```
            cap = rhs.cap;
            // 将 rhs 置于可析构状态
            rhs.elements = rhs.first_free = rhs.cap = nullptr;
        }
        return *this;
    }
```

在此例中，我们直接检查 this 指针与 rhs 的地址是否相同。如果相同，右侧和左侧运算对象指向相同的对象，我们不需要做任何事情。否则，我们释放左侧运算对象所使用的内存，并接管给定对象的内存。与移动构造函数一样，我们将 rhs 中的指针置为 nullptr。

537

我们费心地去检查自赋值情况看起来有些奇怪。毕竟，移动赋值运算符需要右侧运算对象的一个右值。我们进行检查的原因是此右值可能是 move 调用的返回结果。与其他任何赋值运算符一样，关键点是我们不能在使用右侧运算对象的资源之前就释放左侧运算对象的资源（可能是相同的资源）。

移后源对象必须可析构



从一个对象移动数据并不会销毁此对象，但有时在移动操作完成后，源对象会被销毁。因此，当我们编写一个移动操作时，必须确保移后源对象进入一个可析构的状态。我们的 StrVec 的移动操作满足这一要求，这是通过将移后源对象的指针成员置为 nullptr 来实现的。

除了将移后源对象置为析构安全的状态之外，移动操作还必须保证对象仍然是有效的。一般来说，对象有效就是指可以安全地为其赋予新值或者可以安全地使用而不依赖其当前值。另一方面，移动操作对移后源对象中留下的值没有任何要求。因此，我们的程序不应该依赖于移后源对象中的数据。

例如，当我们从一个标准库 string 或容器对象移动数据时，我们知道移后源对象仍然保持有效。因此，我们可以对它执行诸如 empty 或 size 这些操作。但是，我们不知道将会得到什么结果。我们可能期望一个移后源对象是空的，但这并没有保证。

我们的 StrVec 类的移动操作将移后源对象置于与默认初始化的对象相同的状态。因此，我们可以继续对移后源对象执行所有的 StrVec 操作，与任何其他默认初始化的对象一样。而其他内部结构更为复杂的类，可能表现出完全不同的行为。



在移动操作之后，移后源对象必须保持有效的、可析构的状态，但是用户不能对其值进行任何假设。

合成的移动操作

与处理拷贝构造函数和拷贝赋值运算符一样，编译器也会合成移动构造函数和移动赋值运算符。但是，合成移动操作的条件与合成拷贝操作的条件大不相同。

回忆一下，如果我们不声明自己的拷贝构造函数或拷贝赋值运算符，编译器总会为我们合成这些操作（参见 13.1.1 节，第 440 页和 13.1.2 节，第 444 页）。拷贝操作要么被定义为逐成员拷贝，要么被定义为对象赋值，要么被定义为删除的函数。

与拷贝操作不同，编译器根本不会为某些类合成移动操作。特别是，如果一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。因此，某些类就没有移动构造函数或移动赋值运算符。如我们将在第 477 页所见，如果一个类没有移动操作，通过正常的函数匹配，类会使用对应的拷

538

贝操作来代替移动操作。

只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非 static 数据成员都可以移动时，编译器才会为它合成移动构造函数或移动赋值运算符。编译器可以移动内置类型的成员。如果一个成员是类类型，且该类有对应的移动操作，编译器也能移动这个成员：

```
// 编译器会为 X 和 hasX 合成移动操作
struct X {
    int i;            // 内置类型可以移动
    std::string s;    // string 定义了自己的移动操作
};
struct hasX {
    X mem; // X 有合成的移动操作
};
X x, x2 = std::move(x);          // 使用合成的移动构造函数
hasX hx, hx2 = std::move(hx);    // 使用合成的移动构造函数
```



只有当一个类没有定义任何自己版本的拷贝控制成员，且它的所有数据成员都能移动构造或移动赋值时，编译器才会为它合成移动构造函数或移动赋值运算符。

与拷贝操作不同，移动操作永远不会隐式定义为删除的函数。但是，如果我们显式地要求编译器生成=default 的（参见 7.1.4 节，第 237 页）移动操作，且编译器不能移动所有成员，则编译器会将移动操作定义为删除的函数。除了一个重要例外，什么时候将合成的移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类似的原则（参见 13.1.6 节，第 449 页）：

- 与拷贝构造函数不同，移动构造函数被定义为删除的函数的条件是：有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者是有类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数。移动赋值运算符的情况类似。
- 如果有类成员的移动构造函数或移动赋值运算符被定义为删除的或是不可访问的，则类的移动构造函数或移动赋值运算符被定义为删除的。
- 类似拷贝构造函数，如果类的析构函数被定义为删除的或不可访问的，则类的移动构造函数被定义为删除的。
- 类似拷贝赋值运算符，如果有类成员是 const 的或是引用，则类的移动赋值运算符被定义为删除的。

539 例如，假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数：

```
// 假定 Y 是一个类，它定义了自己的拷贝构造函数但未定义自己的移动构造函数
struct hasY {
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; // hasY 将有一个删除的移动构造函数
};
hasY hy, hy2 = std::move(hy); // 错误：移动构造函数是删除的
```

编译器可以拷贝类型为 Y 的对象，但不能移动它们。类 hasY 显式地要求一个移动构造函数，但编译器无法为其生成。因此，hasY 会有一个删除的移动构造函数。如果 hasY 忽略了移动构造函数的声明，则编译器根本不能为它合成一个。如果移动操作可能被定义为

删除的函数，编译器就不会合成它们。

移动操作和合成的拷贝控制成员间还有最后一个相互作用关系：一个类是否定义了自己的移动操作对拷贝操作如何合成有影响。如果类定义了一个移动构造函数和/或一个移动赋值运算符，则该类的合成拷贝构造函数和拷贝赋值运算符会被定义为删除的。



定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作。否则，这些成员默认地被定义为删除的。

移动右值，拷贝左值……

如果一个类既有移动构造函数，也有拷贝构造函数，编译器使用普通的函数匹配规则来确定使用哪个构造函数（参见 6.4 节，第 208 页）。赋值操作的情况类似。例如，在我们的 StrVec 类中，拷贝构造函数接受一个 const StrVec 的引用。因此，它可以用于任何可以转换为 StrVec 的类型。而移动构造函数接受一个 StrVec&&，因此只能用于实参是（非 static）右值的情形：

```
StrVec v1, v2;
v1 = v2;                          // v2 是左值；使用拷贝赋值
StrVec getVec(istream &);         // getVec 返回一个右值
v2 = getVec(cin);                 // getVec(cin)是一个右值；使用移动赋值
```

在第一个赋值中，我们将 v2 传递给赋值运算符。v2 的类型是 StrVec，表达式 v2 是一个左值。因此移动版本的赋值运算符是不可行的（参见 6.6 节，第 217 页），因为我们不能隐式地将一个右值引用绑定到一个左值。因此，这个赋值语句使用拷贝赋值运算符。

在第二个赋值中，我们赋予 v2 的是 getVec 调用的结果。此表达式是一个右值。在此情况下，两个赋值运算符都是可行的——将 getVec 的结果绑定到两个运算符的参数都是允许的。调用拷贝赋值运算符需要进行一次到 const 的转换，而 StrVec&&则是精确匹配。因此，第二个赋值会使用移动赋值运算符。

……但如果没有移动构造函数，右值也被拷贝

540

如果一个类有一个拷贝构造函数但未定义移动构造函数，会发生什么呢？在此情况下，编译器不会合成移动构造函数，这意味着此类将有拷贝构造函数但不会有移动构造函数。如果一个类没有移动构造函数，函数匹配规则保证该类型的对象会被拷贝，即使我们试图通过调用 move 来移动它们时也是如此：

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&); // 拷贝构造函数
    // 其他成员定义，但 Foo 未定义移动构造函数
};
Foo x;
Foo y(x);              // 拷贝构造函数；x 是一个左值
Foo z(std::move(x)); // 拷贝构造函数，因为未定义移动构造函数
```

在对 z 进行初始化时，我们调用了 move(x)，它返回一个绑定到 x 的 Foo&&。Foo 的拷贝构造函数是可行的，因为我们可以将一个 Foo&&转换为一个 const Foo&。因此，z 的初始化将使用 Foo 的拷贝构造函数。

值得注意的是，用拷贝构造函数代替移动构造函数几乎肯定是安全的（赋值运算符的

情况类似)。一般情况下，拷贝构造函数满足对应的移动构造函数的要求：它会拷贝给定对象，并将原对象置于有效状态。实际上，拷贝构造函数甚至都不会改变原对象的值。



如果一个类有一个可用的拷贝构造函数而没有移动构造函数，则其对象是通过拷贝构造函数来“移动”的。拷贝赋值运算符和移动赋值运算符的情况类似。

拷贝并交换赋值运算符和移动操作

我们的 HasPtr 版本定义了一个拷贝并交换赋值运算符（参见 13.3 节，第 459 页），它是函数匹配和移动操作间相互关系的一个很好的示例。如果我们为此类添加一个移动构造函数，它实际上也会获得一个移动赋值运算符：

```
class HasPtr {
public:
    // 添加的移动构造函数
    HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i) {p.ps = 0;}
    // 赋值运算符既是移动赋值运算符，也是拷贝赋值运算符
    HasPtr& operator=(HasPtr rhs)
                    { swap(*this, rhs); return *this; }
    // 其他成员的定义，同 13.2.1 节（第 453 页）
};
```

在这个版本中，我们为类添加了一个移动构造函数，它接管了给定实参的值。构造函数体将给定的 HasPtr 的指针置为 0，从而确保销毁移后源对象是安全的。此函数不会抛出异常，因此我们将其标记为 noexcept（参见 13.6.2 节，第 473 页）。

541

现在让我们观察赋值运算符。此运算符有一个非引用参数，这意味着此参数要进行拷贝初始化（参见 13.1.1 节，第 441 页）。依赖于实参的类型，拷贝初始化要么使用拷贝构造函数，要么使用移动构造函数——左值被拷贝，右值被移动。因此，单一的赋值运算符就实现了拷贝赋值运算符和移动赋值运算符两种功能。

例如，假定 hp 和 hp2 都是 HasPtr 对象：

```
hp = hp2; // hp2 是一个左值；hp2 通过拷贝构造函数来拷贝
hp = std::move(hp2); // 移动构造函数移动 hp2
```

在第一个赋值中，右侧运算对象是一个左值，因此移动构造函数是不可行的。rhs 将使用拷贝构造函数来初始化。拷贝构造函数将分配一个新 string，并拷贝 hp2 指向的 string。

在第二个赋值中，我们调用 std:: move 将一个右值引用绑定到 hp2 上。在此情况下，拷贝构造函数和移动构造函数都是可行的。但是，由于实参是一个右值引用，移动构造函数是精确匹配的。移动构造函数从 hp2 拷贝指针，而不会分配任何内存。

不管使用的是拷贝构造函数还是移动构造函数，赋值运算符的函数体都 swap 两个运算对象的状态。交换 HasPtr 会交换两个对象的指针（及 int）成员。在 swap 之后，rhs 中的指针将指向原来左侧运算对象所拥有的 string。当 rhs 离开其作用域时，这个 string 将被销毁。

建议：更新三/五法则

所有五个拷贝控制成员应该看作一个整体：一般来说，如果一个类定义了任何一个

拷贝操作，它就应该定义所有五个操作。如前所述，某些类必须定义拷贝构造函数、拷贝赋值运算符和析构函数才能正确工作（参见 13.1.4 节，第 447 页）。这些类通常拥有一个资源，而拷贝成员必须拷贝此资源。一般来说，拷贝一个资源会导致一些额外开销。在这种拷贝并非必要的情况下，定义了移动构造函数和移动赋值运算符的类就可以避免此问题。

Message 类的移动操作

定义了自己的拷贝构造函数和拷贝赋值运算符的类通常也会从移动操作受益。例如，我们的 Message 和 Folder 类（参见 13.4 节，第 460 页）就应该定义移动操作。通过定义移动操作，Message 类可以使用 string 和 set 的移动操作来避免拷贝 contents 和 folders 成员的额外开销。

但是，除了移动 folders 成员，我们还必须更新每个指向原 Message 的 Folder。我们必须删除指向旧 Message 的指针，并添加一个指向新 Message 的指针。

移动构造函数和移动赋值运算符都需要更新 Folder 指针，因此我们首先定义一个操作来完成这一共同的工作：

542

```
// 从本 Message 移动 Folder 指针
void Message::move_Folders(Message *m)
{
    folders = std::move(m->folders); // 使用 set 的移动赋值运算符
    for (auto f : folders) { // 对每个 Folder
        f->remMsg(m);        // 从 Folder 中删除旧 Message
        f->addMsg(this);     // 将本 Message 添加到 Folder 中
    }
    m->folders.clear();      // 确保销毁 m 是无害的
}
```

此函数首先移动 folders 集合。通过调用 move，我们使用了 set 的移动赋值运算符而不是它的拷贝赋值运算符。如果我们忽略了 move 调用，代码仍能正常工作，但带来了不必要的拷贝。函数然后遍历所有 Folder，从其中删除指向原 Message 的指针并添加指向新 Message 的指针。

值得注意的是，向 set 插入一个元素可能会抛出一个异常——向容器添加元素的操作要求分配内存，意味着可能会抛出一个 bad_alloc 异常（参见 12.1.2 节，第 409 页）。因此，与我们的 HasPtr 和 StrVec 类的移动操作不同，Message 的移动构造函数和移动赋值运算符可能会抛出异常。因此我们未将它们标记为 noexcept（参见 13.6.2 节，第 473 页）。

函数最后对 m.folders 调用 clear。在执行了 move 之后，我们知道 m.folders 是有效的，但不知道它包含什么内容。由于 Message 的析构函数遍历 folders，我们希望能确定 set 是空的。

Message 的移动构造函数调用 move 来移动 contents，并默认初始化自己的 folders 成员：

```
Message::Message(Message &&m): contents(std::move(m.contents))
{
    move_Folders(&m); // 移动 folders 并更新 Folder 指针
}
```

在构造函数体中，我们调用了 move_Folders 来删除指向 m 的指针并插入指向本 Message 的指针。

移动赋值运算符直接检查自赋值情况：

```
Message& Message::operator=(Message &&rhs)
{
    if (this != &rhs) {          // 直接检查自赋值情况
        remove_from_Folders();
        contents = std::move(rhs.contents); // 移动赋值运算符
        move_Folders(&rhs);  // 重置 Folders 指向本 Message
    }
    return *this;
}
```

543 与任何赋值运算符一样，移动赋值运算符必须销毁左侧运算对象的旧状态。在本例中，销毁左侧运算对象要求我们从现有 folders 中删除指向本 Message 的指针，我们调用 remove_from_Folders 来完成这一工作。完成删除工作后，我们调用 move 从 rhs 将 contents 移动到 this 对象。剩下的就是调用 move_Messages 来更新 Folder 指针了。

移动迭代器

StrVec 的 reallocate 成员（参见 13.5 节，第 469 页）使用了一个 for 循环来调用 construct 从旧内存将元素拷贝到新内存中。作为一种替换方法，如果我们能调用 uninitialized_copy 来构造新分配的内存，将比循环更为简单。但是，uninitialized_copy 恰如其名：它对元素进行拷贝操作。标准库中并没有类似的函数将对象“移动”到未构造的内存中。

C++ 11 新标准库中定义了一种**移动迭代器**（move iterator）适配器（参见 10.4 节，第 358 页）。一个移动迭代器通过改变给定迭代器的解引用运算符的行为来适配此迭代器。一般来说，一个迭代器的解引用运算符返回一个指向元素的左值。与其他迭代器不同，移动迭代器的解引用运算符生成一个右值引用。

我们通过调用标准库的 make_move_iterator 函数将一个普通迭代器转换为一个移动迭代器。此函数接受一个迭代器参数，返回一个移动迭代器。

原迭代器的所有其他操作在移动迭代器中都照常工作。由于移动迭代器支持正常的迭代器操作，我们可以将一对移动迭代器传递给算法。特别是，可以将移动迭代器传递给 uninitialized_copy：

```
void StrVec::reallocate()
{
    // 分配大小两倍于当前规模的内存空间
    auto newcapacity = size() ? 2 * size() : 1;
    auto first = alloc.allocate(newcapacity);
    // 移动元素
    auto last = uninitialized_copy(make_move_iterator(begin()),
                                   make_move_iterator(end()),
                                   first);
    free();                  // 释放旧空间
    elements = first;        // 更新指针
    first_free = last;
```

```
        cap = elements + newcapacity;
    }
```

uninitialized_copy 对输入序列中的每个元素调用 construct 来将元素“拷贝”到目的位置。此算法使用迭代器的解引用运算符从输入序列中提取元素。由于我们传递给它的是移动迭代器，因此解引用运算符生成的是一个右值引用，这意味着 construct 将使用移动构造函数来构造元素。

值得注意的是，标准库不保证哪些算法适用移动迭代器，哪些不适用。由于移动一个对象可能销毁掉原对象，因此你只有在确信算法在为一个元素赋值或将其传递给一个用户定义的函数后不再访问它时，才能将移动迭代器传递给算法。 544

建议：不要随意使用移动操作

由于一个移后源对象具有不确定的状态，对其调用 std::move 是危险的。当我们调用 move 时，必须绝对确认移后源对象没有其他用户。

通过在类代码中小心地使用 move，可以大幅度提升性能。而如果随意在普通用户代码（与类实现代码相对）中使用移动操作，很可能导致莫名其妙的、难以查找的错误，而难以提升应用程序性能。

Best Practices 在移动构造函数和移动赋值运算符这些类实现代码之外的地方，只有当你确信需要进行移动操作且移动操作是安全的，才可以使用 std:: move。

13.6.2 节练习

练习 13.49：为你的 StrVec、String 和 Message 类添加一个移动构造函数和一个移动赋值运算符。

练习 13.50：在你的 String 类的移动操作中添加打印语句，并重新运行 13.6.1 节（第 473 页）的练习 13.48 中的程序，它使用了一个 vector<String>，观察什么时候会避免拷贝。

练习 13.51：虽然 unique_ptr 不能拷贝，但我们在 12.1.5 节（第 418 页）中编写了一个 clone 函数，它以值方式返回一个 unique_ptr。解释为什么函数是合法的，以及为什么它能正确工作。

练习 13.52：详细解释第 478 页中的 HasPtr 对象的赋值发生了什么？特别是，一步一步描述 hp、hp2 以及 HasPtr 的赋值运算符中的参数 rhs 的值发生了什么变化。

练习 13.53：从底层效率的角度看，HasPtr 的赋值运算符并不理想，解释为什么。为 HasPtr 实现一个拷贝赋值运算符和一个移动赋值运算符，并比较你的新的移动赋值运算符中执行的操作和拷贝并交换版本中执行的操作。

练习 13.54：如果我们为 HasPtr 定义了移动赋值运算符，但未改变拷贝并交换运算符，会发生什么？编写代码验证你的答案。

13.6.3 右值引用和成员函数



除了构造函数和赋值运算符之外，如果一个成员函数同时提供拷贝和移动版本，它也能从中受益。这种允许移动的成员函数通常使用与拷贝/移动构造函数和赋值运算符相同的参数模式——一个版本接受一个指向 const 的左值引用，第二个版本接受一个指向非 545

const 的右值引用。

例如，定义了 push_back 的标准库容器提供两个版本：一个版本有一个右值引用参数，而另一个版本有一个 const 左值引用。假定 X 是元素类型，那么这些容器就会定义以下两个 push_back 版本：

```
void push_back(const X&);      // 拷贝：绑定到任意类型的 X
void push_back(X&&);           // 移动：只能绑定到类型 X 的可修改的右值
```

我们可以将能转换为类型 X 的任何对象传递给第一个版本的 push_back。此版本从其参数拷贝数据。对于第二个版本，我们只可以传递给它非 const 的右值。此版本对于非 const 的右值是精确匹配（也是更好的匹配）的，因此当我们传递一个可修改的右值（参见 13.6.2 节，第 477 页）时，编译器会选择运行这个版本。此版本会从其参数窃取数据。

一般来说，我们不需要为函数操作定义接受一个 const X&&或是一个（普通的）X& 参数的版本。当我们希望从实参“窃取”数据时，通常传递一个右值引用。为了达到这一目的，实参不能是 const 的。类似的，从一个对象进行拷贝的操作不应该改变该对象。因此，通常不需要定义一个接受一个（普通的）X&参数的版本。



区分移动和拷贝的重载函数通常有一个版本接受一个 const T&，而另一个版本接受一个 T&&。

作为一个更具体的例子，我们将为 StrVec 类定义另一个版本的 push_back：

```
class StrVec {
public:
    void push_back(const std::string&);   // 拷贝元素
    void push_back(std::string&&);        // 移动元素
    // 其他成员的定义，如前
};
// 与 13.5 节（第 466 页）中的原版本相同
void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // 确保有空间容纳新元素
    // 在 first_free 指向的元素中构造 s 的一个副本
    alloc.construct(first_free++, s);
}
void StrVec::push_back(string &&s)
{
    chk_n_alloc(); // 如果需要的话为 StrVec 重新分配内存
    alloc.construct(first_free++, std::move(s));
}
```

这两个成员几乎是相同的。差别在于右值引用版本调用 move 来将其参数传递给 construct。如前所述，construct 函数使用其第二个和随后的实参的类型来确定使用
546 哪个构造函数。由于 move 返回一个右值引用，传递给 construct 的实参类型是 string&&。因此，会使用 string 的移动构造函数来构造新元素。

当我们调用 push_back 时，实参类型决定了新元素是拷贝还是移动到容器中：

```
StrVec vec; // 空 StrVec
string s = "some string or another";
vec.push_back(s);          // 调用 push_back(const string&)
```

```
vec.push_back("done");    // 调用 push_back(string&&)
```

这些调用的差别在于实参是一个左值还是一个右值（从"done"创建的临时 string），具体调用哪个版本据此来决定。

右值和左值引用成员函数

通常，我们在一个对象上调用成员函数，而不管该对象是一个左值还是一个右值。例如：

```
string s1 = "a value", s2 = "another";
auto n = (s1 + s2).find('a');
```

此例中，我们在一个 string 右值上调用 find 成员（参见 9.5.3 节，第 325 页），该 string 右值是通过连接两个 string 而得到的。有时，右值的使用方式可能令人惊讶：

```
s1 + s2 = "wow!";
```

此处我们对两个 string 的连接结果——一个右值，进行了赋值。

在旧标准中，我们没有办法阻止这种使用方式。为了维持向后兼容性，新标准库类仍然允许向右值赋值。但是，我们可能希望在自己的类中阻止这种用法。在此情况下，我们希望强制左侧运算对象（即，this 指向的对象）是一个左值。

我们指出 this 的左值/右值属性的方式与定义 const 成员函数相同（参见 7.1.2 节，第 231 页），即，在参数列表后放置一个**引用限定符**（reference qualifier）：

C++ 11

```
class Foo {
public:
    Foo &operator=(const Foo&) &; // 只能向可修改的左值赋值
    // Foo 的其他参数
};
Foo &Foo::operator=(const Foo &rhs) &
{
    // 执行将 rhs 赋予本对象所需的工作
    return *this;
}
```

引用限定符可以是&或&&，分别指出 this 可以指向一个左值或右值。类似 const 限定符，引用限定符只能用于（非 static）成员函数，且必须同时出现在函数的声明和定义中。

对于&限定的函数，我们只能将它用于左值；对于&&限定的函数，只能用于右值：

```
Foo &retFoo();    // 返回一个引用；retFoo 调用是一个左值
Foo retVal();     // 返回一个值；retVal 调用是一个右值
Foo i, j;         // i 和 j 是左值
i = j;            // 正确：i 是左值
retFoo() = j;     // 正确：retFoo()返回一个左值
retVal() = j;     // 错误：retVal()返回一个右值
i = retVal();     // 正确：我们可以将一个右值作为赋值操作的右侧运算对象
```

547

一个函数可以同时用 const 和引用限定。在此情况下，引用限定符必须跟随在 const 限定符之后：

```
class Foo {
public:
Foo someMem() & const;        // 错误：const 限定符必须在前
Foo anotherMem() const &;     // 正确：const 限定符在前
};
```

重载和引用函数

就像一个成员函数可以根据是否有 const 来区分其重载版本一样（参见 7.3.2 节，第 247 页），引用限定符也可以区分重载版本。而且，我们可以综合引用限定符和 const 来区分一个成员函数的重载版本。例如，我们将为 Foo 定义一个名为 data 的 vector 成员和一个名为 sorted 的成员函数，sorted 返回一个 Foo 对象的副本，其中 vector 已被排序：

```
class Foo {
public:
    Foo sorted() &&;          // 可用于可改变的右值
    Foo sorted() const &;     // 可用于任何类型的 Foo
    // Foo 的其他成员的定义
private:
    vector<int> data;
};
// 本对象为右值，因此可以原址排序
Foo Foo::sorted() &&
{
    sort(data.begin(), data.end());
    return *this;
}
// 本对象是 const 或是一个左值，哪种情况我们都不能对其进行原址排序
Foo Foo::sorted() const & {
    Foo ret(*this);                              // 拷贝一个副本
    sort(ret.data.begin(), ret.data.end());      // 排序副本
    return ret;                                  // 返回副本
}
```

当我们对一个右值执行 sorted 时，它可以安全地直接对 data 成员进行排序。对象是一个右值，意味着没有其他用户，因此我们可以改变对象。当对一个 const 右值或一个左值执行 sorted 时，我们不能改变对象，因此就需要在排序前拷贝 data。

编译器会根据调用 sorted 的对象的左值/右值属性来确定使用哪个 sorted 版本：

548

```
retVal().sorted(); // retVal()是一个右值，调用 Foo::sorted() &&
retFoo().sorted(); // retFoo()是一个左值，调用 Foo::sorted() const &
```

当我们定义 const 成员函数时，可以定义两个版本，唯一的差别是一个版本有 const 限定而另一个没有。引用限定的函数则不一样。如果我们定义两个或两个以上具有相同名字和相同参数列表的成员函数，就必须对所有函数都加上引用限定符，或者所有都不加：

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // 错误：必须加上引用限定符
    // Comp 是函数类型的类型别名（参见 6.7 节，第 222 页）
    // 此函数类型可以用来比较 int 值
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);              // 正确：不同的参数列表
    Foo sorted(Comp*) const;        // 正确：两个版本都没有引用限定符
    };
```

本例中声明了一个没有参数的 const 版本的 sorted，此声明是错误的。因为 Foo 类中还有一个无参的 sorted 版本，它有一个引用限定符，因此 const 版本也必须有引用限定符。另一方面，接受一个比较操作指针的 sorted 版本是没问题的，因为两个函数都没有引用限定符。



如果一个成员函数有引用限定符，则具有相同参数列表的所有版本都必须有引用限定符。

13.6.3 节练习

练习 13.55：为你的 StrBlob 添加一个右值引用版本的 push_back。

练习 13.56：如果 sorted 定义如下，会发生什么：

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

练习 13.57：如果 sorted 定义如下，会发生什么：

```
Foo Foo::sorted() const & { return Foo(*this).sorted(); }
```

练习 13.58：编写新版本的 Foo 类，其 sorted 函数中有打印语句，测试这个类，来验证你对前两题的答案是否正确。

549 小结

每个类都会控制该类型对象拷贝、移动、赋值以及销毁时发生什么。特殊的成员函数——拷贝构造函数、移动构造函数、拷贝赋值运算符、移动赋值运算符和析构函数定义了这些操作。移动构造函数和移动赋值运算符接受一个（通常是非 const 的）右值引用；而拷贝版本则接受一个（通常是 const 的）普通左值引用。

如果一个类未声明这些操作，编译器会自动为其生成。如果这些操作未定义成删除的，它们会逐成员初始化、移动、赋值或销毁对象：合成的操作依次处理每个非 static 数据成员，根据成员类型确定如何移动、拷贝、赋值或销毁它。

分配了内存或其他资源的类几乎总是需要定义拷贝控制成员来管理分配的资源。如果一个类需要析构函数，则它几乎肯定也需要定义移动和拷贝构造函数及移动和拷贝赋值运算符。

术语表

拷贝并交换（copy and swap） 涉及赋值运算符的技术，首先拷贝右侧运算对象，然后调用 swap 来交换副本和左侧运算对象。

拷贝赋值运算符（copy-assignment operator） 接受一个本类型对象的赋值运算符版本。通常，拷贝赋值运算符的参数是一个 const 的引用，并返回指向本对象的引用。如果类未显式定义拷贝赋值运算符，编译器会为它合成一个。

拷贝构造函数（copy constructor） 一种构造函数，将新对象初始化为同类型另一个对象的副本。当向函数传递对象，或以传值方式从函数返回对象时，会隐式使用拷贝构造函数。如果我们未提供拷贝构造函数，编译器会为我们合成一个。

拷贝控制（copy control） 特殊的成员函数，控制拷贝、移动、赋值及销毁本类类型对象时发生什么。如果类未定义这些操作，编译器会为它合成恰当的定义。

拷贝初始化（copy initialization） 一种初始化形式，当我们使用=为一个新创建的对象提供初始化器时，会使用拷贝初始化。如果我们向函数传递对象或以传值方式从函数返回对象，以及初始化一个数组或一个聚合类时，也会使用拷贝初始化。

删除的函数（deleted function） 不能使用的函数。我们在一个函数的声明上指定=delete 来删除它。删除的函数的一个常见用途是告诉编译器不要为类合成拷贝和/或移动操作。

析构函数（destructor） 特殊的成员函数，当对象离开作用域或被释放时进行清理工作。编译器会自动销毁每个数据成员。类类型的成员通过调用其析构函数来销毁；而内置类型或复合类型的成员的销毁则不需要做任何工作。特别是，析构函数不会释放指针成员指向的对象。

左值引用（lvalue reference） 可以绑定到左值的引用。

550 **逐成员拷贝/赋值（memberwise copy/assign）** 合成的拷贝与移动构造函数及拷贝与移动赋值运算符的工作方式。合成的拷贝或移动构造函数依次处理每个非 static 数据成员，通过从给定对象拷贝或移动对应成员来初始化本对象成员；拷贝或移动赋值运算符从右侧运算对象中将每个成员拷贝赋值或移动赋值到左侧运算对象中。内置类型或复合类型的成员直接进行初始化或赋值。类类型的成员通过成员对应的拷贝/移动构造函数或拷贝/移动赋值运算符进行初始化或赋值。

move 用来将一个右值引用绑定到一个左值的标准库函数。调用 `move` 隐含地承诺我们将不会再使用移后源对象，除了销毁它或赋予它一个新值之外。

移动赋值运算符（move-assignment operator） 接受一个本类型右值引用参数的赋值运算符版本。通常，移动赋值运算符将数据从右侧运算对象移动到左侧运算对象。赋值之后，对右侧运算对象执行析构函数必须是安全的。

移动构造函数（move constructor） 一种构造函数，接受一个本类型的右值引用。通常，移动构造函数将数据从其参数移动到新创建的对象中。移动之后，对给定的实参执行析构函数必须是安全的。

移动迭代器（move iterator） 迭代器适配器，它生成的迭代器在解引用时会得到一个右值引用。

重载运算符（overloaded operator） 一种函数，重定义了运算符应用于类类型的对象时的含义。本章介绍了如何定义赋值运算符；第 14 章中将介绍重载运算符的更多细节内容。

引用计数（reference count） 一种程序设计技术，通常用于拷贝控制成员的设计。引用计数记录了有多少对象共享状态。构造函数（不是拷贝/移动构造函数）将引用计数置为 1。每当创建一个新副本时，计数值递增。当一个对象被销毁时，计数值递减。赋值运算符和析构函数检查递减的引用计数是否为 0，如果是，它们会销毁对象。

引用限定符（reference qualifier） 用来指出一个非 `static` 成员函数可以用于左值或右值的符号。限定符`&`和`&&`应该放在参数列表之后或 `const` 限定符之后（如果有的话）。被`&`限定的函数只能用于左值；被`&&`限定的函数只能用于右值。

右值引用（rvalue reference） 指向一个将要销毁的对象的引用。

合成赋值运算符（synthesized assignment operator） 编译器为未显式定义赋值运算符的类创建的（合成的）拷贝或移动赋值运算符版本。除非定义为删除的，合成赋值运算符会逐成员地将右侧运算对象赋予（移动到）左侧运算对象。

合成拷贝/移动构造函数（synthesized copy/move constructor） 编译器为未显式定义对应的构造函数的类生成的拷贝或移动构造函数版本。除非定义为删除的，合成拷贝或移动构造函数分别通过从给定对象拷贝或移动成员来逐成员地初始化新对象。

合成析构函数（synthesized destructor） 编译器为未显式定义析构函数的类创建的（合成的）版本。合成析构函数的函数体为空。

第 14 章
重载运算与类型转换

内容

14.1 基本概念 490
14.2 输入和输出运算符 494
14.3 算术和关系运算符 497
14.4 赋值运算符 499
14.5 下标运算符 501
14.6 递增和递减运算符 502
14.7 成员访问运算符 504
14.8 函数调用运算符 506
14.9 重载、类型转换与运算符 514
小结 523
术语表 523

在第 4 章中我们看到，C++语言定义了大量运算符以及内置类型的自动转换规则。这些特性使得程序员能编写出形式丰富、含有多种混合类型的表达式。

当运算符被用于类类型的对象时，C++语言允许我们为其指定新的含义；同时，我们也能自定义类类型之间的转换规则。和内置类型的转换一样，类类型转换隐式地将一种类型的对象转换成另一种我们所需类型的对象。

552 当运算符作用于类类型的运算对象时，可以通过运算符重载重新定义该运算符的含义。明智地使用运算符重载能令我们的程序更易于编写和阅读。举个例子，因为在 Sales_item 类（参见 1.5.1 节，第 17 页）中定义了输入、输出和加法运算符，所以可以通过下述形式输出两个 Sales_item 的和：

```
cout << item1 + item2;                    // 输出两个 Sales_item 的和
```

相反的，由于我们的 Sales_data 类（参见 7.1 节，第 228 页）还没有重载这些运算符，因此它的加法代码显得比较冗长而不清晰：

```
print(cout, add(data1, data2));   // 输出两个 Sales_data 的和
```

14.1 基本概念

重载的运算符是具有特殊名字的函数：它们的名字由关键字 operator 和其后要定义的运算符号共同组成。和其他函数一样，重载的运算符也包含返回类型、参数列表以及函数体。

重载运算符函数的参数数量与该运算符作用的运算对象数量一样多。一元运算符有一个参数，二元运算符有两个。对于二元运算符来说，左侧运算对象传递给第一个参数，而右侧运算对象传递给第二个参数。除了重载的函数调用运算符 operator() 之外，其他重载运算符不能含有默认实参（参见 6.5.1 节，第 211 页）。

如果一个运算符函数是成员函数，则它的第一个（左侧）运算对象绑定到隐式的 this 指针上（参见 7.1.2 节，第 231 页），因此，成员运算符函数的（显式）参数数量比运算符的运算对象总数少一个。



当一个重载的运算符是成员函数时，this 绑定到左侧运算对象。成员运算符函数的（显式）参数数量比运算对象的数量少一个。

对于一个运算符函数来说，它或者是类的成员，或者至少含有一个类类型的参数：

```
// 错误：不能为 int 重定义内置的运算符
int operator+(int, int);
```

这一约定意味着当运算符作用于内置类型的运算对象时，我们无法改变该运算符的含义。

我们可以重载大多数（但不是全部）运算符。表 14.1 指明了哪些运算符可以被重载，哪些不行。我们将在 19.1.1 节（第 726 页）介绍重载 new 和 delete 的方法。

我们只能重载已有的运算符，而无权发明新的运算符号。例如，我们不能提供 operator** 来执行幂操作。

有四个符号（+、-、*、&）既是一元运算符也是二元运算符，所有这些运算符都能被重载，从参数的数量我们可以推断到底定义的是哪种运算符。

553 对于一个重载的运算符来说，其优先级和结合律（参见 4.1.2 节，第 121 页）与对应的内置运算符保持一致。不考虑运算对象类型的话，

```
x == y + z;
```

永远等价于 x == (y + z)。

表 14.1：运算符

可以被重载的运算符					
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
不能被重载的运算符					
	::	.*	.	? :	

直接调用一个重载的运算符函数

通常情况下，我们将运算符作用于类型正确的实参，从而以这种间接方式“调用”重载的运算符函数。然而，我们也能像调用普通函数一样直接调用运算符函数，先指定函数名字，然后传入数量正确、类型适当的实参：

```
// 一个非成员运算符函数的等价调用
data1 + data2;                          // 普通的表达式
operator+(data1, data2);                // 等价的函数调用
```

这两次调用是等价的，它们都调用了非成员函数 operator+，传入 data1 作为第一个实参、传入 data2 作为第二个实参。

我们像调用其他成员函数一样显式地调用成员运算符函数。具体做法是，首先指定运行函数的对象（或指针）的名字，然后使用点运算符（或箭头运算符）访问希望调用的函数：

```
data1 += data2;                         // 基于“调用”的表达式
data1.operator+=(data2);                // 对成员运算符函数的等价调用
```

这两条语句都调用了成员函数 operator+=，将 this 绑定到 data1 的地址、将 data2 作为实参传入了函数。

某些运算符不应该被重载

回忆之前介绍过的，某些运算符指定了运算对象求值的顺序。因为使用重载的运算符本质上是一次函数调用，所以这些关于运算对象求值顺序的规则无法应用到重载的运算符上。特别是，逻辑与运算符、逻辑或运算符（参见 4.3 节，第 126 页）和逗号运算符（参见 4.10 节，第 140 页）的运算对象求值顺序规则无法保留下来。除此之外，&&和||运算符的重载版本也无法保留内置运算符的短路求值属性，两个运算对象总是会被求值。 554

因为上述运算符的重载版本无法保留求值顺序和/或短路求值属性，因此不建议重载它们。当代码使用了这些运算符的重载版本时，用户可能会突然发现他们一直习惯的求值规则不再适用了。

还有一个原因使得我们一般不重载逗号运算符和取地址运算符：C++语言已经定义了这两种运算符用于类类型对象时的特殊含义，这一点与大多数运算符都不相同。因为这两种运算符已经有了内置的含义，所以一般来说它们不应该被重载，否则它们的行为将异于常态，从而导致类的用户无法适应。

Best Practices 通常情况下，不应该重载逗号、取地址、逻辑与和逻辑或运算符。

使用与内置类型一致的含义

当你开始设计一个类时，首先应该考虑的是这个类将提供哪些操作。在确定类需要哪些操作之后，才能思考到底应该把每个类操作设成普通函数还是重载的运算符。如果某些操作在逻辑上与运算符相关，则它们适合于定义成重载的运算符：

- 如果类执行 IO 操作，则定义移位运算符使其与内置类型的 IO 保持一致。
- 如果类的某个操作是检查相等性，则定义 operator==；如果类有了 operator==，意味着它通常也应该有 operator!=。
- 如果类包含一个内在的单序比较操作，则定义 operator<；如果类有了 operator<，则它也应该含有其他关系操作。
- 重载运算符的返回类型通常情况下应该与其内置版本的返回类型兼容：逻辑运算符和关系运算符应该返回 bool，算术运算符应该返回一个类类型的值，赋值运算符和复合赋值运算符则应该返回左侧运算对象的一个引用。

提示：尽量明智地使用运算符重载

每个运算符在用于内置类型时都有比较明确的含义。以二元+运算符为例，它明显执行的是加法操作。因此，把二元+运算符映射到类类型的一个类似操作上可以极大地简化记忆。例如对于标准库类型 string 来说，我们就会使用+把一个 string 对象连接到另一个后面，很多编程语言都有类似的用法。

当在内置的运算符和我们自己的操作之间存在逻辑映射关系时，运算符重载的效果最好。此时，使用重载的运算符显然比另起一个名字更自然也更直观。不过，过分滥用运算符重载也会使我们的类变得难以理解。

在实际编程过程中，一般没有特别明显的滥用运算符重载的情况。例如，一般来说没有哪个程序员会定义 operator+并让它执行减法操作。然而经常发生的一种情况是，程序员可能会强行扭曲了运算符的“常规”含义使得其适应某种给定的类型，这显然是我们不希望发生的。因此我们的建议是：只有当操作的含义对于用户来说清晰明了时才使用运算符。如果用户对运算符可能有几种不同的理解，则使用这样的运算符将产生二义性。

赋值和复合赋值运算符

赋值运算符的行为与复合版本的类似：赋值之后，左侧运算对象和右侧运算对象的值相等，并且运算符应该返回它左侧运算对象的一个引用。重载的赋值运算应该继承而非违背其内置版本的含义。

555 如果类含有算术运算符（参见 4.2 节，第 124 页）或者位运算符（参见 4.8 节，第 136 页），则最好也提供对应的复合赋值运算符。无须赘言，+=运算符的行为显然应该与其内置版本一致，即先执行+，再执行=。

选择作为成员或者非成员

当我们定义重载的运算符时，必须首先决定是将其声明为类的成员函数还是声明为一个普通的非成员函数。在某些时候我们别无选择，因为有的运算符必须作为成员；另一些

情况下，运算符作为普通函数比作为成员更好。

下面的准则有助于我们在将运算符定义为成员函数还是普通的非成员函数做出抉择：

- 赋值（=）、下标（[ ]）、调用（( )）和成员访问箭头（->）运算符必须是成员。
- 复合赋值运算符一般来说应该是成员，但并非必须，这一点与赋值运算符略有不同。
- 改变对象状态的运算符或者与给定类型密切相关的运算符，如递增、递减和解引用运算符，通常应该是成员。
- 具有对称性的运算符可能转换任意一端的运算对象，例如算术、相等性、关系和位运算符等，因此它们通常应该是普通的非成员函数。

程序员希望能在含有混合类型的表达式中使用对称性运算符。例如，我们能求一个int和一个double的和，因为它们中的任意一个都可以是左侧运算对象或右侧运算对象，所以加法是对称的。如果我们想提供含有类对象的混合类型表达式，则运算符必须定义成非成员函数。 556

当我们把运算符定义成成员函数时，它的左侧运算对象必须是运算符所属类的一个对象。例如：

```
string s = "world";
string t = s + "!";  // 正确：我们能把一个 const char*加到一个 string 对象中
string u = "hi" + s; // 如果+是 string 的成员，则产生错误
```

如果operator+是string类的成员，则上面的第一个加法等价于s.operator+("!")。同样的，"hi"+s等价于"hi".operator+(s)。显然"hi"的类型是const char*，这是一种内置类型，根本就没有成员函数。

因为string将+定义成了普通的非成员函数，所以"hi"+s等价于operator+("hi",s)。和任何其他函数调用一样，每个实参都能被转换成形参类型。唯一的要求是至少有一个运算对象是类类型，并且两个运算对象都能准确无误地转换成string。

14.1 节练习

练习 14.1：在什么情况下重载的运算符与内置运算符有所区别？在什么情况下重载的运算符又与内置运算符一样？

练习 14.2：为Sales_data编写重载的输入、输出、加法和复合赋值运算符。

练习 14.3：string和vector都定义了重载的==以比较各自的对象，假设svec1和svec2是存放string的vector，确定在下面的表达式中分别使用了哪个版本的==？

(a) "cobble" == "stone"　　(b) svec1[0] == svec2[0]
(c) svec1 == svec2　　(d) "svec1[0] == "stone"

练习 14.4：如何确定下列运算符是否应该是类的成员？

(a) %　(b) %=　(c) ++　(d) ->　(e) <<　(f) &&　(g) ==　(h) ()

练习 14.5：在7.5.1节的练习7.40（第261页）中，编写了下列类中某一个的框架，请问在这个类中应该定义重载的运算符吗？如果是，请写出来。

(a) Book　(b) Date　(c) Employee
(d) Vehicle　(e) Object　(f) Tree

14.2 输入和输出运算符

如我们所知，IO 标准库分别使用>>和<<执行输入和输出操作。对于这两个运算符来
557 说，IO 库定义了用其读写内置类型的版本，而类则需要自定义适合其对象的新版本以支持 IO 操作。

14.2.1 重载输出运算符<<

通常情况下，输出运算符的第一个形参是一个非常量 ostream 对象的引用。之所以 ostream 是非常量是因为向流写入内容会改变其状态；而该形参是引用是因为我们无法直接复制一个 ostream 对象。

第二个形参一般来说是一个常量的引用，该常量是我们想要打印的类类型。第二个形参是引用的原因是我们希望避免复制实参；而之所以该形参可以是常量是因为（通常情况下）打印对象不会改变对象的内容。

为了与其他输出运算符保持一致，operator<<一般要返回它的 ostream 形参。

Sales_data 的输出运算符

举个例子，我们按照如下形式编写 Sales_data 的输出运算符：

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
       << item.revenue << " " << item.avg_price();
    return os;
}
```

除了名字之外，这个函数与之前的 print 函数（参见 7.1.3 节，第 234 页）完全一样。打印一个 Sales_data 对象意味着要分别打印它的三个数据成员以及通过计算得到的平均销售价格，每个元素以空格隔开。完成输出后，运算符返回刚刚使用的 ostream 的引用。

输出运算符尽量减少格式化操作

用于内置类型的输出运算符不太考虑格式化操作，尤其不会打印换行符，用户希望类的输出运算符也像如此行事。如果运算符打印了换行符，则用户就无法在对象的同一行内接着打印一些描述性的文本了。相反，令输出运算符尽量减少格式化操作可以使用户有权控制输出的细节。

Best Practices 通常，输出运算符应该主要负责打印对象的内容而非控制格式，输出运算符不应该打印换行符。

输入输出运算符必须是非成员函数

与 iostream 标准库兼容的输入输出运算符必须是普通的非成员函数，而不能是类的成员函数。否则，它们的左侧运算对象将是我们的类的一个对象：

```
Sales_data data;
data << cout;          // 如果 operator<<是 Sales_data 的成员
```

假设输入输出运算符是某个类的成员，则它们也必须是 istream 或 ostream 的成员。然而，这两个类属于标准库，并且我们无法给标准库中的类添加任何成员。

因此，如果我们希望为类自定义 IO 运算符，则必须将其定义成非成员函数。当然，IO 运算符通常需要读写类的非公有数据成员，所以 IO 运算符一般被声明为友元（参见 7.2.1 节，第 241 页）。 558

14.2.1 节练习

练习 14.6：为你的 Sales_data 类定义输出运算符。

练习 14.7：你在 13.5 节的练习（第 470 页）中曾经编写了一个 String 类，为它定义一个输出运算符。

练习 14.8：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输出运算符。



14.2.2 重载输入运算符>>

通常情况下，输入运算符的第一个形参是运算符将要读取的流的引用，第二个形参是将要读入到的（非常量）对象的引用。该运算符通常会返回某个给定流的引用。第二个形参之所以必须是个非常量是因为输入运算符本身的目的就是将数据读入到这个对象中。

Sales_data 的输入运算符

举个例子，我们将按照如下形式编写 Sales_data 的输入运算符：

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price;// 不需要初始化，因为我们将先读入数据到 price，之后才使用它
    is >> item.bookNo >> item.units_sold >> price;
    if (is)      // 检查输入是否成功
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); // 输入失败：对象被赋予默认的状态
    return is;
}
```

除了 if 语句之外，这个定义与之前的 read 函数（参见 7.1.3 节，第 234 页）完全一样。if 语句检查读取操作是否成功，如果发生了 IO 错误，则运算符将给定的对象重置为空 Sales_data，这样可以确保对象处于正确的状态。



输入运算符必须处理输入可能失败的情况，而输出运算符不需要。

输入时的错误

559

在执行输入运算符时可能发生下列错误：

- 当流含有错误类型的数据时读取操作可能失败。例如在读取完 bookNo 后，输入运算符假定接下来读入的是两个数字数据，一旦输入的不是数字数据，则读取操作及后续对流的其他使用都将失败。
- 当读取操作到达文件末尾或者遇到输入流的其他错误时也会失败。

在程序中我们没有逐个检查每个读取操作，而是等读取了所有数据后赶在使用这些数据前一次性检查：

```
if (is)                          // 检查输入是否成功
    item.revenue = item.units_sold * price;
else
    item = Sales_data();         // 输入失败：对象被赋予默认的状态
```

如果读取操作失败，则 price 的值将是未定义的。因此，在使用 price 前我们需要首先检查输入流的合法性，然后才能执行计算并将结果存入 revenue。如果发生了错误，我们无须在意到底是哪部分输入失败，只要将一个新的默认初始化的 Sales_data 对象赋予 item 从而将其重置为空 Sales_data 就可以了。执行这样的赋值后，item 的 bookNo 成员将是一个空 string，revenue 和 units_sold 成员将等于 0。

如果在发生错误前对象已经有一部分被改变，则适时地将对象置为合法状态显得异常重要。例如在这个输入运算符中，我们可能在成功读取新的 bookNo 后遇到错误，这意味着对象的 units_sold 和 revenue 成员并没有改变，因此有可能会将这两个数据与一条完全不匹配的 bookNo 组合在一起。

通过将对象置为合法的状态，我们能（略微）保护使用者免于受到输入错误的影响。此时的对象处于可用状态，即它的成员都是被正确定义的。而且该对象也不会产生误导性的结果，因为它的数据在本质上确实是一体的。



当读取操作发生错误时，输入运算符应该负责从错误中恢复。

标示错误

一些输入运算符需要做更多数据验证的工作。例如，我们的输入运算符可能需要检查 bookNo 是否符合规范的格式。在这样的例子中，即使从技术上来看 IO 是成功的，输入运算符也应该设置流的条件状态以标示出失败信息（参见 8.1.2 节，第 279 页）。通常情况下，输入运算符只设置 failbit。除此之外，设置 eofbit 表示文件耗尽，而设置 badbit 表示流被破坏。最好的方式是由 IO 标准库自己来标示这些错误。

560

14.2.2 节练习

练习 14.9：为你的 Sales_data 类定义输入运算符。

练习 14.10：对于 Sales_data 的输入运算符来说如果给定了下面的输入将发生什么情况？

(a) 0-201-99999-9 10 24.95　　(b) 10 24.95 0-210-99999-9

练习 14.11：下面的 Sales_data 输入运算符存在错误吗？如果有，请指出来。对于这个输入运算符如果仍然给定上个练习的输入将发生什么情况？

```
istream& operator>>(istream& in, Sales_data& s)
{
    double price;
    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}
```

练习 14.12：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，为它定义一个输入运算符并确保该运算符可以处理输入错误。

14.3 算术和关系运算符

通常情况下，我们把算术和关系运算符定义成非成员函数以允许对左侧或右侧的运算对象进行转换（参见 14.1 节，第 492 页）。因为这些运算符一般不需要改变运算对象的状态，所以形参都是常量的引用。

算术运算符通常会计算它的两个运算对象并得到一个新值，这个值有别于任意一个运算对象，常常位于一个局部变量之内，操作完成后返回该局部变量的副本作为其结果。如果类定义了算术运算符，则它一般也会定义一个对应的复合赋值运算符。此时，最有效的方式是使用复合赋值来定义算术运算符：

```
// 假设两个对象指向同一本书
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;          // 把 lhs 的数据成员拷贝给 sum
    sum += rhs;                    // 将 rhs 加到 sum 中
    return sum;
}
```

这个定义与原来的 add 函数（参见 7.1.3 节，第 234 页）是完全等价的。我们把 lhs 拷贝给局部变量 sum，然后使用 Sales_data 的复合赋值运算符（将在第 500 页定义）将 rhs 的值加到 sum 中，最后函数返回 sum 的副本。



如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符。

561

14.3 节练习

练习 14.13：你认为 Sales_data 类还应该支持哪些其他算术运算符（参见表 4.1，第 124 页）？如果有的话，请给出它们的定义。

练习 14.14：你觉得为什么调用 operator+=来定义 operator+比其他方法更有效？

练习 14.15：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有其他算术运算符吗？如果是，请实现它们；如果不是，解释原因。

14.3.1 相等运算符



通常情况下，C++中的类通过定义相等运算符来检验两个对象是否相等。也就是说，它们会比较对象的每一个数据成员，只有当所有对应的成员都相等时才认为两个对象相等。依据这一思想，我们的 Sales_data 类的相等运算符不但应该比较 bookNo，还应该比较具体的销售数据：

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
```

```
{
    return !(lhs == rhs);
}
```

就上面这些函数的定义本身而言，它们似乎比较简单，也没什么价值，对于我们来说重要的是从这些函数中体现出来的设计准则：

- 如果一个类含有判断两个对象是否相等的操作，则它显然应该把函数定义成 operator==而非一个普通的命名函数：因为用户肯定希望能使用==比较对象，所以提供了==就意味着用户无须再费时费力地学习并记忆一个全新的函数名字。此外，类定义了==运算符之后也更容易使用标准库容器和算法。
- 如果类定义了 operator==，则该运算符应该能判断一组给定的对象中是否含有重复数据。
- 通常情况下，相等运算符应该具有传递性，换句话说，如果 a==b 和 b==c 都为真，则 a==c 也应该为真。 562
- 如果类定义了 operator==，则这个类也应该定义 operator!=。对于用户来说，当他们能使用==时肯定也希望能使用!=，反之亦然。
- 相等运算符和不相等运算符中的一个应该把工作委托给另外一个，这意味着其中一个运算符应该负责实际比较对象的工作，而另一个运算符则只是调用那个真正工作的运算符。



如果某个类在逻辑上有相等性的含义，则该类应该定义 operator==，这样做可以使得用户更容易使用标准库算法来处理这个类。

14.3.1 节练习

练习 14.16：为你的 StrBlob 类（参见 12.1.1 节，第 405 页）、StrBlobPtr 类（参见 12.1.6 节，第 421 页）、StrVec 类（参见 13.5 节，第 465 页）和 String 类（参见 13.5 节，第 470 页）分别定义相等运算符和不相等运算符。

练习 14.17：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有相等运算符吗？如果是，请实现它；如果不是，解释原因。



14.3.2 关系运算符

定义了相等运算符的类也常常（但不总是）包含关系运算符。特别是，因为关联容器和一些算法要用到小于运算符，所以定义 operator<会比较有用。

通常情况下关系运算符应该

1．定义顺序关系，令其与关联容器中对关键字的要求一致（参见 11.2.2 节，第 378 页）；并且

2．如果类同时也含有==运算符的话，则定义一种关系令其与==保持一致。特别是，如果两个对象是!=的，那么一个对象应该<另外一个。



尽管我们可能会认为 Sales_data 类应该支持关系运算符，但事实证明并非如此，其中的缘由比较微妙，值得读者深思。

一开始我们可能会认为应该像 compareIsbn（参见 11.2.2 节，第 379 页）那样定义<，该函数通过比较 ISBN 来实现对两个对象的比较。然而，尽管 compareIsbn 提供的

顺序关系符合要求1，但是函数得到的结果显然与我们定义的==不一致，因此它不满足要求2。

对于Sales_data的==运算符来说，如果两笔交易的revenue和units_sold成员不同，那么即使它们的ISBN相同也无济于事，它们仍然是不相等的。如果我们定义的<运算符仅仅比较ISBN成员，那么将发生这样的情况：两个ISBN相同但revenue和units_sold不同的对象经比较是不相等的，但是其中的任何一个都不比另一个小。然而实际情况是，如果我们有两个对象并且哪个都不比另一个小，则从道理上来讲这两个对象应该是相等的。 563

基于上述分析我们也许会认为，只要让operator<依次比较每个数据元素就能解决问题了，比方说让operator<先比较isbn，相等的话继续比较units_sold，还相等再继续比较revenue。

然而，这样的排序没有任何必要。根据将来使用Sales_data类的实际需要，我们可能会希望先比较units_sold，也可能希望先比较revenue。有的时候，我们希望units_sold少的对象“小于”units_sold多的对象；另一些时候，则可能希望revenue少的对象“小于”revenue多的对象。

因此对于Sales_data类来说，不存在一种逻辑可靠的<定义，这个类不定义<运算符也许更好。



如果存在唯一一种逻辑可靠的<定义，则应该考虑为这个类定义<运算符。如果类同时还包含==，则当且仅当<的定义和==产生的结果一致时才定义<运算符。

14.3.2 节练习

练习14.18：为你的StrBlob类、StrBlobPtr类、StrVec类和String类定义关系运算符。

练习14.19：你在7.5.1节的练习7.40（第261页）中曾经选择并编写了一个类，你认为它应该含有关系运算符吗？如果是，请实现它；如果不是，解释原因。

14.4 赋值运算符

之前已经介绍过拷贝赋值和移动赋值运算符（参见13.1.2节，第443页和13.6.2节，第474页），它们可以把类的一个对象赋值给该类的另一个对象。此外，类还可以定义其他赋值运算符以使用别的类型作为右侧运算对象。

举个例子，在拷贝赋值和移动赋值运算符之外，标准库vector类还定义了第三种赋值运算符，该运算符接受花括号内的元素列表作为参数（参见9.2.5节，第302页）。我们能以如下的形式使用该运算符：

```
vector<string> v;
v = {"a", "an", "the"};
```

同样，也可以把这个运算符添加到StrVec类中（参见13.5节，第465页）：

```
class StrVec {
public:
    StrVec &operator=(std::initializer_list<std::string>);
    // 其他成员与13.5节（第465页）一致
};
```

564 为了与内置类型的赋值运算符保持一致（也与我们已经定义的拷贝赋值和移动赋值运算一致），这个新的赋值运算符将返回其左侧运算对象的引用：

```
StrVec &StrVec::operator=(initializer_list<string> il)
{
    // alloc_n_copy 分配内存空间并从给定范围内拷贝元素
    auto data = alloc_n_copy(il.begin(), il.end());
    free();                    // 销毁对象中的元素并释放内存空间
    elements = data.first;     // 更新数据成员使其指向新空间
    first_free = cap = data.second;
    return *this;
}
```

和拷贝赋值及移动赋值运算符一样，其他重载的赋值运算符也必须先释放当前内存空间，再创建一片新空间。不同之处是，这个运算符无须检查对象向自身的赋值，这是因为它的形参 initializer_list<string>（参见 6.2.6 节，第 198 页）确保 il 与 this 所指的不是同一个对象。



我们可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。

复合赋值运算符

复合赋值运算符不非得是类的成员，不过我们还是倾向于把包括复合赋值在内的所有赋值运算都定义在类的内部。为了与内置类型的复合赋值保持一致，类中的复合赋值运算符也要返回其左侧运算对象的引用。例如，下面是 Sales_data 类中复合赋值运算符的定义：

```
// 作为成员的二元运算符：左侧运算对象绑定到隐式的 this 指针
// 假定两个对象表示的是同一本书
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```



赋值运算符必须定义成类的成员，复合赋值运算符通常情况下也应该这样做。这两类运算符都应该返回左侧运算对象的引用。

14.4 节练习

练习 14.20：为你的 Sales_data 类定义加法和复合赋值运算符。

练习 14.21：编写 Sales_data 类的+和+=运算符，使得+执行实际的加法操作而+=调用+。相比于 14.3 节（第 497 页）和 14.4 节（第 500 页）对这两个运算符的定义，本题的定义有何缺点？试讨论之。

练习 14.22：定义赋值运算符的一个新版本，使得我们能把一个表示 ISBN 的 string 赋给一个 Sales_data 对象。

练习 14.23：为你的 StrVec 类定义一个 initializer_list 赋值运算符。

练习 14.24：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有拷贝赋值和移动赋值运算符吗？如果是，请实现它们。

练习 14.25：上题的这个类还需要定义其他赋值运算符吗？如果是，请实现它们；同时说明运算对象应该是什么类型并解释原因。

14.5 下标运算符

表示容器的类通常可以通过元素在容器中的位置访问元素，这些类一般会定义下标运算符 operator[]。



下标运算符必须是成员函数。 565

为了与下标的原始定义兼容，下标运算符通常以所访问元素的引用作为返回值，这样做的好处是下标可以出现在赋值运算符的任意一端。进一步，我们最好同时定义下标运算符的常量版本和非常量版本，当作用于一个常量对象时，下标运算符返回常量引用以确保我们不会给返回的对象赋值。



如果一个类包含下标运算符，则它通常会定义两个版本：一个返回普通引用，另一个是类的常量成员并且返回常量引用。

举个例子，我们按照如下形式定义 StrVec（参见 13.5 节，第 465 页）的下标运算符：

```
class StrVec {
public:
    std::string& operator[](std::size_t n)
        { return elements[n]; }
    const std::string& operator[](std::size_t n) const
        { return elements[n]; }
    // 其他成员与 13.5（第 465 页）一致
private:
    std::string *elements;          // 指向数组首元素的指针
};
```

上面这两个下标运算符的用法类似于 vector 或者数组中的下标。因为下标运算符返回的是元素的引用，所以当 StrVec 是非常量时，我们可以给元素赋值；而当我们对常量对象取下标时，不能为其赋值：

566

```
// 假设 svec 是一个 StrVec 对象
const StrVec cvec = svec;                  // 把 svec 的元素拷贝到 cvec 中
// 如果 svec 中含有元素，对第一个元素运行 string 的 empty 函数
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero";                      // 正确：下标运算符返回 string 的引用
    cvec[0] = "Zip";                       // 错误：对 cvec 取下标返回的是常量引用
}
```

14.5 节练习

练习 14.26：为你的 StrBlob 类、StrBlobPtr 类、StrVec 类和 String 类定义下标运算符。

14.6 递增和递减运算符

在迭代器类中通常会实现递增运算符（++）和递减运算符（--），这两种运算符使得类可以在元素的序列中前后移动。C++语言并不要求递增和递减运算符必须是类的成员，但是因为它们改变的正好是所操作对象的状态，所以建议将其设定为成员函数。

对于内置类型来说，递增和递减运算符既有前置版本也有后置版本。同样，我们也应该为类定义两个版本的递增和递减运算符。接下来我们首先介绍前置版本，然后实现后置版本。



定义递增和递减运算符的类应该同时定义前置版本和后置版本。这些运算符通常应该被定义成类的成员。

定义前置递增/递减运算符

为了说明递增和递减运算符，我们不妨在 StrBlobPtr 类（参见 12.1.6 节，第 421 页）中定义它们：

```
class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr& operator++();              // 前置运算符
    StrBlobPtr& operator--();
    // 其他成员和之前的版本一致
};
```



为了与内置版本保持一致，前置运算符应该返回递增或递减后对象的引用。

567 递增和递减运算符的工作机理非常相似：它们首先调用 check 函数检验 StrBlobPtr 是否有效，如果是，接着检查给定的索引值是否有效。如果 check 函数没有抛出异常，则运算符返回对象的引用。

在递增运算符的例子中，我们把 curr 的当前值传递给 check 函数。如果这个值小于 vector 的大小，则 check 正常返回；否则，如果 curr 已经到达了 vector 的末尾，check 将抛出异常：

```
// 前置版本：返回递增/递减对象的引用
StrBlobPtr& StrBlobPtr::operator++()
{
    // 如果 curr 已经指向了容器的尾后位置，则无法递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr;                    // 将 curr 在当前状态下向前移动一个元素
    return *this;
}
```

```
StrBlobPtr& StrBlobPtr::operator--()
{
    // 如果 curr 是 0，则继续递减它将产生一个无效下标
    --curr;                          // 将 curr 在当前状态下向后移动一个元素
    check(curr, "decrement past begin of StrBlobPtr");
    return *this;
}
```

递减运算符先递减 curr，然后调用 check 函数。此时，如果 curr（一个无符号数）已经是 0 了，那么我们传递给 check 的值将是一个表示无效下标的非常大的正数值（参见 2.1.2 节，第 33 页）。

区分前置和后置运算符

要想同时定义前置和后置运算符，必须首先解决一个问题，即普通的重载形式无法区分这两种情况。前置和后置版本使用的是同一个符号，意味着其重载版本所用的名字将是相同的，并且运算对象的数量和类型也相同。

为了解决这个问题，后置版本接受一个额外的（不被使用）int 类型的形参。当我们使用后置运算符时，编译器为这个形参提供一个值为 0 的实参。尽管从语法上来说后置函数可以使用这个额外的形参，但是在实际过程中通常不会这么做。这个形参的唯一作用就是区分前置版本和后置版本的函数，而不是真的要在实现后置版本时参与运算。

接下来我们为 StrBlobPtr 添加后置运算符：

```
class StrBlobPtr {
public:
    // 递增和递减运算符
    StrBlobPtr operator++(int);       // 后置运算符
    StrBlobPtr operator--(int);
    // 其他成员和之前的版本一致
};
```



为了与内置版本保持一致，后置运算符应该返回对象的原值（递增或递减之前的值），返回的形式是一个值而非引用。 568

对于后置版本来说，在递增对象之前需要首先记录对象的状态：

```
// 后置版本：递增/递减对象的值但是返回原值
StrBlobPtr StrBlobPtr::operator++(int)
{
    // 此处无须检查有效性，调用前置递增运算时才需要检查
    StrBlobPtr ret = *this;  // 记录当前的值
    ++*this;                 // 向前移动一个元素，前置++需要检查递增的有效性
    return ret;              // 返回之前记录的状态
}
StrBlobPtr StrBlobPtr::operator--(int)
{
    // 此处无须检查有效性，调用前置递减运算时才需要检查
    StrBlobPtr ret = *this;  // 记录当前的值
    --*this;                 // 向后移动一个元素，前置--需要检查递减的有效性
    return ret;              // 返回之前记录的状态
}
```

由上可知，我们的后置运算符调用各自的前置版本来完成实际的工作。例如后置递增运算符执行

```
++*this
```

该表达式调用前置递增运算符，前置递增运算符首先检查递增操作是否安全，根据检查的结果抛出一个异常或者执行递增 curr 的操作。假定通过了检查，则后置函数返回事先存好的 ret 的副本。因此最终的效果是，对象本身向前移动了一个元素，而返回的结果仍然反映对象在未递增之前原始的值。



因为我们不会用到 int 形参，所以无须为其命名。

显式地调用后置运算符

如在第 491 页介绍的，可以显式地调用一个重载的运算符，其效果与在表达式中以运算符号的形式使用它完全一样。如果我们想通过函数调用的方式调用后置版本，则必须为它的整型参数传递一个值：

```
StrBlobPtr p(a1);              // p 指向 a1 中的 vector
p.operator++(0);               // 调用后置版本的 operator++
p.operator++();                // 调用前置版本的 operator++
```

尽管传入的值通常会被运算符函数忽略，但却必不可少，因为编译器只有通过它才能知道应该使用后置版本。

569

14.6 节练习

练习 14.27：为你的 StrBlobPtr 类添加递增和递减运算符。

练习 14.28：为你的 StrBlobPtr 类添加加法和减法运算符，使其可以实现指针的算术运算（参见 3.5.3 节，第 106 页）。

练习 14.29：为什么不定义 const 版本的递增和递减运算符？

14.7 成员访问运算符

在迭代器类及智能指针类（参见 12.1 节，第 400 页）中常常用到解引用运算符（*）和箭头运算符（->）。我们以如下形式向 StrBlobPtr 类添加这两种运算符：

```
class StrBlobPtr {
public:
    std::string& operator*() const
    { auto p = check(curr, "dereference past end");
    return (*p)[curr];         // (*p)是对象所指的 vector
    }
    std::string* operator->() const
    { // 将实际工作委托给解引用运算符
     return & this->operator*();
    }
    // 其他成员与之前的版本一致
}
```

解引用运算符首先检查 curr 是否仍在作用范围内，如果是，则返回 curr 所指元素的一个引用。箭头运算符不执行任何自己的操作，而是调用解引用运算符并返回解引用结果元素的地址。



箭头运算符必须是类的成员。解引用运算符通常也是类的成员，尽管并非必须如此。

值得注意的是，我们将这两个运算符定义成了 const 成员，这是因为与递增和递减运算符不一样，获取一个元素并不会改变 StrBlobPtr 对象的状态。同时，它们的返回值分别是非常量 string 的引用或指针，因为一个 StrBlobPtr 只能绑定到非常量的 StrBlob 对象（参见 12.1.6 节，第 421 页）。

这两个运算符的用法与指针或者 vector 迭代器的对应操作完全一致：

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1);                    // p 指向 a1 中的 vector
*p = "okay";                         // 给 a1 的首元素赋值
cout << p->size() << endl;           // 打印 4，这是 a1 首元素的大小
cout << (*p).size() << endl;         // 等价于 p->size()
```

对箭头运算符返回值的限定

570

和大多数其他运算符一样（尽管这么做不太好），我们能令 operator*完成任何我们指定的操作。换句话说，我们可以让 operator*返回一个固定值 42，或者打印对象的内容，或者其他。箭头运算符则不是这样，它永远不能丢掉成员访问这个最基本的含义。当我们重载箭头时，可以改变的是箭头从哪个对象当中获取成员，而箭头获取成员这一事实则永远不变。

对于形如 point->mem 的表达式来说，point 必须是指向类对象的指针或者是一个重载了 operator->的类的对象。根据 point 类型的不同，point->mem 分别等价于

```
(*point).mem;                        // point 是一个内置的指针类型
point.operator()->mem;               // point 是类的一个对象
```

除此之外，代码都将发生错误。point->mem 的执行过程如下所示：

1. 如果 point 是指针，则我们应用内置的箭头运算符，表达式等价于(*point).mem。首先解引用该指针，然后从所得的对象中获取指定的成员。如果 point 所指的类型没有名为 mem 的成员，程序会发生错误。
2. 如果point 是定义了 operator->的类的一个对象，则我们使用 point.operator->()的结果来获取 mem。其中，如果该结果是一个指针，则执行第 1 步；如果该结果本身含有重载的 operator->()，则重复调用当前步骤。最终，当这一过程结束时程序或者返回了所需的内容，或者返回一些表示程序错误的信息。



重载的箭头运算符必须返回类的指针或者自定义了箭头运算符的某个类的对象。

14.7 节练习

练习 14.30：为你的 StrBlobPtr 类和在 12.1.6 节练习 12.22（第 423 页）中定义的 ConstStrBlobPtr 类分别添加解引用运算符和箭头运算符。注意：因为 ConstStrBlobPtr 的数据成员指向 const vector，所以 ConstStrBlobPtr 中的运算符必须返回常量引用。

练习 14.31：我们的 StrBlobPtr 类没有定义拷贝构造函数、赋值运算符及析构函数，为什么？

练习 14.32：定义一个类令其含有指向 StrBlobPtr 对象的指针，为这个类定义重载的箭头运算符。

14.8 函数调用运算符

571 如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。因为这样的类同时也能存储状态，所以与普通函数相比它们更加灵活。

举个简单的例子，下面这个名为 absInt 的 struct 含有一个调用运算符，该运算符负责返回其参数的绝对值：

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};
```

这个类只定义了一种操作：函数调用运算符，它负责接受一个 int 类型的实参，然后返回该实参的绝对值。

我们使用调用运算符的方式是令一个 absInt 对象作用于一个实参列表，这一过程看起来非常像调用函数的过程：

```
int i = -42;
absInt absObj;                  // 含有函数调用运算符的对象
int ui = absObj(i);             // 将 i 传递给 absObj.operator()
```

即使 absObj 只是一个对象而非函数，我们也能“调用”该对象。调用对象实际上是在运行重载的调用运算符。在此例中，该运算符接受一个 int 值并返回其绝对值。



函数调用运算符必须是成员函数。一个类可以定义多个不同版本的调用运算符，相互之间应该在参数数量或类型上有所区别。

如果类定义了调用运算符，则该类的对象称作**函数对象**（function object）。因为可以调用这种对象，所以我们说这些对象的“行为像函数一样”。

含有状态的函数对象类

和其他类一样，函数对象类除了 operator() 之外也可以包含其他成员。函数对象类通常含有一些数据成员，这些成员被用于定制调用运算符中的操作。

举个例子，我们将定义一个打印 string 实参内容的类。默认情况下，我们的类会将

内容写入到 cout 中，每个 string 之间以空格隔开。同时也允许类的用户提供其他可写入的流及其他分隔符。我们将该类定义如下：

```
class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) { }
    void operator()(const string &s) const { os << s << sep; }
private:
    ostream &os;                    // 用于写入的目的流
    char sep;                       // 用于将不同输出隔开的字符
};
```

我们的类有一个构造函数，它接受一个输出流的引用以及一个用于分隔的字符，这两个形参的默认实参（参见 6.5.1 节，第 211 页）分别是 cout 和空格。之后的函数调用运算符使用这些成员协助其打印给定的 string。 <572>

当定义 PrintString 的对象时，对于分隔符及输出流既可以使用默认值也可以提供我们自己的值：

```
PrintString printer;            // 使用默认值，打印到 cout
printer(s);                     // 在 cout 中打印 s，后面跟一个空格
PrintString errors(cerr, '\n');
errors(s);                      // 在 cerr 中打印 s，后面跟一个换行符
```

函数对象常常作为泛型算法的实参。例如，可以使用标准库 for_each 算法（参见 10.3.2 节，第 348 页）和我们自己的 PrintString 类来打印容器的内容：

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

for_each 的第三个实参是类型 PrintString 的一个临时对象，其中我们用 cerr 和换行符初始化了该对象。当程序调用 for_each 时，将会把 vs 中的每个元素依次打印到 cerr 中，元素之间以换行符分隔。

14.8 节练习

练习 14.33：一个重载的函数调用运算符应该接受几个运算对象？

练习 14.34：定义一个函数对象类，令其执行 if-then-else 的操作：该类的调用运算符接受三个形参，它首先检查第一个形参，如果成功返回第二个形参的值；如果不成功返回第三个形参的值。

练习 14.35：编写一个类似于 PrintString 的类，令其从 istream 中读取一行输入，然后返回一个表示我们所读内容的 string。如果读取失败，返回空 string。

练习 14.36：使用前一个练习定义的类读取标准输入，将每一行保存为 vector 的一个元素。

练习 14.37：编写一个类令其检查两个值是否相等。使用该对象及标准库算法编写程序，令其替换某个序列中具有给定值的所有实例。

14.8.1 lambda 是函数对象

在前一节中，我们使用一个 PrintString 对象作为调用 for_each 的实参，这一

用法类似于我们在 10.3.2 节（第 346 页）中编写的使用 lambda 表达式的程序。当我们编写了一个 lambda 后，编译器将该表达式翻译成一个未命名类的未命名对象（参见 10.3.3 节，第 349 页）。在 lambda 表达式产生的类中含有一个重载的函数调用运算符，例如，对于我们传递给 stable_sort 作为其最后一个实参的 lambda 表达式来说：

573

```
// 根据单词的长度对其进行排序，对于长度相同的单词按照字母表顺序排序
stable_sort(words.begin(), words.end(),
            [](const string &a, const string &b)
              { return a.size() < b.size();});
```

其行为类似于下面这个类的一个未命名对象

```
class ShorterString {
public:
    bool operator()(const string &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};
```

产生的类只有一个函数调用运算符成员，它负责接受两个 string 并比较它们的长度，它的形参列表和函数体与 lambda 表达式完全一样。如我们在 10.3.3 节（第 352 页）所见，默认情况下 lambda 不能改变它捕获的变量。因此在默认情况下，由 lambda 产生的类当中的函数调用运算符是一个 const 成员函数。如果 lambda 被声明为可变的，则调用运算符就不是 const 的了。

用这个类替代 lambda 表达式后，我们可以重写并重新调用 stable_sort：

```
stable_sort(words.begin(), words.end(), ShorterString());
```

第三个实参是新构建的 ShorterString 对象，当 stable_sort 内部的代码每次比较两个 string 时就会“调用”这一对象，此时该对象将调用运算符的函数体，判断第一个 string 的大小小于第二个时返回 true。

表示 lambda 及相应捕获行为的类

如我们所知，当一个 lambda 表达式通过引用捕获变量时，将由程序负责确保 lambda 执行时引用所引的对象确实存在（参见 10.3.3 节，第 350 页）。因此，编译器可以直接使用该引用而无须在 lambda 产生的类中将其存储为数据成员。

相反，通过值捕获的变量被拷贝到 lambda 中（参见 10.3.3 节，第 350 页）。因此，这种 lambda 产生的类必须为每个值捕获的变量建立对应的数据成员，同时创建构造函数，令其使用捕获的变量的值来初始化数据成员。举个例子，在 10.3.2 节（第 347 页）中有一个 lambda，它的作用是找到第一个长度不小于给定值的 string 对象：

```
// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(),
            [sz](const string &a)
                { return a.size() >= sz;});
```

该 lambda 表达式产生的类将形如：

574

```
class SizeComp {
    SizeComp(size_t n): sz(n) { }     // 该形参对应捕获的变量
    // 该调用运算符的返回类型、形参和函数体都与 lambda 一致
    bool operator()(const string &s) const
        { return s.size() >= sz; }
```

```
private:
    size_t sz;                          // 该数据成员对应通过值捕获的变量
};
```

和我们的 ShorterString 类不同，上面这个类含有一个数据成员以及一个用于初始化该成员的构造函数。这个合成的类不含有默认构造函数，因此要想使用这个类必须提供一个实参：

```
// 获得第一个指向满足条件元素的迭代器，该元素满足 size() is >= sz
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
```

lambda 表达式产生的类不含默认构造函数、赋值运算符及默认析构函数；它是否含有默认的拷贝/移动构造函数则通常要视捕获的数据成员类型而定（参见 13.1.6 节，第 450 页和 13.6.2 节，第 475 页）。

14.8.1 节练习

练习 14.38：编写一个类令其检查某个给定的 string 对象的长度是否与一个阈值相等。使用该对象编写程序，统计并报告在输入的文件中长度为 1 的单词有多少个、长度为 2 的单词有多少个、……、长度为 10 的单词又有多少个。

练习 14.39：修改上一题的程序令其报告长度在 1 至 9 之间的单词有多少个、长度在 10 以上的单词又有多少个。

练习 14.40：重新编写 10.3.2 节（第 349 页）的 biggies 函数，使用函数对象类替换其中的 lambda 表达式。

练习 14.41：你认为 C++11 新标准为什么要增加 lambda？对于你自己来说，什么情况下会使用 lambda，什么情况下会使用类？

14.8.2 标准库定义的函数对象

标准库定义了一组表示算术运算符、关系运算符和逻辑运算符的类，每个类分别定义了一个执行命名操作的调用运算符。例如，plus 类定义了一个函数调用运算符用于对一对运算对象执行+的操作；modulus 类定义了一个调用运算符执行二元的%操作；equal_to 类执行==，等等。

这些类都被定义成模板的形式，我们可以为其指定具体的应用类型，这里的类型即调用运算符的形参类型。例如，plus<string>令 string 加法运算符作用于 string 对象；plus<int>的运算对象是 int；plus<Sales_data>对 Sales_data 对象执行加法运算，以此类推：

```
plus<int> intAdd;               // 可执行 int 加法的函数对
negate<int> intNegate;          // 可对 int 值取反的函数对象
// 使用 intAdd::operator(int, int)求 10 和 20 的和
int sum = intAdd(10, 20);                   // 等价于 sum = 30
sum = intNegate(intAdd(10, 20));            // 等价于 sum = 30
// 使用 intNegate::operator(int)生成-10
// 然后将-10 作为 intAdd::operator(int, int)的第二个参数
sum = intAdd(10, intNegate(10));            // sum = 0
```

575

表 14.2 所列的类型定义在 functional 头文件中。

表 14.2：标准库函数对象

算术	关系	逻辑
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

在算法中使用标准库函数对象

表示运算符的函数对象类常用来替换算法中的默认运算符。如我们所知，在默认情况下排序算法使用 operator<将序列按照升序排列。如果要执行降序排列的话，我们可以传入一个 greater 类型的对象。该类将产生一个调用运算符并负责执行待排序类型的大于运算。例如，如果 svec 是一个 vector<string>，

```
// 传入一个临时的函数对象用于执行两个 string 对象的>比较运算
sort(svec.begin(), svec.end(), greater<string>());
```

则上面的语句将按照降序对 svec 进行排序。第三个实参是 greater<string>类型的一个未命名的对象，因此当 sort 比较元素时，不再是使用默认的<运算符，而是调用给定的 greater 函数对象。该对象负责在 string 元素之间执行>比较运算。

需要特别注意的是，标准库规定其函数对象对于指针同样适用。我们之前曾经介绍过比较两个无关指针将产生未定义的行为（参见 3.5.3 节，第 107 页），然而我们可能会希望通过比较指针的内存地址来 sort 指针的 vector。直接这么做将产生未定义的行为，因此我们可以使用一个标准库函数对象来实现该目的：

```
vector<string *> nameTable;                    // 指针的 vector
// 错误：nameTable 中的指针彼此之间没有关系，所以<将产生未定义的行为
sort(nameTable.begin(), nameTable.end(),
     [](string *a, string *b) { return a < b; });
// 正确：标准库规定指针的 less 是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

576 关联容器使用 less<key_type>对元素排序，因此我们可以定义一个指针的 set 或者在 map 中使用指针作为关键值而无须直接声明 less。

14.8.2 节练习

练习 14.42：使用标准库函数对象及适配器定义一条表达式，令其

(a) 统计大于 1024 的值有多少个。
(b) 找到第一个不等于 pooh 的字符串。
(c) 将所有的值乘以 2。

练习 14.43：使用标准库函数对象判断一个给定的 int 值是否能被 int 容器中的所有元素整除。

14.8.3 可调用对象与 function

C++语言中有几种可调用的对象：函数、函数指针、lambda 表达式（参见 10.3.2 节，第 346 页）、bind 创建的对象（参见 10.3.4 节，第 354 页）以及重载了函数调用运算符的类。

和其他对象一样，可调用的对象也有类型。例如，每个 lambda 有它自己唯一的（未命名）类类型；函数及函数指针的类型则由其返回值类型和实参类型决定，等等。

然而，两个不同类型的可调用对象却可能共享同一种**调用形式**（call signature）。调用形式指明了调用返回的类型以及传递给调用的实参类型。一种调用形式对应一个函数类型，例如：

```
int(int, int)
```

是一个函数类型，它接受两个 int、返回一个 int。

不同类型可能具有相同的调用形式

对于几个可调用对象共享同一种调用形式的情况，有时我们会希望把它们看成具有相同的类型。例如，考虑下列不同类型的可调用对象：

```
// 普通函数
int add(int i, int j) { return i + j; }
// lambda，其产生一个未命名的函数对象类
auto mod = [](int i, int j) { return i % j; };
// 函数对象类
struct divide {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
```

上面这些可调用对象分别对其参数执行了不同的算术运算，尽管它们的类型各不相同，但是共享同一种调用形式：

577

```
int(int, int)
```

我们可能希望使用这些可调用对象构建一个简单的桌面计算器。为了实现这一目的，需要定义一个**函数表**（function table）用于存储指向这些可调用对象的“指针”。当程序需要执行某个特定的操作时，从表中查找该调用的函数。

在 C++语言中，函数表很容易通过 map 来实现。对于此例来说，我们使用一个表示运算符符号的 string 对象作为关键字；使用实现运算符的函数作为值。当我们需要求给定运算符的值时，先通过运算符索引 map，然后调用找到的那个元素。

假定我们的所有函数都相互独立，并且只处理关于 int 的二元运算，则 map 可以定义成如下的形式：

```
// 构建从运算符到函数指针的映射关系，其中函数接受两个 int、返回一个 int
map<string, int(*)(int,int)> binops;
```

我们可以按照下面的形式将 add 的指针添加到 binops 中：

```
// 正确：add 是一个指向正确类型函数的指针
binops.insert({"+", add});    // {"+", add}是一个 pair（参见 11.2.3 节，379 页）
```

但是我们不能将 mod 或者 divide 存入 binops：

```
binops.insert({"%", mod});          // 错误：mod 不是一个函数指针
```

问题在于 mod 是个 lambda 表达式，而每个 lambda 有它自己的类类型，该类型与存储在 binops 中的值的类型不匹配。

标准库 function 类型

C++ 11

我们可以使用一个名为 **function** 的新的标准库类型解决上述问题，function 定义在 functional 头文件中，表 14.3 列举出了 function 定义的操作。

表 14.3：function 的操作

function<T> f;	f 是一个用来存储可调用对象的空 function，这些可调用对象的调用形式应该与函数类型 T 相同（即 T 是 *retType(args)*）
function<T> f(nullptr);	显式地构造一个空 function
function<T> f(obj);	在 f 中存储可调用对象 obj 的副本
f	将 f 作为条件：当 f 含有一个可调用对象时为真；否则为假
f(*args*)	调用 f 中的对象，参数是 *args*
定义为 function<T>的成员的类型	
result_type	该 function 类型的可调用对象返回的类型
argument_type	当 T 有一个或两个实参时定义的类型。如果 T 只有一个实参，则 argument_type 是该类型的同义词；如果 T 有两个实参，则 first_argument_type 和 second_argument_type 分别代表两个实参的类型
first_argument_type	
second_argument_type	

function 是一个模板，和我们使用过的其他模板一样，当创建一个具体的 function 类型时我们必须提供额外的信息。在此例中，所谓额外的信息是指该 function 类型能够表示的对象的调用形式。参考其他模板，我们在一对尖括号内指定类型：

```
function<int(int, int)>
```

在这里我们声明了一个 function 类型，它可以表示接受两个 int、返回一个 int 的可调用对象。因此，我们可以用这个新声明的类型表示任意一种桌面计算器用到的类型：

```
function<int(int, int)> f1 = add;              // 函数指针
function<int(int, int)> f2 = divide();         // 函数对象类的对象
function<int(int, int)> f3 = [](int i, int j) // lambda
                             { return i * j; };
cout << f1(4,2) << endl;                  // 打印 6
cout << f2(4,2) << endl;                  // 打印 2
cout << f3(4,2) << endl;                  // 打印 8
```

578 使用这个 function 类型我们可以重新定义 map：

```
// 列举了可调用对象与二元运算符对应关系的表格
// 所有可调用对象都必须接受两个 int、返回一个 int
// 其中的元素可以是函数指针、函数对象或者 lambda
map<string, function<int(int, int)>> binops;
```

我们能把所有可调用对象，包括函数指针、lambda 或者函数对象在内，都添加到这个 map 中：

```
map<string, function<int(int, int)>> binops = {
    {"+", add},                                  // 函数指针
    {"-", std::minus<int>()},                    // 标准库函数对象
    {"/", divide()},                             // 用户定义的函数对象
    {"*", [](int i, int j) { return i * j; }},   // 未命名的 lambda
    {"%", mod} };                                // 命名了的 lambda 对象
```

我们的 map 中包含 5 个元素，尽管其中的可调用对象的类型各不相同，我们仍然能够把所有这些类型都存储在同一个 function<int (int , int)>类型中。

一如往常，当我们索引 map 时将得到关联值的一个引用。如果我们索引 binops，将得到 function 对象的引用。function 类型重载了调用运算符，该运算符接受它自己的实参然后将其传递给存好的可调用对象：

```
binops["+"](10, 5); // 调用 add(10, 5)
binops["-"](10, 5); // 使用 minus<int>对象的调用运算符
binops["/"](10, 5); // 使用 divide 对象的调用运算符
binops["*"](10, 5); // 调用 lambda 函数对象
binops["%"](10, 5); // 调用 lambda 函数对象
```

我们依次调用了 binops 中存储的每个操作。在第一个调用中，我们获得的元素存放着一个指向 add 函数的指针，因此调用 binops["+"](10, 5) 实际上是使用该指针调用 add，并传入 10 和 5。在接下来的调用中，binops["-"]返回一个存放着 std::minus<int> 类型对象的 function，我们将执行该对象的调用运算符。

重载的函数与 function

我们不能（直接）将重载函数的名字存入 function 类型的对象中：

```
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data&, const Sales_data&);
map<string, function<int(int, int)>> binops;
binops.insert( {"+", add} );      // 错误：哪个 add?
```

解决上述二义性问题的一条途径是存储函数指针（参见 6.7 节，第 221 页）而非函数的名字：

```
int (*fp)(int,int) = add;         // 指针所指的 add 是接受两个 int 的版本
binops.insert( {"+", fp} );       // 正确：fp 指向一个正确的 add 版本
```

同样，我们也能使用 lambda 来消除二义性：

579

```
// 正确：使用 lambda 来指定我们希望使用的 add 版本
binops.insert( {"+", [](int a, int b) {return add(a, b);} } );
```

lambda 内部的函数调用传入了两个 int，因此该调用只能匹配接受两个 int 的 add 版本，而这也正是执行 lambda 时真正调用的函数。



新版本标准库中的 function 类与旧版本中的 unary_function 和 binary_function 没有关联，后两个类已经被更通用的 bind 函数替代了（参见 10.3.4 节，第 357 页）。

14.8.3 节练习

练习 14.44：编写一个简单的桌面计算器使其能处理二元运算。

14.9 重载、类型转换与运算符

在 7.5.4 节（第 263 页）中我们看到由一个实参调用的非显式构造函数定义了一种隐式的类型转换，这种构造函数将实参类型的对象转换成类类型。我们同样能定义对于类类型的类型转换，通过定义类型转换运算符可以做到这一点。转换构造函数和类型转换运算符共同定义了**类类型转换**（class-type conversions），这样的转换有时也被称作**用户定义的类型转换**（user-defined conversions）。

580

14.9.1 类型转换运算符

类型转换运算符（conversion operator）是类的一种特殊成员函数，它负责将一个类类型的值转换成其他类型。类型转换函数的一般形式如下所示：

```
operator type() const;
```

其中 *type* 表示某种类型。类型转换运算符可以面向任意类型（除了 void 之外）进行定义，只要该类型能作为函数的返回类型（参见 6.1 节，第 184 页）。因此，我们不允许转换成数组或者函数类型，但允许转换成指针（包括数组指针及函数指针）或者引用类型。

类型转换运算符既没有显式的返回类型，也没有形参，而且必须定义成类的成员函数。类型转换运算符通常不应该改变待转换对象的内容，因此，类型转换运算符一般被定义成 const 成员。



一个类型转换函数必须是类的成员函数；它不能声明返回类型，形参列表也必须为空。类型转换函数通常应该是 const。

定义含有类型转换运算符的类

举个例子，我们定义一个比较简单的类，令其表示 0 到 255 之间的一个整数：

```
class SmallInt {
public:
    SmallInt(int i = 0): val(i)
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

我们的 SmallInt 类既定义了向类类型的转换，也定义了从类类型向其他类型的转换。其中，构造函数将算术类型的值转换成 SmallInt 对象，而类型转换运算符将 SmallInt 对象转换成 int：

```
SmallInt si;
```

```
si = 4;        // 首先将 4 隐式地转换成 SmallInt，然后调用 SmallInt::operator=
si + 3;        // 首先将 si 隐式地转换成 int，然后执行整数的加法
```

尽管编译器一次只能执行一个用户定义的类型转换（参见 4.11.2 节，第 144 页），但是隐式的用户定义类型转换可以置于一个标准（内置）类型转换之前或之后（参见 4.11.1 节，第 141 页），并与其一起使用。因此，我们可以将任何算术类型传递给 SmallInt 的构造函数。类似的，我们也能使用类型转换运算符将一个 SmallInt 对象转换成 int，然后再将所得的 int 转换成任何其他算术类型：

581

```
// 内置类型转换将 double 实参转换成 int
SmallInt si = 3.14;        // 调用 SmallInt(int)构造函数
// SmallInt 的类型转换运算符将 si 转换成 int
si + 3.14;                 // 内置类型转换将所得的 int 继续转换成 double
```

因为类型转换运算符是隐式执行的，所以无法给这些函数传递实参，当然也就不能在类型转换运算符的定义中使用任何形参。同时，尽管类型转换函数不负责指定返回类型，但实际上每个类型转换函数都会返回一个对应类型的值：

```
class SmallInt;
operator int(SmallInt&);                    // 错误：不是成员函数
class SmallInt {
public:
    int operator int() const;               // 错误：指定了返回类型
    operator int(int = 0) const;            // 错误：参数列表不为空
    operator int*() const { return 42; }    // 错误：42 不是一个指针
};
```

提示：避免过度使用类型转换函数

和使用重载运算符的经验一样，明智地使用类型转换运算符也能极大地简化类设计者的工作，同时使得使用类更加容易。然而，如果在类类型和转换类型之间不存在明显的映射关系，则这样的类型转换可能具有误导性。

例如，假设某个类表示 Date，我们也许会为它添加一个从 Date 到 int 的转换。然而，类型转换函数的返回值应该是什么？一种可能的解释是，函数返回一个十进制数，依次表示年、月、日，例如，July 30, 1989 可能转换为 int 值 19890730。同时还存在另外一种合理的解释，即类型转换运算符返回的 int 表示的是从某个时间节点（比如 January 1, 1970）开始经过的天数。显然这两种理解都合情合理，毕竟从形式上看它们产生的效果都是越靠后的日期对应的整数值越大，而且两种转换都有实际的用处。

问题在于 Date 类型的对象和 int 类型的值之间不存在明确的一对一映射关系。因此在此例中，不定义该类型转换运算符也许会更好。作为替代的手段，类可以定义一个或多个普通的成员函数以从各种不同形式中提取所需的信息。

类型转换运算符可能产生意外结果

在实践中，类很少提供类型转换运算符。在大多数情况下，如果类型转换自动发生，用户可能会感觉比较意外，而不是感觉受到了帮助。然而这条经验法则存在一种例外情况：对于类来说，定义向 bool 的类型转换还是比较普遍的现象。

582

在 C++标准的早期版本中，如果类想定义一个向 bool 的类型转换，则它常常遇到一个问题：因为 bool 是一种算术类型，所以类类型的对象转换成 bool 后就能被用在任何

需要算术类型的上下文中。这样的类型转换可能引发意想不到的结果，特别是当 istream 含有向 bool 的类型转换时，下面的代码仍将编译通过：

```
int i = 42;
cin << i;  // 如果向 bool 的类型转换不是显式的，则该代码在编译器看来将是合法的！
```

这段程序试图将输出运算符作用于输入流。因为 istream 本身并没有定义<<，所以本来代码应该产生错误。然而，该代码能使用 istream 的 bool 类型转换运算符将 cin 转换成 bool，而这个 bool 值接着会被提升成 int 并用作内置的左移运算符的左侧运算对象。这样一来，提升后的 bool 值（1 或 0）最终会被左移 42 个位置。这一结果显然与我们的预期大相径庭。

显式的类型转换运算符

C++ 11

为了防止这样的异常情况发生，C++11 新标准引入了**显式的类型转换运算符**（explicit conversion operator）：

```
class SmallInt {
public:
    // 编译器不会自动执行这一类型转换
    explicit operator int() const { return val; }
    // 其他成员与之前的版本一致
};
```

和显式的构造函数（参见 7.5.4 节，第 265 页）一样，编译器（通常）也不会将一个显式的类型转换运算符用于隐式类型转换：

```
SmallInt si = 3;      // 正确：SmallInt 的构造函数不是显式的
si + 3;               // 错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3;     // 正确：显式地请求类型转换
```

当类型转换运算符是显式的时，我们也能执行类型转换，不过必须通过显式的强制类型转换才可以。

该规定存在一个例外，即如果表达式被用作条件，则编译器会将显式的类型转换自动应用于它。换句话说，当表达式出现在下列位置时，显式的类型转换将被隐式地执行：

- if、while 及 do 语句的条件部分
- for 语句头的条件表达式
- 逻辑非运算符（!）、逻辑或运算符（||）、逻辑与运算符（&&）的运算对象
- 条件运算符（? :）的条件表达式。

583

转换为 bool

在标准库的早期版本中，IO 类型定义了向 void* 的转换规则，以求避免上面提到的问题。在 C++11 新标准下，IO 标准库通过定义一个向 bool 的显式类型转换实现同样的目的。

无论我们什么时候在条件中使用流对象，都会使用为 IO 类型定义的 operator bool。例如：

```
while (std::cin >> value)
```

while 语句的条件执行输入运算符，它负责将数据读入到 value 并返回 cin。为了对条件求值，cin 被 istream operator bool 类型转换函数隐式地执行了转换。如果 cin 的条件状态是 good（参见 8.1.2 节，第 280 页），则该函数返回为真；否则该函数返回为假。



向 bool 的类型转换通常用在条件部分，因此 operator bool 一般定义成 explicit 的。

14.9.1 节练习

练习 14.45：编写类型转换运算符将一个 Sales_data 对象分别转换成 string 和 double，你认为这些运算符的返回值应该是什么？

练习 14.46：你认为应该为 Sales_data 类定义上面两种类型转换运算符吗？应该把它们声明成 explicit 的吗？为什么？

练习 14.47：说明下面这两个类型转换运算符的区别。

```
struct Integral {
    operator const int();
    operator int() const;
};
```

练习 14.48：你在 7.5.1 节的练习 7.40（第 261 页）中曾经选择并编写了一个类，你认为它应该含有向 bool 的类型转换运算符吗？如果是，解释原因并说明该运算符是否应该是 explicit 的；如果不是，也请解释原因。

练习 14.49：为上一题提到的类定义一个转换目标是 bool 的类型转换运算符，先不用在意这么做是否应该。

14.9.2 避免有二义性的类型转换



如果类中包含一个或多个类型转换，则必须确保在类类型和目标类型之间只存在唯一一种转换方式。否则的话，我们编写的代码将很可能会具有二义性。

在两种情况下可能产生多重转换路径。第一种情况是两个类提供相同的类型转换：例如，当 A 类定义了一个接受 B 类对象的转换构造函数，同时 B 类定义了一个转换目标是 A 类的类型转换运算符时，我们就说它们提供了相同的类型转换。

第二种情况是类定义了多个转换规则，而这些转换涉及的类型本身可以通过其他类型转换联系在一起。最典型的例子是算术运算符，对某个给定的类来说，最好只定义最多一个与算术类型有关的转换规则。 584



通常情况下，不要为类定义相同的类型转换，也不要在类中定义两个及两个以上转换源或转换目标是算术类型的转换。

实参匹配和相同的类型转换

在下面的例子中，我们定义了两种将 B 转换成 A 的方法：一种使用 B 的类型转换运算符、另一种使用 A 的以 B 为参数的构造函数：

```
// 最好不要在两个类之间构建相同的类型转换
struct B;
struct A {
    A() = default;
    A(const B&);          // 把一个 B 转换成 A
    // 其他数据成员
```

```
};
struct B {
    operator A() const;  // 也是把一个 B 转换成 A
    // 其他数据成员
};
A f(const A&);
B b;
A a = f(b);  // 二义性错误：含义是 f(B::operator A())
             // 还是 f(A::A(const B&))?
```

因为同时存在两种由 B 获得 A 的方法，所以造成编译器无法判断应该运行哪个类型转换，也就是说，对 f 的调用存在二义性。该调用可以使用以 B 为参数的 A 的构造函数，也可以使用 B 当中把 B 转换成 A 的类型转换运算符。因为这两个函数效果相当、难分伯仲，所以该调用将产生错误。

如果我们确实想执行上述的调用，就不得不显式地调用类型转换运算符或者转换构造函数：

```
A a1 = f(b.operator A());     // 正确：使用 B 的类型转换运算符
A a2 = f(A(b));               // 正确：使用 A 的构造函数
```

值得注意的是，我们无法使用强制类型转换来解决二义性问题，因为强制类型转换本身也面临二义性。

二义性与转换目标为内置类型的多重类型转换

另外如果类定义了一组类型转换，它们的转换源（或者转换目标）类型本身可以通过其他类型转换联系在一起，则同样会产生二义性的问题。最简单也是最困扰我们的例子就是类当中定义了多个参数都是算术类型的构造函数，或者转换目标都是算术类型的类型转换运算符。

例如，在下面的类中包含两个转换构造函数，它们的参数是两种不同的算术类型；同时还包含两个类型转换运算符，它们的转换目标也恰好是两种不同的算术类型：

585

```
struct A {
    A(int = 0);             // 最好不要创建两个转换源都是算术类型的类型转换
    A(double);
    operator int() const; // 最好不要创建两个转换对象都是算术类型的类型转换
    operator double() const;
    // 其他成员
};
void f2(long double);
A a;
f2(a);        // 二义性错误：含义是 f(A::operator int())
              // 还是 f(A::operator double())?
long lg;
A a2(lg);     // 二义性错误：含义是 A::A(int) 还是 A::A(double)?
```

在对 f2 的调用中，哪个类型转换都无法精确匹配 long double。然而这两个类型转换都可以使用，只要后面再执行一次生成 long double 的标准类型转换即可。因此，在上面的两个类型转换中哪个都不比另一个更好，调用将产生二义性。

当我们试图用 long 初始化 a2 时也遇到了同样问题，哪个构造函数都无法精确匹配 long 类型。它们在使用构造函数前都要求先将实参进行类型转换：

- 先执行 long 到 double 的标准类型转换，再执行 A(double)
- 先执行 long 到 int 的标准类型转换，再执行 A(int)

编译器没办法区分这两种转换序列的好坏，因此该调用将产生二义性。

调用 f2 及初始化 a2 的过程之所以会产生二义性，根本原因是它们所需的标准类型转换级别一致（参见 6.6.1 节，第 219 页）。当我们使用用户定义的类型转换时，如果转换过程包含标准类型转换，则标准类型转换的级别将决定编译器选择最佳匹配的过程：

```
short s = 42;
// 把 short 提升成 int 优于把 short 转换成 double
A a3(s);                    // 使用 A::A(int)
```

在此例中，把 short 提升成 int 的操作要优于把 short 转换成 double 的操作，因此编译器将使用 A::A(int) 构造函数构造 a3，其中实参是 s（提升后）的值。



当我们使用两个用户定义的类型转换时，如果转换函数之前或之后存在标准类型转换，则标准类型转换将决定最佳匹配到底是哪个。

提示：类型转换与运算符

586

要想正确地设计类的重载运算符、转换构造函数及类型转换函数，必须加倍小心。尤其是当类同时定义了类型转换运算符及重载运算符时特别容易产生二义性。以下的经验规则可能对你有所帮助：

- 不要令两个类执行相同的类型转换：如果 Foo 类有一个接受 Bar 类对象的构造函数，则不要在 Bar 类中再定义转换目标是 Foo 类的类型转换运算符。
- 避免转换目标是内置算术类型的类型转换。特别是当你已经定义了一个转换成算术类型的类型转换时，接下来
  - 不要再定义接受算术类型的重载运算符。如果用户需要使用这样的运算符，则类型转换操作将转换你的类型的对象，然后使用内置的运算符。
  - 不要定义转换到多种算术类型的类型转换。让标准类型转换完成向其他算术类型转换的工作。

一言以蔽之：除了显式地向 bool 类型的转换之外，我们应该尽量避免定义类型转换函数并尽可能地限制那些“显然正确”的非显式构造函数。

重载函数与转换构造函数

当我们调用重载的函数时，从多个类型转换中进行选择将变得更加复杂。如果两个或多个类型转换都提供了同一种可行匹配，则这些类型转换一样好。

举个例子，当几个重载函数的参数分属不同的类类型时，如果这些类恰好定义了同样的转换构造函数，则二义性问题将进一步提升：

```
struct C {
    C(int);
    // 其他成员
```

```
};
struct D {
    D(int);
    // 其他成员
};
void manip(const C&);
void manip(const D&);
manip(10);              // 二义性错误：含义是 manip(C(10))还是 manip(D(10))
```

其中 C 和 D 都包含接受 int 的构造函数，两个构造函数各自匹配 manip 的一个版本。因此调用将具有二义性：它的含义可能是把 int 转换成 C，然后调用 manip 的第一个版本；也可能是把 int 转换成 D，然后调用 manip 的第二个版本。

调用者可以显式地构造正确的类型从而消除二义性：

```
manip(C(10));     // 正确：调用 manip(const C&)
```



如果在调用重载函数时我们需要使用构造函数或者强制类型转换来改变实参的类型，则这通常意味着程序的设计存在不足。

重载函数与用户定义的类型转换

当调用重载函数时，如果两个（或多个）用户定义的类型转换都提供了可行匹配，则
587
我们认为这些类型转换一样好。在这个过程中，我们不会考虑任何可能出现的标准类型转换的级别。只有当重载函数能通过同一个类型转换函数得到匹配时，我们才会考虑其中出现的标准类型转换。

例如当我们调用 manip 时，即使其中一个类定义了需要对实参进行标准类型转换的构造函数，这次调用仍然会具有二义性：

```
struct E {
    E(double);
    // 其他成员
};
void manip2(const C&);
void manip2(const E&);
// 二义性错误：两个不同的用户定义的类型转换都能用在此处
manip2(10);       // 含义是 manip2(C(10))还是 manip2(E(double(10)))
```

在此例中，C 有一个转换源为 int 的类型转换，E 有一个转换源为 double 的类型转换。对于 manip2(10)来说，两个 manip2 函数都是可行的：

- manip2(const C&)是可行的，因为 C 有一个接受 int 的转换构造函数，该构造函数与实参精确匹配。
- manip2(const E&)是可行的，因为 E 有一个接受 double 的转换构造函数，而且为了使用该函数我们可以利用标准类型转换把 int 转换成所需的类型。

因为调用重载函数所请求的用户定义的类型转换不止一个且彼此不同，所以该调用具有二义性。即使其中一个调用需要额外的标准类型转换而另一个调用能精确匹配，编译器也会将该调用标示为错误。



在调用重载函数时，如果需要额外的标准类型转换，则该转换的级别只有当所有可行函数都请求同一个用户定义的类型转换时才有用。如果所需的用户定义的类型转换不止一个，则该调用具有二义性。

14.9.2 节练习

练习 14.50：在初始化 ex1 和 ex2 的过程中，可能用到哪些类类型的转换序列呢？说明初始化是否正确并解释原因。

```
struct LongDouble {
    LongDouble(double = 0.0);
    operator double();
    operator float();
};
LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;
```

练习 14.51：在调用 calc 的过程中，可能用到哪些类型转换序列呢？说明最佳可行函数是如何被选出来的。

```
void calc(int);
void calc(LongDouble);
double dval;
calc(dval);                    // 哪个 calc?
```

14.9.3 函数匹配与重载运算符



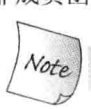
重载的运算符也是重载的函数。因此，通用的函数匹配规则（参见 6.4 节，第 208 页）同样适用于判断在给定的表达式中到底应该使用内置运算符还是重载的运算符。不过当运算符函数出现在表达式中时，候选函数集的规模要比我们使用调用运算符调用函数时更大。如果 a 是一种类类型，则表达式 a *sym* b 可能是

```
a.operatorsym(b); // a 有一个 operatorsym 成员函数
operatorsym(a, b);// operatorsym 是一个普通函数
```

和普通函数调用不同，我们不能通过调用的形式来区分当前调用的是成员函数还是非成员函数。

当我们使用重载运算符作用于类类型的运算对象时，候选函数中包含该运算符的普通非成员版本和内置版本。除此之外，如果左侧运算对象是类类型，则定义在该类中的运算符的重载版本也包含在候选函数内。 588

当我们调用一个命名的函数时，具有该名字的成员函数和非成员函数不会彼此重载，这是因为我们用来调用命名函数的语法形式对于成员函数和非成员函数来说是不相同的。当我们通过类类型的对象（或者该对象的指针及引用）进行函数调用时，只考虑该类的成员函数。而当我们在表达式中使用重载的运算符时，无法判断正在使用的是成员函数还是非成员函数，因此二者都应该在考虑的范围内。



表达式中运算符的候选函数集既应该包括成员函数，也应该包括非成员函数。

举个例子，我们为 SmallInt 类定义一个加法运算符：

```
class SmallInt {
    friend
    SmallInt operator+(const SmallInt&, const SmallInt&);
public:
    SmallInt(int = 0);                    // 转换源为 int 的类型转换
    operator int() const { return val; }  // 转换目标为 int 的类型转换
private:
    std::size_t val;
};
```

589 可以使用这个类将两个 SmallInt 对象相加，但如果我们试图执行混合模式的算术运算，就将遇到二义性的问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2;          // 使用重载的 operator+
int i = s3 + 0;                 // 二义性错误
```

第一条加法语句接受两个 SmallInt 值并执行+运算符的重载版本。第二条加法语句具有二义性：因为我们可以把 0 转换成 SmallInt，然后使用 SmallInt 的+；或者把 s3 转换成 int，然后对于两个 int 执行内置的加法运算。



如果我们对同一个类既提供了转换目标是算术类型的类型转换，也提供了重载的运算符，则将会遇到重载运算符与内置运算符的二义性问题。

14.9.3 节练习

练习 14.52：在下面的加法表达式中分别选用了哪个 operator+？列出候选函数、可行函数及为每个可行函数的实参执行的类型转换：

```
struct LongDouble {
    // 用于演示的成员 operator+；在通常情况下+是个非成员
    LongDouble operator+(const SmallInt&);
    // 其他成员与 14.9.2 节（第 521 页）一致
};
LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
```

练习 14.53：假设我们已经定义了如第 522 页所示的 SmallInt，判断下面的加法表达式是否合法。如果合法，使用了哪个加法运算符？如果不合法，应该怎样修改代码才能使其合法？

```
SmallInt s1;
double d = s1 + 3.14;
```

小结

590

一个重载的运算符必须是某个类的成员或者至少拥有一个类类型的运算对象。重载运算符的运算对象数量、结合律、优先级与对应的用于内置类型的运算符完全一致。当运算符被定义为类的成员时，类对象的隐式 this 指针绑定到第一个运算对象。赋值、下标、函数调用和箭头运算符必须作为类的成员。

如果类重载了函数调用运算符 operator()，则该类的对象被称作“函数对象”。这样的对象常用在标准函数中。lambda 表达式是一种简便的定义函数对象类的方式。

在类中可以定义转换源或转换目的是该类型本身的类型转换，这样的类型转换将自动执行。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的类型转换；而非显式的类型转换运算符则定义了从类类型到其他类型的转换。

术语表

调用形式（call signature） 表示一个可调用对象的接口。在调用形式中包括返回类型以及一个实参类型列表，该列表在一对圆括号内，实参类型之间以逗号分隔。

类类型转换（class-type conversion） 包括由构造函数定义的从其他类型到类类型的转换以及由类型转换运算符定义的从类类型到其他类型的转换。只接受单独一个实参的非显式构造函数定义了从实参类型到类类型的转换；而类型转换运算符则定义了从类类型到某个指定类型的转换。

类型转换运算符（conversion operator） 是类的成员函数，定义了从类类型到其他类型的转换。类型转换运算符必须是它要转换的类的成员，并且通常被定义为常量成员。这类运算符既没有返回类型，也不接受参数。它们返回一个可变为转换运算符类型的值，也就是说，operator int 返回一个 int，operator string 返回一个 string，依此类推。

显式的类型转换运算符（explicit conversion operator） 由关键字 explicit 限定的类型转换运算符。这样的运算符用于条件中的隐式类型转换。

函数对象（function object） 定义了重载调用运算符的对象。在需要使用函数的地方都能使用函数对象。

函数表（function table） 形如 map 或 vector 的容器，容器中所存的值可以被调用。

函数模板（function template） 能够表示任意可调用类型的标准库模板。

重载的运算符（overloaded operator） 重定义了某种内置运算符的含义的函数。重载的运算符函数含有关键字 operator，之后是要定义的符号。重载的运算符必须含有至少一个类类型的运算对象。重载运算符的优先级、结合律、运算对象数量都与其内置版本一致。

用户定义的类型转换（user-defined conversion） 类类型转换的同义词。

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 594 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）。



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，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。

595

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于一个基类的情况将在 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

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：

C++ 11

```
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

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 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



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。

603

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
604
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一个。

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
```

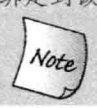
当我们通过一个具有普通类型（非引用非指针）的表达式调用虚函数时，在编译时就会将调用的版本确定下来。例如，如果我们使用 base 调用 net_price，则应该运行 net_price 的哪个版本是显而易见的。我们可以改变 base 表示的对象的值（即内容），但是不会改变该对象的类型。因此，在编译时该调用就会被解析成 Quote 的 net_price。

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。

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，
607 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
609 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，关键问题并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
611 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。

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。
614 - 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被允许：

```
Base *p = &d1;                  // d1 的类型是 Pub_Derv
p = &d2;                        // d2 的类型是 Priv_Derv
```

```
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。

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。

一如往常，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。

620 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哪个版本。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。

621

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 f2()，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

622

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
```



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。

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：

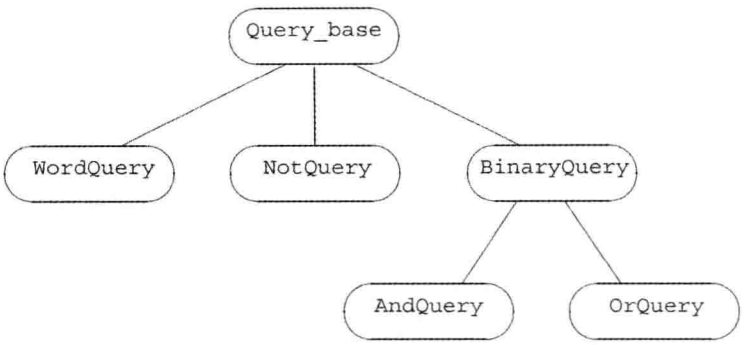


图 15.2：Query_base 继承体系

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。 638

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。

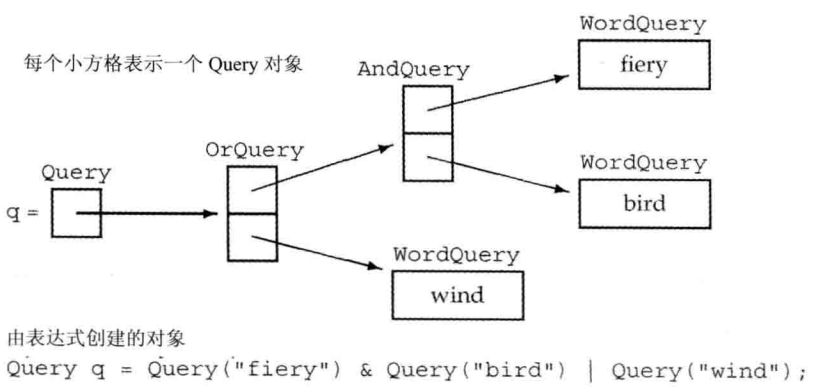


图 15.3：使用 Query 表达式创建的对象

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:                                                    641
    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 {                                    643
    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));
}
```

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。