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。 C++ 11

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。 <624

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： C++ 11

- 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();
```

```
    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。

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。

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
```

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
628 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。

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。

C++11

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：

629

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

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_prt<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 631
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。

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），所以我们能把 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

635 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。

637

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：

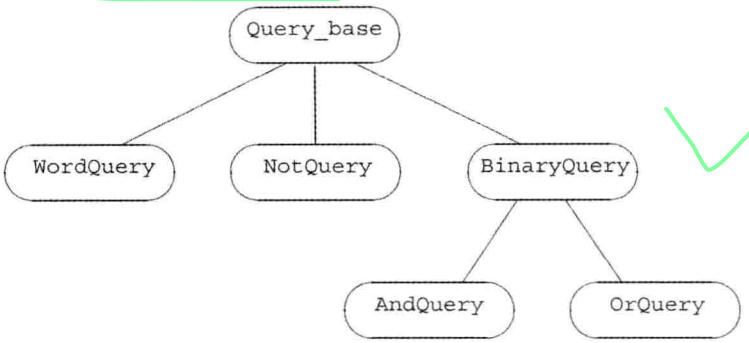

图 15.2：Query_base 继承体系

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 638
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。

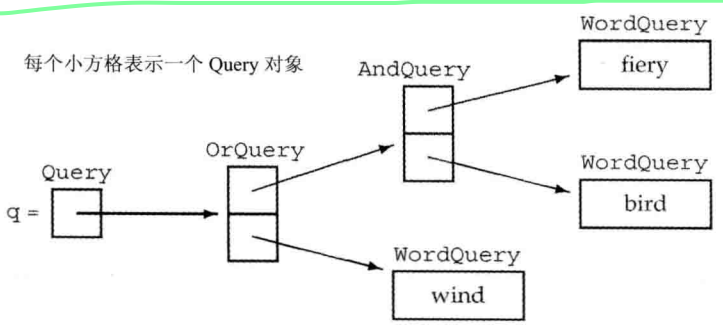

图 15.3：使用 Query 表达式创建的对象

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_prt 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，原因是我们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(); }
```

641

```
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：

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

```
// 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 set_intersection 在附录 A.2.8（第 779 页）中有详细的描述。 647

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。

done!

done!

小结

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哪个版本。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。 650

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

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。

done!

第 16 章
模板与泛型编程

内容

面向对象编程（OOP）和泛型编程都能处理在编写程序时不知道类型的情况。不同之处在于：OOP 能处理类型在程序运行之前都未知的情况；而在泛型编程中，在编译时就能获知类型了。

本书第 II 部分中介绍的容器、迭代器和算法都是泛型编程的例子。当我们编写一个泛型程序时，是独立于任何特定类型来编写代码的。当使用一个泛型程序时，我们提供类型或值，程序实例可在其上运行。

例如，标准库为每个容器提供了单一的、泛型的定义，如 `vector`。我们可以使用这个泛型定义来定义很多类型的 `vector`，它们的差异就在于包含的元素类型不同。

模板是泛型编程的基础。我们不必了解模板是如何定义的就能使用它们，实际上我们已经这样用了。在本章中，我们将学习如何定义自己的模板。

652 模板是 C++中泛型编程的基础。一个模板就是一个创建类或函数的蓝图或者说公式。当使用一个 vector 这样的泛型类型，或者 find 这样的泛型函数时，我们提供足够的信息，将蓝图转换为特定的类或函数。这种转换发生在编译时。在本书第 3 章和第 II 部分中我们已经学习了如何使用模板。在本章中，我们将学习如何定义模板。

16.1 定义模板

假定我们希望编写一个函数来比较两个值，并指出第一个值是小于、等于还是大于第二个值。在实际中，我们可能想要定义多个函数，每个函数比较一种给定类型的值。我们的初次尝试可能定义多个重载函数：

```
// 如果两个值相等，返回 0，如果 v1 小返回-1，如果 v2 小返回 1
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

这两个函数几乎是相同的，唯一的差异是参数的类型，函数体则完全一样。

如果对每种希望比较的类型都不得不重复定义完全一样的函数体，是非常烦琐且容易出错的。更麻烦的是，在编写程序的时候，我们就要确定可能要 compare 的所有类型。如果希望能在用户提供的类型上使用此函数，这种策略就失效了。

16.1.1 函数模板

我们可以定义一个通用的**函数模板**（function template），而不是为每个类型都定义一个新函数。一个函数模板就是一个公式，可用来生成针对特定类型的函数版本。compare 的模板版本可能像下面这样：

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

653 模板定义以关键字 template 开始，后跟一个**模板参数列表**（template parameter list），这是一个逗号分隔的一个或多个**模板参数**（template parameter）的列表，用小于号（<）和大于号（>）包围起来。

在模板定义中，模板参数列表不能为空。

模板参数列表的作用很像函数参数列表。函数参数列表定义了若干特定类型的局部变量，但并未指出如何初始化它们。在运行时，调用者提供实参来初始化形参。

类似的，模板参数表示在类或函数定义中用到的类型或值。当使用模板时，我们（隐式地或显式地）指定**模板实参**（template argument），将其绑定到模板参数上。

我们的 compare 函数声明了一个名为 T 的类型参数。在 compare 中，我们用名字 T 表示一个类型。而 T 表示的实际类型则在编译时根据 compare 的使用情况来确定。

实例化函数模板

当我们调用一个函数模板时，编译器（通常）用函数实参来为我们推断模板实参。即，当我们调用 compare 时，编译器使用实参的类型来确定绑定到模板参数 T 的类型。例如，在下面的调用中：

```
cout << compare(1, 0) << endl; // T 为 int
```

实参类型是 int。编译器会推断出模板实参为 int，并将它绑定到模板参数 T。

编译器用推断出的模板参数来为我们**实例化**（instantiate）一个特定版本的函数。当编译器实例化一个模板时，它使用实际的模板实参代替对应的模板参数来创建出模板的一个新“实例”。例如，给定下面的调用：

```
// 实例化出 int compare(const int&, const int&)
cout << compare(1, 0) << endl; // T 为 int
// 实例化出 int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T 为 vector<int>
```

编译器会实例化出两个不同版本的 compare。对于第一个调用，编译器会编写并编译一个 compare 版本，其中 T 被替换为 int：

```
int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

对于第二个调用，编译器会生成另一个 compare 版本，其中 T 被替换为 vector<int>。这些编译器生成的版本通常被称为模板的**实例**（instantiation）。

模板类型参数

654

我们的 compare 函数有一个模板**类型参数**（type parameter）。一般来说，我们可以将类型参数看作类型说明符，就像内置类型或类类型说明符一样使用。特别是，类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换：

```
// 正确：返回类型和参数类型相同
template <typename T> T foo(T* p)
{
    T tmp = *p; // tmp 的类型将是指针 p 指向的类型
    //...
    return tmp;
}
```

类型参数前必须使用关键字 class 或 typename：

```
// 错误：U 之前必须加上 class 或 typename
template <typename T, U> T calc(const T&, const U&);
```

在模板参数列表中，这两个关键字的含义相同，可以互换使用。一个模板参数列表中可以同时使用这两个关键字：

```
// 正确：在模板参数列表中，typename 和 class 没有什么不同
template <typename T, class U> calc (const T&, const U&);
```

看起来用关键字 typename 来指定模板类型参数比用 class 更为直观。毕竟，我们可以用内置（非类）类型作为模板类型实参。而且，typename 更清楚地指出随后的名字是一个类型名。但是，typename 是在模板已经广泛使用之后才引入 C++语言的，某些程序员仍然只用 class。

非类型模板参数

除了定义类型参数，还可以在模板中定义**非类型参数**（nontype parameter）。一个非类型参数表示一个值而非一个类型。我们通过一个特定的类型名而非关键字 class 或 typename 来指定非类型参数。

当一个模板被实例化时，非类型参数被一个用户提供的或编译器推断出的值所代替。这些值必须是常量表达式（参见 2.4.4 节，第 58 页），从而允许编译器在编译时实例化模板。

例如，我们可以编写一个 compare 版本处理字符串字面常量。这种字面常量是 const char 的数组。由于不能拷贝一个数组，所以我们将自己的参数定义为数组的引用（参见 6.2.4 节，第 195 页）。由于我们希望能比较不同长度的字符串字面常量，因此为模板定义了两个非类型的参数。第一个模板参数表示第一个数组的长度，第二个参数表示第二个数组的长度：

655

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    return strcmp(p1, p2);
}
```

当我们调用这个版本的 compare 时：

```
compare("hi", "mom")
```

编译器会使用字面常量的大小来代替 N 和 M，从而实例化模板。记住，编译器会在一个字符串字面常量的末尾插入一个空字符作为终结符（参见 2.1.3 节，第 36 页），因此编译器会实例化出如下版本：

```
int compare(const char (&p1)[3], const char (&p2)[4])
```

一个非类型参数可以是一个整型，或者是一个指向对象或函数类型的指针或（左值）引用。绑定到非类型整型参数的实参必须是一个常量表达式。绑定到指针或引用非类型参数的实参必须具有静态的生存期（参见第 12 章，第 400 页）。我们不能用一个普通（非 static）局部变量或动态对象作为指针或引用非类型模板参数的实参。指针参数也可以用 nullptr 或一个值为 0 的常量表达式来实例化。

在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用

非类型参数，例如，指定数组大小。

非类型模板参数的模板实参必须是常量表达式。

inline 和 constexpr 的函数模板

函数模板可以声明为 inline 或 constexpr 的，如同非模板函数一样。inline 或 constexpr 说明符放在模板参数列表之后，返回类型之前：

```
// 正确：inline 说明符跟在模板参数列表之后
template <typename T> inline T min(const T&, const T&);
// 错误：inline 说明符的位置不正确
inline template <typename T> T min(const T&, const T&);
```

编写类型无关的代码

我们最初的 compare 函数虽然简单，但它说明了编写泛型代码的两个重要原则：

- 模板中的函数参数是 const 的引用。
- 函数体中的条件判断仅使用<比较运业。

通过将函数参数设定为 const 的引用，我们保证了函数可以用于不能拷贝的类型。大多数类型，包括内置类型和我们已经用过的标准库类型（除 unique_ptr 和 IO 类型之外），都是允许拷贝的。但是，不允许拷贝的类类型也是存在的。通过将参数设定为 const 的引用，保证了这些类型可以用我们的 compare 函数来处理。而且，如果 compare 用于处理大对象，这种设计策略还能使函数运行得更快。

656

你可能认为既使用<运算符又使用>运算符来进行比较操作会更为自然：

```
// 期望的比较操作
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

但是，如果编写代码时只使用<运算符，我们就降低了 compare 函数对要处理的类型的要求。这些类型必须支持<，但不必同时支持>。

实际上，如果我们真的关心类型无关和可移植性，可能需要用 less（参见 14.8.2 节，第 510 页）来定义我们的函数：

```
// 即使用于指针也正确的 compare 版本；参见 14.8.2 节（第 510 页）
template <typename T> int compare(const T &v1, const T &v2)
{
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
```

原始版本存在的问题是，如果用户调用它比较两个指针，且两个指针未指向相同的数组，则代码的行为是未定义的（据查阅资料，less<T>的默认实现用的就是<，所以这其实并未起到让这种比较有一个良好定义的作用——译者注）。

模板程序应该尽量减少对实参类型的要求。

模板编译

当编译器遇到一个模板定义时，它并不生成代码。只有当我们实例化出模板的一个特定版本时，编译器才会生成代码。当我们使用（而不是定义）模板时，编译器才生成代码，这一特性影响了我们如何组织代码以及错误何时被检测到。

通常，当我们调用一个函数时，编译器只需要掌握函数的声明。类似的，当我们使用一个类类型的对象时，类定义必须是可用的，但成员函数的定义不必已经出现。因此，我们将类定义和函数声明放在头文件中，而普通函数和类的成员函数的定义放在源文件中。

模板则不同：为了生成一个实例化版本，编译器需要掌握函数模板或类模板成员函数的定义。因此，与非模板代码不同，模板的头文件通常既包括声明也包括定义。

657

函数模板和类模板成员函数的定义通常放在头文件中。

关键概念：模板和头文件

模板包含两种名字：

- 那些不依赖于模板参数的名字
- 那些依赖于模板参数的名字

当使用模板时，所有不依赖于模板参数的名字都必须是可见的，这是由模板的提供者来保证的。而且，模板的提供者必须保证，当模板被实例化时，模板的定义，包括类模板的成员的定义，也必须是可见的。

用来实例化模板的所有函数、类型以及与类型关联的运算符的声明都必须是可见的，这是由模板的用户来保证的。

通过组织良好的程序结构，恰当使用头文件，这些要求都很容易满足。模板的设计者应该提供一个头文件，包含模板定义以及在类模板或成员定义中用到的所有名字的声明。模板的用户必须包含模板的头文件，以及用来实例化模板的任何类型的头文件。

大多数编译错误在实例化期间报告

模板直到实例化时才会生成代码，这一特性影响了我们何时才会获知模板内代码的编译错误。通常，编译器会在三个阶段报告错误。

第一个阶段是编译模板本身时。在这个阶段，编译器通常不会发现很多错误。编译器可以检查语法错误，例如忘记分号或者变量名拼错等，但也就这么多了。

第二个阶段是编译器遇到模板使用时。在此阶段，编译器仍然没有很多可检查的。对于函数模板调用，编译器通常会检查实参数目是否正确。它还能检查参数类型是否匹配。对于类模板，编译器可以检查用户是否提供了正确数目的模板实参，但也仅限于此了。

第三个阶段是模板实例化时，只有这个阶段才能发现类型相关的错误。依赖于编译器如何管理实例化，这类错误可能在链接时才报告。

当我们编写模板时，代码不能是针对特定类型的，但模板代码通常对其所使用的类型有一些假设。例如，我们最初的 compare 函数中的代码就假定实参类型定义了<运算符。

```
if (v1 < v2) return -1;   // 要求类型 T 的对象支持<操作
if (v2 < v1) return 1;    // 要求类型 T 的对象支持<操作
```

```
    return 0;                    // 返回 int；不依赖于 T
```

当编译器处理此模板时，它不能验证 if 语句中的条件是否合法。如果传递给 compare 的实参定义了<运算符，则代码就是正确的，否则就是错误的。例如，

658

```
Sales_data data1, data2;
cout << compare(data1, data2) << endl; // 错误：Sales_data 未定义<
```

此调用实例化了 compare 的一个版本，将 T 替换为 Sales_data。if 条件试图对 Sales_data 对象使用<运算符，但 Sales_data 并未定义此运算符。此实例化生成了一个无法编译通过的函数版本。但是，这样的错误直至编译器在类型 Sales_data 上实例化 compare 时才会被发现。

保证传递给模板的实参支持模板所要求的操作，以及这些操作在模板中能正确工作，是调用者的责任。

16.1.1 节练习

练习 16.1：给出实例化的定义。

练习 16.2：编写并测试你自己版本的 compare 函数。

练习 16.3：对两个 Sales_data 对象调用你的 compare 函数，观察编译器在实例化过程中如何处理错误。

练习 16.4：编写行为类似标准库 find 算法的模板。函数需要两个模板类型参数，一个表示函数的迭代器参数，另一个表示值的类型。使用你的函数在一个 vector<int>和一个 list<string>中查找给定值。

练习 16.5：为 6.2.4 节（第 195 页）中的 print 函数编写模板版本，它接受一个数组的引用，能处理任意大小、任意元素类型的数组。

练习 16.6：你认为接受一个数组实参的标准库函数 begin 和 end 是如何工作的？定义你自己版本的 begin 和 end。

练习 16.7：编写一个 constexpr 模板，返回给定数组的大小。

练习 16.8：在第 97 页的“关键概念”中，我们注意到，C++程序员喜欢使用!=而不喜欢<。解释这个习惯的原因。

16.1.2 类模板

类模板（class template）是用来生成类的蓝图的。与函数模板的不同之处是，编译器不能为类模板推断模板参数类型。如我们已经多次看到的，为了使用类模板，我们必须在模板名后的尖括号中提供额外信息（参见 3.3 节，第 87 页）——用来代替模板参数的模板实参列表。

659

定义类模板

作为一个例子，我们将实现 StrBlob（参见 12.1.1 节，第 405 页）的模板版本。我们将此模板命名为 Blob，意指它不再针对 string。类似 StrBlob，我们的模板会提供对元素的共享（且核查过的）访问能力。与类不同，我们的模板可以用于更多类型的元素。与标准库容器相同，当使用 Blob 时，用户需要指出元素类型。

类似函数模板，类模板以关键字 template 开始，后跟模板参数列表。在类模板（及其成员）的定义中，我们将模板参数当作替身，代替使用模板时用户需要提供的类型或值：

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // 构造函数
    Blob();
    Blob(std::initializer_list<T> il);
    // Blob 中的元素数目
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const T &t) {data->push_back(t);}
    // 移动版本，参见 13.6.3 节（第 484 页）
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    // 元素访问
    T& back();
    T& operator[](size_type i); // 在 14.5 节（第 501 页）中定义
private:
    std::shared_ptr<std::vector<T>> data;
    // 若 data[i]无效，则抛出 msg
    void check(size_type i, const std::string &msg) const;
};
```

我们的 Blob 模板有一个名为 T 的模板类型参数，用来表示 Blob 保存的元素的类型。例如，我们将元素访问操作的返回类型定义为 T&。当用户实例化 Blob 时，T 就会被替换为特定的模板实参类型。

除了模板参数列表和使用 T 代替 string 之外，此类模板的定义与 12.1.1 节（第 405 页）中定义的类版本及 12.1.6 节（第 422 页）和第 13 章、第 14 章中更新的版本是一样的。

660

实例化类模板

我们已经多次见到，当使用一个类模板时，我们必须提供额外信息。我们现在知道这些额外信息是**显式模板实参**（explicit template argument）列表，它们被绑定到模板参数。编译器使用这些模板实参来实例化出特定的类。

例如，为了用我们的 Blob 模板定义一个类型，必须提供元素类型：

```
Blob<int> ia;                  // 空 Blob<int>
Blob<int> ia2 = {0,1,2,3,4}; // 有 5 个元素的 Blob<int>
```

ia 和 ia2 使用相同的特定类型版本的 Blob（即 Blob<int>）。从这两个定义，编译器会实例化出一个与下面定义等价的类：

```
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    //...
    int& operator[](size_type i);
```

```
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg) const;
};
```

当编译器从我们的 Blob 模板实例化出一个类时，它会重写 Blob 模板，将模板参数 T 的每个实例替换为给定的模板实参，在本例中是 int。

对我们指定的每一种元素类型，编译器都生成一个不同的类：

```
// 下面的定义实例化出两个不同的 Blob 类型
Blob<string> names; // 保存 string 的 Blob
Blob<double> prices;// 不同的元素类型
```

这两个定义会实例化出两个不同的类。names 的定义创建了一个 Blob 类，每个 T 都被替换为 string。prices 的定义生成了另一个 Blob 类，T 被替换为 double。

一个类模板的每个实例都形成一个独立的类。类型 Blob<string>与任何其他 Blob 类型都没有关联，也不会对任何其他 Blob 类型的成员有特殊访问权限。

在模板作用域中引用模板类型

为了阅读模板类代码，应该记住类模板的名字不是一个类型名（参见 3.3 节，第 87 页）。类模板用来实例化类型，而一个实例化的类型总是包含模板参数的。

可能令人迷惑的是，一个类模板中的代码如果使用了另外一个模板，通常不将一个实际类型（或值）的名字用作其模板实参。相反的，我们通常将模板自己的参数当作被使用模板的实参。例如，我们的 data 成员使用了两个模板，vector 和 shared_ptr。我们知道，无论何时使用模板都必须提供模板实参。在本例中，我们提供的模板实参就是 Blob 的模板参数。因此，data 的定义如下：

661

```
std::shared_ptr<std::vector<T>> data;
```

它使用了 Blob 的类型参数来声明 data 是一个 shared_ptr 的实例，此 shared_ptr 指向一个保存类型为 T 的对象的 vector 实例。当我们实例化一个特定类型的 Blob，例如 Blob<string>时，data 会成为：

```
shared_ptr<vector<string>>
```

如果我们实例化 Blob<int>，则 data 会成为 shared_ptr<vector<int>>，依此类推。

类模板的成员函数

与其他任何类相同，我们既可以在类模板内部，也可以在类模板外部为其定义成员函数，且定义在类模板内的成员函数被隐式声明为内联函数。

类模板的成员函数本身是一个普通函数。但是，类模板的每个实例都有其自己版本的成员函数。因此，类模板的成员函数具有和模板相同的模板参数。因而，定义在类模板之外的成员函数就必须以关键字 template 开始，后接类模板参数列表。

与往常一样，当我们在类外定义一个成员时，必须说明成员属于哪个类。而且，从一个模板生成的类的名字中必须包含其模板实参。当我们定义一个成员函数时，模板实参与模板形参相同。即，对于 StrBlob 的一个给定的成员函数

```
ret-type StrBlob::member-name(parm-list)
```

对应的 Blob 的成员应该是这样的：

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

check 和元素访问成员

我们首先定义 check 成员，它检查一个给定的索引：

```
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

除了类名中的不同之处以及使用了模板参数列表外，此函数与原 StrBlob 类的 check 成员完全一样。

下标运算符和 back 函数用模板参数指出返回类型，其他未变：

662

```
template <typename T>
T& Blob<T>::back()
{
    check(0, "back on empty Blob");
    return data->back();
}
template <typename T>
T& Blob<T>::operator[](size_type i)
{
    // 如果 i 太大，check 会抛出异常，阻止访问一个不存在的元素
    check(i, "subscript out of range");
    return (*data)[i];
}
```

在原 StrBlob 类中，这些运算符返回 string&。而模板版本则返回一个引用，指向用来实例化 Blob 的类型。

pop_back 函数与原 StrBlob 的成员几乎相同：

```
template <typename T> void Blob<T>::pop_back()
{
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
```

在原 StrBlob 类中，下标运算符和 back 成员都对 const 对象进行了重载。我们将这些成员及 front 成员的定义留作练习。

Blob 构造函数

与其他任何定义在类模板外的成员一样，构造函数的定义要以模板参数开始：

```
template <typename T>
Blob<T>::Blob(): data(std::make_shared<std::vector<T>>()) { }
```

这段代码在作用域 Blob<T>中定义了名为 Blob 的成员函数。类似 StrBlob 的默认构造