty, disc) { }
```

如果派生类含有自己的数据成员，则这些成员将被默认初始化（参见 7.1.4 节，第 238 页）。

### 继承的构造函数的特点

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。例如，不管 `using` 声明出现在哪儿，基类的私有构造函数在派生类中还是一个私有构造函数；受保护的构造函数和公有构造函数也是同样的规则。

而且，一个 `using` 声明语句不能指定 `explicit` 或 `constexpr`。如果基类的构造函数是 `explicit`（参见 7.5.4 节，第 265 页）或者 `constexpr`（参见 7.5.6 节，第 267

C++  
11

629

页), 则继承的构造函数也拥有相同的属性。

当一个基类构造函数含有默认实参(参见 6.5.1 节, 第 211 页)时, 这些实参并不会被继承。相反, 派生类将获得多个继承的构造函数, 其中每个构造函数分别省略掉一个含有默认实参的形参。例如, 如果基类有一个接受两个形参的构造函数, 其中第二个形参含有默认实参, 则派生类将获得两个构造函数: 一个构造函数接受两个形参(没有默认实参), 另一个构造函数只接受一个形参, 它对应于基类中最左侧的没有默认值的那个形参。

如果基类含有几个构造函数, 则除了两个例外情况, 大多数时候派生类会继承所有这些构造函数。第一个例外是派生类可以继承一部分构造函数, 而为其他构造函数定义自己的版本。如果派生类定义的构造函数与基类的构造函数具有相同的参数列表, 则该构造函数将不会被继承。定义在派生类中的构造函数将替换继承而来的构造函数。

第二个例外是默认、拷贝和移动构造函数不会被继承。这些构造函数按照正常规则被合成。继承的构造函数不会被作为用户定义的构造函数来使用, 因此, 如果一个类只含有继承的构造函数, 则它也将拥有一个合成的默认构造函数。

#### 15.7.4 节练习

**练习 15.27:** 重新定义你的 Bulk\_quote 类, 令其继承构造函数。



## 15.8 容器与继承

630 >

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。

举个例子, 假定我们想定义一个 vector, 令其保存用户准备购买的几种书籍。显然我们不应该用 vector 保存 Bulk\_quote 对象。因为我们不能将 Quote 对象转换成 Bulk\_quote (参见 15.2.3 节, 第 534 页), 所以我们将无法把 Quote 对象放置在该 vector 中。

其实, 我们也不应该使用 vector 保存 Quote 对象。此时, 虽然我们可以把 Bulk\_quote 对象放置在容器中, 但是这些对象再也不是 Bulk\_quote 对象了:

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// 正确: 但是只能把对象的 Quote 部分拷贝给 basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本, 打印 750, 即 15 * $50
cout << basket.back().net_price(15) << endl;
```

basket 的元素是 Quote 对象, 因此当我们向该 vector 中添加一个 Bulk\_quote 对象时, 它的派生类部分将被忽略掉 (参见 15.2.3 节, 第 535 页)。



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（参见 12.1 节，第 400 页））。和往常一样，这些指针所指对象的动态类型可能是基类类型，也可能是派生类类型：

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// 调用 Quote 定义的版本；打印 562.5，即在 15*50 中扣除掉折扣金额
cout << basket.back()->net_price(15) << endl;
```

因为 `basket` 存放着 `shared_ptr`，所以我们必须解引用 `basket.back()` 的返回值以获得运行 `net_price` 的对象。我们通过在 `net_price` 的调用中使用 `->` 以达到这个目的。如我们所知，实际调用的 `net_price` 版本依赖于指针所指对象的动态类型。

值得注意的是，我们将 `basket` 定义成 `shared_ptr<Quote>`，但是在第二个 `push_back` 中传入的是一个 `Bulk_quote` 对象的 `shared_ptr`。正如我们可以将一个派生类的普通指针转换成基类指针一样（参见 15.2.2 节，第 530 页），我们也能把一个派生类的智能指针转换成基类的智能指针。在此例中，`make_shared<Bulk_quote>` 返回一个 `shared_ptr<Bulk_quote>` 对象，当我们调用 `push_back` 时该对象被转换成 `shared_ptr<Quote>`。因此尽管在形式上有所差别，但实际上 `basket` 的所有元素的类型都是相同的。

&lt;631

## 15.8 节练习

**练习 15.28：** 定义一个存放 `Quote` 对象的 `vector`，将 `Bulk_quote` 对象传入其中。  
计算 `vector` 中所有元素总的 `net_price`。

**练习 15.29：** 再运行一次你的程序，这次传入 `Quote` 对象的 `shared_ptr`。如果这次计算出的总额与之前的程序不一致，解释为什么；如果一致，也请说明原因。

### 15.8.1 编写 Basket 类



对于 C++ 面向对象的编程来说，一个悖论是我们无法直接使用对象进行面向对象编程。相反，我们必须使用指针和引用。因为指针会增加程序的复杂性，所以我们经常定义一些辅助的类来处理这种复杂情况。首先，我们定义一个表示购物篮的类：

```
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};
};
```

我们的类使用一个 `multiset` (参见 11.2.1 节, 第 377 页) 来存放交易信息, 这样我们就能保存同一本书的多条交易记录, 而且对于一本给定的书籍, 它的所有交易信息都保存在一起 (参见 11.2.2 节, 第 378 页)。

`multiset` 的元素是 `shared_ptr`。因为 `shared_ptr` 没有定义小于运算符, 所以为了对元素排序我们必须提供自己的比较运算符 (参见 11.2.2 节, 第 378 页)。在此例中, 我们定义了一个名为 `compare` 的私有静态成员, 该成员负责比较 `shared_ptr` 所指的对象的 `isbn`。我们初始化 `multiset`, 通过类内初始值调用比较函数 (参见 7.3.1 节, 第 246 页):

632 // multiset 保存多个报价, 按照 compare 成员排序  

```
std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
    items{compare};
```

这个声明看起来不太容易理解, 但是从左向右读的话, 我们就能明白它其实是定义了一个指向 `Quote` 对象的 `shared_ptr` 的 `multiset`。这个 `multiset` 将使用一个与 `compare` 成员类型相同的函数来对其中的元素进行排序。`multiset` 成员的名字是 `items`, 我们初始化 `items` 并令其使用我们的 `compare` 函数。

## 定义 Basket 的成员

`Basket` 类只定义两个操作。第一个成员是我们在类的内部定义的 `add_item` 成员, 该成员接受一个指向动态分配的 `Quote` 的 `shared_ptr`, 然后将这个 `shared_ptr` 放置在 `multiset` 中。第二个成员的名字是 `total_receipt`, 它负责将购物篮的内容逐项打印成清单, 然后返回购物篮中所有物品的总价格:

```
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位置
    for (auto iter = items.cbegin();
        iter != items.cend();
        iter = items.upper_bound(*iter)) {
        // 我们知道在当前的 Basket 中至少有一个该关键字的元素
        // 打印该书籍对应的项目
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl; // 打印最终的总价格
    return sum;
}
```

我们的 `for` 循环首先定义并初始化 `iter`, 令其指向 `multiset` 的第一个元素。条件部分检查 `iter` 是否等于 `items.cend()`: 如果相等, 表明我们已经处理完了所有购买记录, 接下来应该跳出 `for` 循环; 否则, 如果不相等, 则继续处理下一本书籍。

比较有趣的是, `for` 循环中的“递增”表达式。与通常的循环语句依次读取每个元素不同, 我们直接令 `iter` 指向下一个关键字, 调用 `upper_bound` 函数可以令我们跳过与当前关键字相同的所有元素 (参见 11.3.5 节, 第 390 页)。对于 `upper_bound` 函数来说, 它返回的是一个迭代器, 该迭代器指向所有与 `iter` 关键字相等的元素中最后一个元素的

下一位置。因此，我们得到的迭代器或者指向集合的末尾，或者指向下一本书籍。

在 `for` 循环内部，我们通过调用 `print_total`（参见 15.1 节，第 527 页）来打印购物篮中每本书籍的细节：

```
sum += print_total(os, **iter, items.count(*iter));
```

`print_total` 的实参包括一个用于写入数据的 `ostream`、一个待处理的 `Quote` 对象和一个计数值。当我们解引用 `iter` 后将得到一个指向准备打印的对象的 `shared_ptr`。为了得到这个对象，必须解引用该 `shared_ptr`。因此，`**iter` 是一个 `Quote` 对象（或者 `Quote` 的派生类的对象）。我们使用 `multiset` 的 `count` 成员（参见 11.3.5 节，第 388 页）来统计在 `multiset` 中有多少元素的键值相同（即 ISBN 相同）。 633

如我们所知，`print_total` 调用了虚函数 `net_price`，因此最终的计算结果依赖于 `**iter` 的动态类型。`print_total` 函数打印并返回给定书籍的总价格，我们把这个结果添加到 `sum` 当中，最后当循环结束后打印 `sum`。

## 隐藏指针

`Basket` 的用户仍然必须处理动态内存，原因是 `add_item` 需要接受一个 `shared_ptr` 参数。因此，用户不得不按照如下形式编写代码：

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

我们的下一步是重新定义 `add_item`，使得它接受一个 `Quote` 对象而非 `shared_ptr`。新版本的 `add_item` 将负责处理内存分配，这样它的用户就不必再受困于此了。我们将定义两个版本，一个拷贝它给定的对象，另一个则采取移动操作（参见 13.6.3 节，第 481 页）：

```
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
```

唯一的问题是 `add_item` 不知道要分配的类型。当 `add_item` 进行内存分配时，它将拷贝（或移动）它的 `sale` 参数。在某处可能会有一条如下形式的 `new` 表达式：

```
new Quote(sale)
```

不幸的是，这条表达式所做的工作可能是不正确的：`new` 为我们请求的类型分配内存，因此这条表达式将分配一个 `Quote` 类型的对象并且拷贝 `sale` 的 `Quote` 部分。然而，`sale` 实际指向的可能是 `Bulk_quote` 对象，此时，该对象将被迫切掉一部分。

## 模拟虚拷贝



为了解决上述问题，我们给 `Quote` 类添加一个虚函数，该函数将申请一份当前对象的拷贝。

```
class Quote {
public:
    // 该虚函数返回当前对象的一份动态分配的拷贝
    // 这些成员使用的引用限定符参见 13.6.3 节（第 483 页）
    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
};
```

634 >

```
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
};
```

因为我们拥有 `add_item` 的拷贝和移动版本，所以我们分别定义 `clone` 的左值和右值版本(参见 13.6.3 节, 第 483 页)。每个 `clone` 函数分配当前类型的一个新对象，其中，`const` 左值引用成员将它自己拷贝给新分配的对象；右值引用成员则将自己移动到新数据中。

我们可以使用 `clone` 很容易地写出新版本的 `add_item`:

```
class Basket {
public:
    void add_item(const Quote& sale)      // 拷贝给定的对象
    { items.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
};
```

和 `add_item` 本身一样，`clone` 函数也根据作用于左值还是右值而分为不同的重载版本。在此例中，第一个 `add_item` 函数调用 `clone` 的 `const` 左值版本，第二个函数调用 `clone` 的右值引用版本。在右值版本中，尽管 `sale` 的类型是右值引用类型，但实际上 `sale` 本身(和任何其他变量一样) 是个左值(参见 13.6.1 节, 第 471 页)。因此，我们调用 `move` 把一个右值引用绑定到 `sale` 上。

我们的 `clone` 函数也是一个虚函数。`sale` 的动态类型(通常)决定了到底运行 `Quote` 的函数还是 `Bulk_quote` 的函数。无论我们是拷贝还是移动数据，`clone` 都返回一个新分配对象的指针，该对象与 `clone` 所属的类型一致。我们把一个 `shared_ptr` 绑定到这个对象上，然后调用 `insert` 将这个新分配的对象添加到 `items` 中。注意，因为 `shared_ptr` 支持派生类向基类的类型转换(参见 15.2.2 节, 第 530 页)，所以我们将把 `shared_ptr<Quote>` 绑定到 `Bulk_quote*` 上。

### 15.8.1 节练习

**练习 15.30:** 编写你自己的 `Basket` 类，用它计算上一个练习中交易记录的总价格。

## 15.9 文本查询程序再探

接下来，我们扩展 12.3 节(第 430 页)的文本查询程序，用它作为说明继承的最后一个例子。在上一版的程序中，我们可以查询在文件中某个指定单词的出现情况。我们将在本节扩展该程序使其支持更多更复杂的查询操作。在后面的例子中，我们将针对下面这个小故事展开查询：

```
Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
```

```

A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
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
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.

...

```

- 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.

```

此外，我们还希望能够混合使用这些运算符，比如：

```
fiery & bird | wind
```

在类似这样的例子中，我们将使用 C++ 通用的优先级规则（参见 4.1.2 节，第 121 页）对复杂表达式求值。因此，这条查询语句所得行应该是如下二者之一：在该行中或者 `fiery` 和 `bird` 同时出现，或者出现了 `wind`：

```

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,

```

636

在输出内容中首先是那条查询语句，我们使用圆括号来表示查询被解释和执行的次序。与之前实现的版本一样，接下来系统将按照查询结果中行号的升序显示结果并且每一行只显示一次。

### 15.9.1 面向对象的解决方案

我们可能会认为使用 12.3.2 节（第 432 页）的 `TextQuery` 类来表示单词查询，然后

从该类中派生出其他查询是一种可行的方案。

然而，这样的设计实际上存在缺陷。为了理解其中的原因，我们不妨考虑逻辑非查询。单词查询查找一个指定的单词，为了让逻辑非查询按照单词查询的方式执行，我们将不得不定义逻辑非查询所要查找的单词。但是在一般情况下，我们无法得到这样的单词。相反，一个逻辑非查询中含有一个结果值需要取反的查询语句（单词查询或任何其他查询）；类似的，一个逻辑与查询和一个逻辑或查询各包含两个结果值需要合并的查询语句。

由上述观察结果可知，我们应该将几种不同的查询建模成相互独立的类，这些类共享一个公共基类：

```
WordQuery      // Daddy
NotQuery       // ~Alice
OrQuery        // hair | Alice
AndQuery       // hair & Alice
```

这些类将只包含两个操作：

- eval，接受一个 `TextQuery` 对象并返回一个 `QueryResult`，`eval` 函数使用给定的 `TextQuery` 对象查找与之匹配的行。
- rep，返回基础查询的 `string` 表示形式，`eval` 函数使用 `rep` 创建一个表示匹配结果的 `QueryResult`，输出运算符使用 `rep` 打印查询表达式。

### 关键概念：继承与组合

继承体系的设计本身是一个非常复杂的问题，已经超出了本书的范围。然而，有一条设计准则非常重要也非常基础，每个程序员都应该熟悉它。

当我们令一个类公有地继承另一个类时，派生类应当反映与基类的“是一种 (Is A)”关系。在设计良好的类体系当中，公有派生类的对象应该可以用在任何需要基类对象的地方。

类型之间的另一种常见关系是“有一个 (Has A)”关系，具有这种关系的类暗含成员的意思。

在我们的书店示例中，基类表示的是按规定价格销售的书籍的报价。`Bulk_quote` “是一种” 报价结果，只不过它使用的价格策略不同。我们的书店类都“有一个” 价格成员和 `ISBN` 成员。

## 抽象基类

如我们所知，在这四种查询之间并不存在彼此的继承关系，从概念上来说它们互为兄弟。因为所有这些类都共享同一个接口，所以我们需要定义一个抽象基类（参见 15.4 节，第 541 页）来表示该接口。我们将所需的抽象基类命名为 `Query_base`，以此来表示它的角色是整个查询继承体系的根节点。

我们的 `Query_base` 类将把 `eval` 和 `rep` 定义成纯虚函数（参见 15.4 节，第 541 页），其他代表某种特定查询类型的类必须覆盖这两个函数。我们将从 `Query_base` 直接派生出 `WordQuery` 和 `NotQuery`。`AndQuery` 和 `OrQuery` 都具有系统中其他类所不具备的一个特殊属性：它们各自包含两个运算对象。为了对这种属性建模，我们定义另外一个名为 `BinaryQuery` 的抽象基类，该抽象基类用于表示含有两个运算对象的查询。`AndQuery` 和 `OrQuery` 继承自 `BinaryQuery`，而 `BinaryQuery` 继承自 `Query_base`。由这些分

析我们将得到如图 15.2 所示的类设计结果：

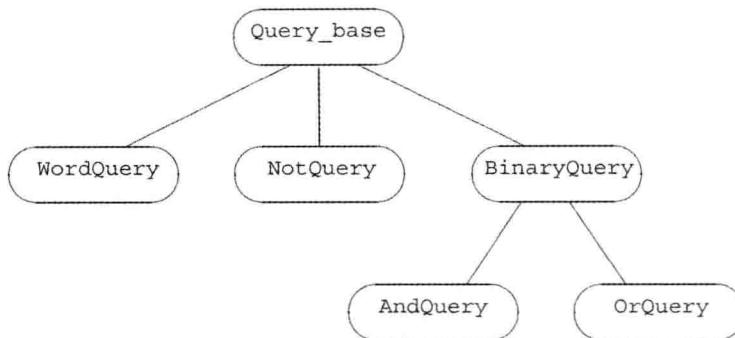


图 15.2: Query\_base 继承体系

### 将层次关系隐藏于接口类中

我们的程序将致力于计算查询结果，而非仅仅构建查询的体系。为了使程序能正常运行，我们必须首先创建查询命令，最简单的办法是编写 C++ 表达式。例如，可以编写下面的代码来生成之前描述的复合查询：

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

如上所述，其隐含的意思是用户层代码将不会直接使用这些继承的类；相反，我们将定义一个名为 `Query` 的接口类，由它负责隐藏整个继承体系。`Query` 类将保存一个 `Query_base` 指针，该指针绑定到 `Query_base` 的派生类对象上。`Query` 类与 `Query_base` 类提供的操作是相同的：`eval` 用于求查询的结果，`rep` 用于生成查询的 `string` 版本，同时 `Query` 也会定义一个重载的输出运算符用于显示查询。

用户将通过 `Query` 对象的操作间接地创建并处理 `Query_base` 对象。我们定义 `Query` 对象的三个重载运算符以及一个接受 `string` 参数的 `Query` 构造函数，这些函数动态分配一个新的 `Query_base` 派生类的对象：

- & 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。

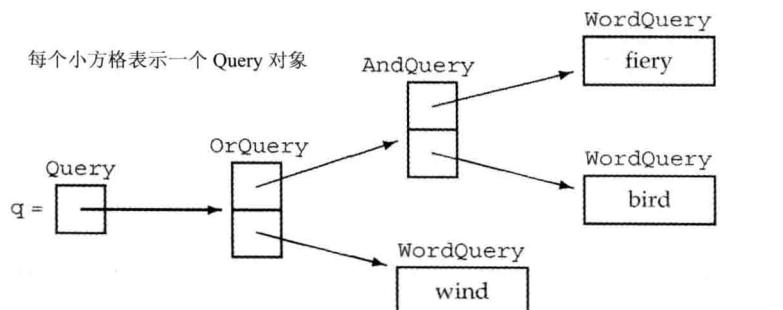


图 15.3: 使用 Query 表达式创建的对象

## 理解这些类的工作机理

在这个应用程序中，很大一部分工作是构建代表用户查询的对象，对于读者来说认识到这一点非常重要。例如，像上面这样的表达式将生成如图 15.3 所示的一系列相关对象的集合。

一旦对象树构建完成后，对某一条查询语句的求值（或生成表示形式的）过程基本上就转换为沿着箭头方向依次对每个对象求值（或显示）的过程（由编译器为我们组织管理）。

**639** 例如，如果我们对 q（即树的根节点）调用 eval 函数，则该调用语句将令 q 所指的 OrQuery 对象 eval 它自己。对该 OrQuery 求值实际上是对它的两个运算对象执行 eval 操作：一个运算对象是 AndQuery，另一个是查找单词 wind 的 WordQuery。接下来，对 AndQuery 求值转化为对它的两个 WordQuery 求值，分别生成单词 fiery 和 bird 的查询结果。

对于面向对象编程的新手来说，要想理解一个程序，最困难的部分往往是理解程序的设计思路。一旦你掌握了程序的设计思路，接下来的实现也就水到渠成了。为了帮助读者理解程序设计的过程，我们在表 15.1 中整理了之前那个例子用到的类，并对其进行了简要的描述。

640

表 15.1：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（12.3.2 节，第 432 页）
QueryResult	该类保存一个 query 操作的结果（12.3.2 节，第 433 页）
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

### 15.9.1 节练习

练习 15.31：已知 s1、s2、s3 和 s4 都是 string，判断下面的表达式分别创建了什么样的对象：

- (a) Query(s1) | Query(s2) & ~ Query(s3);
- (b) Query(s1) | (Query(s2) & ~ Query(s3));
- (c) (Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));

### 15.9.2 Query\_base 类和 Query 类

下面我们开始程序的实现过程，首先定义 `Query_base` 类：

```
// 这是一个抽象基类，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 = TextQuery::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;
};
```

`eval` 和 `rep` 都是纯虚函数，因此 `Query_base` 是一个抽象基类（参见 15.4 节，第 541 页）。因为我们不希望用户或者派生类直接使用 `Query_base`，所以它没有 `public` 成员。所有对 `Query_base` 的使用都需要通过 `Query` 对象，因为 `Query` 需要调用 `Query_base` 的虚函数，所以我们将 `Query` 声明成 `Query_base` 的友元。

受保护的成员 `line_no` 将在 `eval` 函数内部使用。类似的，析构函数也是受保护的，因为它将（隐式地）在派生类析构函数中使用。

### Query 类

`Query` 类对外提供接口，同时隐藏了 `Query_base` 的继承体系。每个 `Query` 对象都含有一个指向 `Query_base` 对象的 `shared_ptr`。因为 `Query` 是 `Query_base` 的唯一接口，所以 `Query` 必须定义自己的 `eval` 和 `rep` 版本。

接受一个 `string` 参数的 `Query` 构造函数将创建一个新的 `WordQuery` 对象，然后将它的 `shared_ptr` 成员绑定到这个新创建的对象上。`&`、`|` 和 `~` 运算符分别创建 `AndQuery`、`OrQuery` 和 `NotQuery` 对象，这些运算符将返回一个绑定到新创建的对象上的 `Query` 对象。为了支持这些运算符，`Query` 还需要另外一个构造函数，它接受指向 `Query_base` 的 `shared_ptr` 并且存储给定的指针。我们将这个构造函数声明为私有的，原因是不希望一般的用户代码能随便定义 `Query_base` 对象。因为这个构造函数是私有的，所以我们需要将三个运算符声明为友元。

在形成了上述设计思路后，`Query` 类本身就比较简单了：

```
// 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;
};

```

我们首先将创建 `Query` 对象的运算符声明为友元，之所以这么做是因为这些运算符需要访问那个私有构造函数。

在 `Query` 的公有接口部分，我们声明了接受 `string` 的构造函数，不过没有对其进行定义。因为这个构造函数将要创建一个 `WordQuery` 对象，所以我们应该首先定义 `WordQuery` 类，随后才能定义接受 `string` 的 `Query` 构造函数。

另外两个公有成员是 `Query_base` 的接口。其中，`Query` 操作使用它的 `Query_base` 指针来调用各自的 `Query_base` 虚函数。实际调用哪个函数版本将由 `q` 所指的对象类型决定，并且直到运行时才能最终确定下来。



## Query 的输出运算符

输出运算符可以很好地解释我们的整个查询系统是如何工作的：

```

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();
}

```

当我们打印一个 `Query` 时，输出运算符调用 `Query` 类的公有 `rep` 成员。运算符函数通过指针成员虚调用当前 `Query` 所指对象的 `rep` 成员。也就是说，当我们编写如下代码时：

```

Query andq = Query(sought1) & Query(sought2);
cout << andq << endl;

```

输出运算符将调用 `andq` 的 `Query::rep`，而 `Query::rep` 通过它的 `Query_base` 指针虚调用 `Query_base` 版本的 `rep` 函数。因为 `andq` 指向的是一个 `AndQuery` 对象，所以本次的函数调用将运行 `AndQuery::rep`。

### 15.9.2 节练习

**练习 15.32:** 当一个 `Query` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

**练习 15.33:** 当一个 `Query_base` 类型的对象被拷贝、移动、赋值或销毁时，将分别发生什么？

642 >

## 15.9.3 派生类

对于 `Query_base` 的派生类来说，最有趣的部分是这些派生类如何表示一个真实的查询。其中 `WordQuery` 类最直接，它的任务就是保存要查找的单词。

其他类分别操作一个或两个运算对象。`NotQuery` 有一个运算对象，`AndQuery` 和 `OrQuery` 有两个。在这些类当中，运算对象可以是 `Query_base` 的任意一个派生类的对象：一个 `NotQuery` 对象可以被用在 `WordQuery`、`AndQuery`、`OrQuery` 或另一个 `NotQuery` 中。为了支持这种灵活性，运算对象必须以 `Query_base` 指针的形式存储，

这样我们就能把该指针绑定到任何我们需要的具体类上。

然而，实际上我们的类并不存储 `Query_base` 指针，而是直接使用一个 `Query` 对象。就像用户代码可以通过接口类得到简化一样，我们也可以使用接口类来简化我们自己的类。

至此我们已经清楚了所有类的设计思路，接下来依次实现它们。

## WordQuery 类

一个 `WordQuery` 查找一个给定的 `string`，它是在给定的 `TextQuery` 对象上实际执行查询的唯一一个操作：

```
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
};
```

和 `Query_base` 一样，`WordQuery` 没有公有成员。同时，`Query` 必须作为 `WordQuery` 的友元，这样 `Query` 才能访问 `WordQuery` 的构造函数。

每个表示具体查询的类都必须定义继承而来的纯虚函数 `eval` 和 `rep`。我们在 `WordQuery` 类的内部定义这两个操作：`eval` 调用其 `TextQuery` 参数的 `query` 成员，由 `query` 成员在文件中实际进行查找；`rep` 返回这个 `WordQuery` 表示的 `string`（即 `query_word`）。

定义了 `WordQuery` 类之后，我们就能定义接受 `string` 的 `Query` 构造函数了：

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

这个构造函数分配一个 `WordQuery`，然后令其指针成员指向新分配的对象。

## NotQuery 类及~运算符

`~` 运算符生成一个 `NotQuery`，其中保存着一个需要对其取反的 `Query`：

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const 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));
}
```

643

因为 `NotQuery` 的所有成员都是私有的，所以我们一开始就要把`~`运算符设定为友元。为

了 rep 一个 NotQuery，我们需要将~符号与基础的 Query 连接在一起。我们在输出的结果中加上适当的括号，这样读者就可以清楚地知道查询的优先级了。

值得注意的是，在 NotQuery 自己的 rep 成员中对 rep 的调用最终执行的是一个虚调用：query.rep() 是对 Query 类 rep 成员的非虚调用，接着 Query::rep 将调用 q->rep()，这是一个通过 Query\_base 指针进行的虚调用。

~运算符动态分配一个新的 NotQuery 对象，其 return 语句隐式地使用接受一个 shared\_ptr<Query\_base> 的 Query 构造函数。也就是说，return 语句等价于：

```
// 分配一个新的 NotQuery 对象
// 将所得的 NotQuery 指针绑定到一个 shared_ptr<Query_base>
shared_ptr<Query_base> tmp(new NotQuery(expr));
return Query(tmp);           // 使用接受一个 shared_ptr 的 Query 构造函数
```

eval 成员比较复杂，因此我们将在类的外部实现它，15.9.4 节（第 573 页）将专门介绍如何定义 eval 函数。

## BinaryQuery 类

BinaryQuery 类也是一个抽象基类，它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), opSym(s) { }
    // 抽象类: 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
};
```

**644** BinaryQuery 中的数据是两个运算对象及相应的运算符符号，构造函数负责接受两个运算对象和一个运算符符号，然后将它们存储在对应的数据成员中。

要想 rep 一个 BinaryQuery，我们需要生成一个带括号的表达式。表达式的内容依次包括左侧运算对象、运算符以及右侧运算对象。就像我们显示 NotQuery 的方法一样，对 rep 的调用最终是对 lhs 和 rhs 所指 Query\_base 对象的 rep 函数进行虚调用。



BinaryQuery 不定义 eval，而是继承了该纯虚函数。因此，BinaryQuery 也是一个抽象基类，我们不能创建 BinaryQuery 类型的对象。

## AndQuery 类、OrQuery 类及相应的运算符

AndQuery 类和 OrQuery 类以及它们的运算符都非常相似：

```
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, const 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));
}

```

这两个类将各自的运算符定义成友元，并且各自定义了一个构造函数通过运算符创建 BinaryQuery 基类部分。它们继承 BinaryQuery 的 rep 函数，但是覆盖了 eval 函数。

和~运算符一样，&和|运算符也返回一个绑定到新分配对象上的 shared\_ptr。在这些运算符中，return 语句负责将 shared\_ptr 转换成 Query。

### 15.9.3 节练习

645

**练习 15.34:** 针对图 15.3（第 565 页）构建的表达式：

- 列举出在处理表达式的过程中执行的所有构造函数。
- 列举出 cout<<q 所调用的 rep。
- 列举出 q.eval() 所调用的 eval。

**练习 15.35:** 实现 Query 类和 Query\_base 类，其中需要定义 rep 而无须定义 eval。

**练习 15.36:** 在构造函数和 rep 成员中添加打印语句，运行你的代码以检验你对本节第一个练习中 (a)、(b) 两小题的回答是否正确。

**练习 15.37:** 如果在派生类中含有 shared\_ptr<Query\_base>类型的成员而非 Query 类型的成员，则你的类需要做出怎样的改变？

**练习 15.38:** 下面的声明合法吗？如果不合法，请解释原因；如果合法，请指出该声明的含义。

```

BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");

```

### 15.9.4 eval 函数

eval 函数是我们这个查询系统的核心。每个 eval 函数作用于各自的运算对象，同时遵循的内在逻辑也有所区别：OrQuery 的 eval 操作返回两个运算对象查询结果的并集，而 AndQuery 返回交集。与它们相比，NotQuery 的 eval 函数更加复杂一些：它需要返回运算对象没有出现的文本行。

为了支持上述 eval 函数的处理，我们需要使用 QueryResult，在它当中定义了 12.3.2 节练习（第 435 页）添加的成员。假设 QueryResult 包含 begin 和 end 成员，它们允许我们在 QueryResult 保存的行号 set 中进行迭代；另外假设 QueryResult 还包含一个名为 get\_file 的成员，它返回一个指向待查询文件的 shared\_ptr。



我们的 Query 类使用了 12.3.2 节练习(第 435 页)为 QueryResult 定义的成员。

### OrQuery::eval

一个 OrQuery 表示的是它的两个运算对象结果的并集，对于每个运算对象来说，我们通过调用 eval 得到它的查询结果。因为这些运算对象的类型是 Query，所以调用 eval 也就是调用 Query::eval，而后者实际上是对潜在的 query\_base 对象的 eval 进行虚调用。每次调用完成后，得到的结果是一个 QueryResult，它表示运算对象出现的行号。我们把这些行号组织在一个新 set 中：

646 &gt;

```
// 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());
}
```

我们使用接受一对迭代器的 set 构造函数初始化 ret\_lines。一个 QueryResult 的 begin 和 end 成员返回行号 set 的迭代器，因此，创建 ret\_lines 的过程实际上是拷贝了 left 集合的元素。接下来对 ret\_lines 调用 insert，并将 right 的元素插入进来。调用结束后，ret\_lines 将包含在 left 或 right 中出现过的所有行号。

eval 函数在最后构建并返回一个表示混合查询匹配的 QueryResult。QueryResult 的构造函数（参见 12.3.2 节，第 434 页）接受三个实参：一个表示查询的 string、一个指向匹配行号 set 的 shared\_ptr 和一个指向输入文件 vector 的 shared\_ptr。我们调用 rep 生成所需的 string，调用 get\_file 获取指向文件的 shared\_ptr。因为 left 和 right 指向的是同一个文件，所以使用哪个执行 get\_file 函数并不重要。

### AndQuery::eval

AndQuery 的 eval 和 OrQuery 很类似，唯一的区别是它调用了一个标准库算法来求得两个查询结果中共有的行：

```
// 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，以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_intersection(left.begin(), left.end(),
                 right.begin(), right.end(),
                 inserter(*ret_lines, ret_lines->begin()));
return QueryResult(rep(), ret_lines, left.get_file());
}

```

其中我们使用标准库算法 `set_intersection` 来合并两个 `set`，关于 647  
`set_intersection` 在附录 A.2.8 (第 779 页) 中有详细的描述。

`set_intersection` 算法接受五个迭代器。它使用前四个迭代器表示两个输入序列 (参见 10.5.2 节, 第 368 页), 最后一个实参表示目的位置。该算法将两个输入序列中共同出现的元素写入到目的位置中。

在上述调用中我们传入一个插入迭代器 (参见 10.4.1 节, 第 357 页) 作为目的位置。当 `set_intersection` 向这个迭代器写入内容时, 实际上是向 `ret_lines` 插入一个新元素。

和 `OrQuery` 的 `eval` 函数一样, `AndQuery` 的 `eval` 函数也在最后构建并返回一个表示混合查询匹配的 `QueryResult`。

### NotQuery::eval

`NotQuery` 查找运算对象没有出现的文本行:

```

// 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<set<line_no>>();
    // 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, 如果该行不在 result 当中, 则将其添加到 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());
}

```

和其他 `eval` 函数一样, 我们首先对当前的运算对象调用 `eval`, 所得的结果

`QueryResult` 中包含的是运算对象出现的行号，但我们想要的是运算对象未出现的行号。也就是说，我们需要的是存在于文件中，但是不在 `result` 中的行。

要想得到最终的结果，我们需要遍历不超过输出文件大小的所有整数，并将所有不在 `result` 中的行号放入到 `ret_lines` 中。我们使用 `beg` 和 `end` 分别表示 `result` 的第一个元素和最后一个元素的下一位置。因为遍历的对象是一个 `set`，所以当遍历结束后获得的行号将按照升序排列。

648 循环体负责检查当前的编号是否在 `result` 当中。如果不在，将这个数字添加到 `ret_lines` 中；如果该数字属于 `result`，则我们递增 `result` 的迭代器 `beg`。

一旦处理完所有行号，就返回包含 `ret_lines` 的一个 `QueryResult` 对象；和之前版本的 `eval` 类似，该 `QueryResult` 对象还包含 `rep` 和 `get_file` 的运行结果。

#### 15.9.4 节练习

**练习 15.39：**实现 `Query` 类和 `Query_base` 类，求图 15.3（第 565 页）中表达式的值并打印相关信息，验证你的程序是否正确。

**练习 15.40：**在 `OrQuery` 的 `eval` 函数中，如果 `rhs` 成员返回的是空集将发生什么？如果 `lhs` 是空集呢？如果 `lhs` 和 `rhs` 都是空集又将发生什么？

**练习 15.41：**重新实现你的类，这次使用指向 `Query_base` 的内置指针而非 `shared_ptr`。请注意，做出上述改动后你的类将不能再使用合成的拷贝控制成员。

**练习 15.42：**从下面的几种改进中选择一种，设计并实现它：

- (a) 按句子查询并打印单词，而不再是按行打印。
- (b) 引入一个历史系统，用户可以按编号查阅之前的某个查询，并可以在其中增加内容或者将其与其他查询组合。
- (c) 允许用户对结果做出限制，比如从给定范围的行中挑出匹配的进行显示。

## 小结

&lt; 649

继承使得我们可以编写一些新的类，这些新类既能共享其基类的行为，又能根据需要覆盖或添加行为。动态绑定使得我们可以忽略类型之间的差异，其机理是在运行时根据对象的动态类型来选择运行函数的那个版本。继承和动态绑定的结合使得我们能够编写具有特定类型行为但又独立于类型的程序。

在 C++ 语言中，动态绑定只作用于虚函数，并且需要通过指针或引用调用。

在派生类对象中包含有与它的每个基类对应的子对象。因为所有派生类对象都含有基类部分，所以我们能将派生类的引用或指针转换为一个可访问的基类引用或指针。

当执行派生类的构造、拷贝、移动和赋值操作时，首先构造、拷贝、移动和赋值其中的基类部分，然后才轮到派生类部分。析构函数的执行顺序则正好相反，首先销毁派生类，接下来执行基类子对象的析构函数。基类通常都应该定义一个虚析构函数，即使基类根本不需要析构函数也最好这么做。将基类的析构函数定义成虚函数的原因是为了确保当我们删除一个基类指针，而该指针实际指向一个派生类对象时，程序也能正确运行。

派生类为它的每个基类提供一个保护级别。`public` 基类的成员也是派生类接口的一部分；`private` 基类的成员是不可访问的；`protected` 基类的成员对于派生类的派生类是可访问的，但是对于派生类的用户不可访问。

## 术语表

**抽象基类（abstract base class）** 含有一个或多个纯虚函数的类，我们无法创建抽象基类的对象。

**可访问的（accessible）** 能被派生类对象访问的基类成员。可访问性由派生类的派生列表中所用的访问说明符和基类中成员的访问级别共同决定。例如，通过公有继承而来的一个公有成员对于派生类的用户来说是可访问的；而私有继承而来的公有成员是不可访问的。

**基类（base class）** 可供其他类继承的类。基类的成员也将成为派生类的成员。

**类派生列表（class derivation list）** 罗列了所有基类，每个基类包含一个可选的访问级别，它定义了派生类继承该基类的方式。如果没有提供访问说明符，则当派生类通过关键字 `struct` 定义时继承是公有的；而当派生类通过关键字 `class` 定义时继承是私有的。

**派生类（derived class）** 从其他类派生而

来的类。派生类可以覆盖其基类的虚函数，也可以定义自己的新成员。派生类的作用域嵌套在基类作用域当中；派生类的成员能直接访问基类的成员。

**派生类向基类的类型转换（derived-to-base conversion）** 派生类对象向基类引用或者派生类指针向基类指针的隐式类型转换。

**直接基类（direct base class）** 派生类直接继承的基类，直接基类在派生类的派生列表中说明。直接基类本身也可以是一个派生类。

**动态绑定（dynamic binding）** 直到运行时才确定到底执行函数的哪个版本。在 C++ 语言中，动态绑定的意思是在运行时根据引用或指针所绑定对象的实际类型来选择执行虚函数的某一个版本。

**动态类型（dynamic type）** 对象在运行时的类型。引用所引对象或者指针所指对象的动态类型可能与该引用或指针的静态类型不同。基类的指针或引用可以指向一个

&lt; 650

派生类对象。在这样的情况中，静态类型是基类的引用（或指针），而动态类型是派生类的引用（或指针）。

**间接基类 (indirect base class)** 不出现在派生类的派生列表中的基类。直接基类以直接或间接方式继承的类是派生类的间接基类。

**继承 (inheritance)** 由一个已有的类（基类）定义一个新类（派生类）的编程技术。派生类将继承基类的成员。

**面向对象编程 (object-oriented programming)** 利用数据抽象、继承以及动态绑定等技术编写程序的方法。

**覆盖 (override)** 派生类中定义的虚函数如果与基类中定义的同名虚函数有相同的形参列表，则派生类版本将覆盖基类的版本。

**多态性 (polymorphism)** 当用于面向对象编程的范畴时，多态性的含义是指程序能通过引用或指针的动态类型获取类型特定行为的能力。

**私有继承 (private inheritance)** 在私有继承中，基类的公有成员和受保护成员是派生类的私有成员。

**protected 访问说明符 (protected access specifier)** `protected` 关键字之后定义的成员能被派生类的成员和友元访问。但是这些成员只对派生类对象是可访问的，对类的普通用户则是不可访问的。

**受保护的继承 (protected inheritance)** 在受保护的继承中，基类的公有成员和受保护成员是派生类的受保护成员。

**公有继承 (public inheritance)** 基类的公有接口是派生类公有接口的组成部分。

**纯虚函数 (pure virtual)** 在类的内部声明虚函数时，在分号之前使用了`=0`。一个纯虚函数不需要（但是可以）被定义。含有纯虚函数的类是抽象基类。如果派生类没有对继承而来的纯虚函数定义自己的版本，则该派生类也是抽象的。

**重构 (refactoring)** 重新设计程序以便将一些相关的部分搜集到一个单独的抽象中，然后使用新的抽象替换原来的代码。通常情况下，重构类的方式是将数据成员和函数成员移动到继承体系的高级别节点当中，从而避免代码冗余。

**运行时绑定 (run-time binding)** 参见“动态绑定”。

**切掉 (sliced down)** 当我们用一个派生类对象初始化基类对象或者为基类对象赋值时发生的情况。对象的派生类部分将被“切掉”，只剩下基类部分赋值给基类对象。

**静态类型 (static type)** 对象被定义的类型或表达式产生的类型。静态类型在编译时是已知的。

**虚函数 (virtual function)** 用于定义类型特定行为的成员函数。通过引用或指针对虚函数的调用直到运行时才被解析，依据是引用或指针所绑定对象的类型。