



15.7.1 虚析构函数

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

如前所述，当我们 `delete` 一个动态分配的对象的指针时将执行析构函数（参见 13.1.3 节，第 445 页）。如果该指针指向继承体系中的某个类型，则有可能出现指针的静态类型与被删除对象的动态类型不符的情况（参见 15.2.2 节，第 530 页）。例如，如果我们 `delete` 一个 `Quote*` 类型的指针，则该指针有可能实际指向了一个 `Bulk_quote` 类型的对象。如果这样的话，编译器就必须清楚它应该执行的是 `Bulk_quote` 的析构函数。和其他函数一样，我们通过在基类中将析构函数定义成虚函数以确保执行正确的析构函数版本：

```
class Quote {
public:
    // 如果我们删除的是一个指向派生类对象的基类指针，则需要虚析构函数
    virtual ~Quote() = default;           // 动态绑定析构函数
};
```

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

```
Quote *itemP = new Quote;           // 静态类型与动态类型一致
delete itemP;                      // 调用 Quote 的析构函数
itemP = new Bulk_quote;             // 静态类型与动态类型不一致
delete itemP;                      // 调用 Bulk_quote 的析构函数
```



WARNING 如果基类的析构函数不是虚函数，则 `delete` 一个指向派生类对象的基类指针将产生未定义的行为。

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

623 > 虚析构函数将阻止合成移动操作

基类需要一个虚析构函数这一事实还会对基类和派生类的定义产生另外一个间接的影响：如果一个类定义了析构函数，即使它通过`=default` 的形式使用了合成的版本，编译器也不会为这个类合成移动操作（参见 13.6.2 节，第 475 页）。

15.7.1 节练习

练习 15.24：哪种类需要虚析构函数？虚析构函数必须执行什么样的操作？



15.7.2 合成拷贝控制与继承

基类或派生类的合成拷贝控制成员的行为与其他合成的构造函数、赋值运算符或析构函数类似：它们对类本身的成员依次进行初始化、赋值或销毁的操作。此外，这些合成的成员还负责使用直接基类中对应的操作对一个对象的直接基类部分进行初始化、赋值或销

毁的操作。例如，

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

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

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

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

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

派生类中删除的拷贝控制与基类的关系

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

- 如果基类中的默认构造函数、拷贝构造函数、拷贝赋值运算符或析构函数是被删除的函数或者不可访问（参见 15.5 节，第 543 页），则派生类中对应的成员将是被删除的，原因是编译器不能使用基类成员来执行派生类对象基类部分的构造、赋值或销毁操作。
- 如果在基类中有一个不可访问或删除掉的析构函数，则派生类中合成的默认和拷贝构造函数将是被删除的，因为编译器无法销毁派生类对象的基类部分。
- 和过去一样，编译器将不会合成一个删除掉的移动操作。当我们使用`=default`请求一个移动操作时，如果基类中的对应操作是删除的或不可访问的，那么派生类中该函数将是被删除的，原因是派生类对象的基类部分不可移动。同样，如果基类的析构函数是删除的或不可访问的，则派生类的移动构造函数也将是被删除的。

举个例子，对于下面的基类 B 来说：

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

C++
11

624

C++
11

```

B(const B&) = delete;
// 其他成员，不含有移动构造函数

};

class D : public B {
    // 没有声明任何构造函数
};

D d;                      // 正确：D 的合成默认构造函数使用 B 的默认构造函数
D d2(d);                  // 错误：D 的合成拷贝构造函数是被删除的
D d3(std::move(d));       // 错误：隐式地使用 D 的被删除的拷贝构造函数

```

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

625 移动操作与继承

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

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

```

class Quote {
public:
    Quote() = default;                                // 对成员依次进行默认初始化
    Quote(const Quote&) = default;                   // 对成员依次拷贝
    Quote(Quote&&) = default;                        // 对成员依次拷贝
    Quote& operator=(const Quote&) = default;        // 拷贝赋值
    Quote& operator=(Quote&&) = default;            // 移动赋值
    virtual ~Quote() = default;
    // 其他成员与之前的版本一致
};

```

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

15.7.2 节练习

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



15.7.3 派生类的拷贝控制成员

如我们在 15.2.2 节(第 531 页)介绍过的，派生类构造函数在其初始化阶段中不但要初始化派生类自己的成员，还负责初始化派生类对象的基类部分。因此，派生类的拷贝和

移动构造函数在拷贝和移动自有成员的同时，也要拷贝和移动基类部分的成员。类似的，派生类赋值运算符也必须为其基类部分的成员赋值。

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



当派生类定义了拷贝或移动操作时，该操作负责拷贝或移动包括基类部分成员在内的整个对象。

<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。

<627

派生类赋值运算符

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

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

<627

```

D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // 为基类部分赋值
    // 按照过去的方式为派生类的成员赋值
    // 酌情处理自赋值及释放已有资源等情况
    return *this;
}

```

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

值得注意的是，无论基类的构造函数或赋值运算符是自定义的版本还是合成的版本，派生类的对应操作都能使用它们。例如，对于 `Base::operator=` 的调用语句将执行 `Base` 的拷贝赋值运算符，至于该运算符是由 `Base` 显式定义的还是由编译器合成的无关紧要。

派生类析构函数

如前所述，在析构函数体执行完成后，对象的成员会被隐式销毁（参见 13.1.3 节，第 445 页）。类似的，对象的基类部分也是隐式销毁的。因此，和构造函数及赋值运算符不同的是，派生类析构函数只负责销毁由派生类自己分配的资源：

```

class D: public Base {
public:
    // Base::~Base 被自动调用执行
    ~D() { /* 该处由用户定义清除派生类成员的操作 */ }
};

```

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

在构造函数和析构函数中调用虚函数

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

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

为了理解上述行为，不妨考虑当基类构造函数调用虚函数的派生类版本时会发生什么情况。这个虚函数可能会访问派生类的成员，毕竟，如果它不需要访问派生类成员的话，则派生类直接使用基类的虚函数版本就可以了。然而，当执行基类构造函数时，它要用到的派生类成员尚未初始化，如果我们允许这样的访问，则程序很可能会崩溃。



如果构造函数或析构函数调用了某个虚函数，则我们应该执行与构造函数或析构函数所属类型相对应的虚函数版本。

15.7.3 节练习

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

15.7.4 继承的构造函数

在 C++11 新标准中，派生类能够重用其直接基类定义的构造函数。尽管如我们所知，这些构造函数并非以常规的方式继承而来，但是为了方便，我们不妨姑且称其为“继承”的。一个类只初始化它的直接基类，出于同样的原因，一个类也只继承其直接基类的构造函数。类不能继承默认、拷贝和移动构造函数。如果派生类没有直接定义这些构造函数，则编译器将为派生类合成它们。

派生类继承基类构造函数的方式是提供一条注明了（直接）基类名的 `using` 声明语句。举个例子，我们可以重新定义 `Bulk_quote` 类（参见 15.4 节，第 541 页），令其继承 `Disc_quote` 类的构造函数：

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // 继承 Disc_quote 的构造函数
    double net_price(std::size_t) const;
};
```

通常情况下，`using` 声明语句只是令某个名字在当前作用域内可见。而当作用于构造函数时，`using` 声明语句将令编译器产生代码。对于基类的每个构造函数，编译器都生成一个与之对应的派生类构造函数。换句话说，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数。

这些编译器生成的构造函数形如：

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

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

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

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

继承的构造函数的特点

和普通成员的 `using` 声明不一样，一个构造函数的 `using` 声明不会改变该构造函数的访问级别。例如，不管 `using` 声明出现在哪儿，基类的私有构造函数在派生类中还是一个私有构造函数；受保护的构造函数和公有构造函数也是同样的规则。

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

C++
11

629

页), 则继承的构造函数也拥有相同的属性。

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

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

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

15.7.4 节练习

练习 15.27: 重新定义你的 Bulk_quote 类, 令其继承构造函数。



15.8 容器与继承

630 >

当我们使用容器存放继承体系中的对象时, 通常必须采取间接存储的方式。因为不允许在容器中保存不同类型的元素, 所以我们不能把具有继承关系的多种类型的对象直接存放在容器当中。

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

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

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

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



当派生类对象被赋值给基类对象时, 其中的派生类部分将被“切掉”, 因此容器和存在继承关系的类型无法兼容。

在容器中放置（智能）指针而非对象

当我们希望在容器中存放具有继承关系的对象时，我们实际上存放的通常是基类的指针（更好的选择是智能指针（参见 12.1 节，第 400 页））。和往常一样，这些指针所指对象的动态类型可能是基类类型，也可能是派生类类型：

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

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

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

<631

15.8 节练习

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

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

15.8.1 编写 Basket 类



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

```
class Basket {
public:
    // Basket 使用合成的默认构造函数和拷贝控制成员
    void add_item(const std::shared_ptr<Quote> &sale)
    { items.insert(sale); }
    // 打印每本书的总价和购物篮中所有书的总价
    double total_receipt(std::ostream&) const;
private:
    // 该函数用于比较 shared_ptr，multiset 成员会用到它
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset 保存多个报价，按照 compare 成员排序
```

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

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

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

632 // multiset 保存多个报价，按照 compare 成员排序
`std::multiset<std::shared_ptr<Quote>, decltype(compare) *>`
`items{compare};`

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

定义 Basket 的成员

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

```
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0; // 保存实时计算出的总价格
    // iter 指向 ISBN 相同的一批元素中的第一个
    // upper_bound 返回一个迭代器，该迭代器指向这批元素的尾后位置
    for (auto iter = items.cbegin();
        iter != items.cend();
        iter = items.upper_bound(*iter)) {
        // 我们知道在当前的 Basket 中至少有一个该关键字的元素
        // 打印该书籍对应的项目
        sum += print_total(os, **iter, items.count(*iter));
    }
    os << "Total Sale: " << sum << endl; // 打印最终的总价格
    return sum;
}
```

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

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

下一位置。因此，我们得到的迭代器或者指向集合的末尾，或者指向下一本书籍。

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

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

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

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

隐藏指针

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

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

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

```
void add_item(const Quote& sale);           // 拷贝给定的对象
void add_item(Quote&& sale);                 // 移动给定的对象
```

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

```
new Quote(sale)
```

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

模拟虚拷贝

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

```
class Quote {
public:
    // 该虚函数返回当前对象的一份动态分配的拷贝
    // 这些成员使用的引用限定符参见 13.6.3 节（第 483 页）
    virtual Quote* clone() const & {return new Quote(*this);}
    virtual Quote* clone() &&
        {return new Quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

634 >

```
class Bulk_quote : public Quote {
    Bulk_quote* clone() const & {return new Bulk_quote(*this);}
    Bulk_quote* clone() &&
        {return new Bulk_quote(std::move(*this));}
    // 其他成员与之前的版本一致
};
```

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

我们可以使用 `clone` 很容易地写出新版本的 `add_item`:

```
class Basket {
public:
    void add_item(const Quote& sale)      // 拷贝给定的对象
        { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale)           // 移动给定的对象
        { items.insert(
            std::shared_ptr<Quote>(std::move(sale).clone())); }
    // 其他成员与之前的版本一致
};
```

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

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

15.8.1 节练习

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

15.9 文本查询程序再探

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

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

```

A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

```

我们的系统将支持如下查询形式。

- 单词查询，用于得到匹配某个给定 string 的所有行：

```

Executing Query for: Daddy
Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"

```

- 逻辑非查询，使用~运算符得到不匹配查询条件的所有行：

```

Executing Query for: ~(Alice)
~(Alice) occurs 9 times
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.

...

```

- 逻辑或查询，使用 | 运算符返回匹配两个条件中任意一个的行：

```

Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,

```

- 逻辑与查询，使用 & 运算符返回匹配全部两个条件的行：

```

Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.

```

此外，我们还希望能够混合使用这些运算符，比如：

```
fiery & bird | wind
```

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

```

Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,

```

636

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

15.9.1 面向对象的解决方案

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

从该类中派生出其他查询是一种可行的方案。

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

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

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

这些类将只包含两个操作：

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

关键概念：继承与组合

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

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

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

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

抽象基类

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

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

析我们将得到如图 15.2 所示的类设计结果：

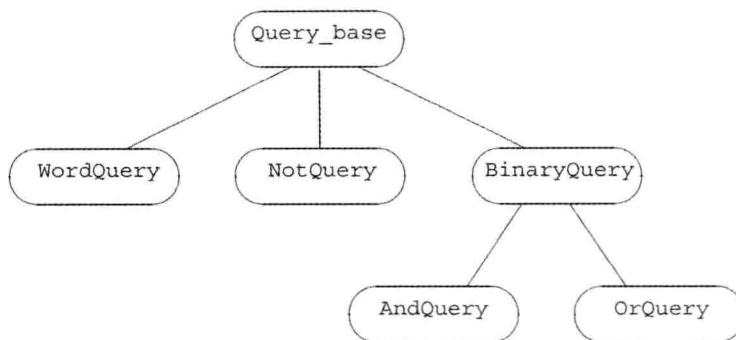


图 15.2: Query_base 继承体系

将层次关系隐藏于接口类中

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

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

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

< 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。

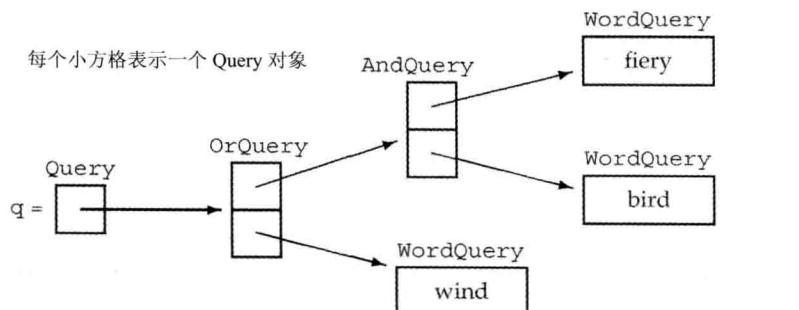


图 15.3: 使用 Query 表达式创建的对象

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

在形成了上述设计思路后，`Query` 类本身就比较简单了：

```
// 这是一个管理 Query_base 继承体系的接口类
class Query {
    // 这些运算符需要访问接受 shared_ptr 的构造函数，而该函数是私有的
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const std::string&); // 构建一个新的 WordQuery
    // 接口函数：调用对应的 Query_base 操作
    QueryResult eval(const TextQuery &t) const
        { return q->eval(t); }
    std::string rep() const { return q->rep(); }
```

```

private:
    Query(std::shared_ptr<Query_base> query): q(query) { }
    std::shared_ptr<Query_base> q;
};

```

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

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

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



Query 的输出运算符

输出运算符可以很好地解释我们的整个查询系统是如何工作的：

```

std::ostream &
operator<<(std::ostream &os, const Query &query)
{
    // Query::rep 通过它的 Query_base 指针对 rep() 进行了虚调用
    return os << query.rep();
}

```

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

```

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

```

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

15.9.2 节练习

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

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

642

15.9.3 派生类

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

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

这样我们就能把该指针绑定到任何我们需要的具体类上。

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

至此我们已经清楚了所有类的设计思路，接下来依次实现它们。

WordQuery 类

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

```
class WordQuery: public Query_base {
    friend class Query; // Query 使用 WordQuery 构造函数
    WordQuery(const std::string &s): query_word(s) { }
    // 具体的类：WordQuery 将定义所有继承而来的纯虚函数
    QueryResult eval(const TextQuery &t) const
        { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word; // 要查找的单词
};
```

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

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

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

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

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

NotQuery 类及~运算符

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

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // 具体的类：NotQuery 将定义所有继承而来的纯虚函数
    std::string rep() const { return "~(" + query.rep() + ")"; }
    QueryResult eval(const TextQuery&) const;
    Query query;
};
inline Query operator~(const Query &operand)
{
    return std::shared_ptr<Query_base>(new NotQuery(operand));
}
```

643

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

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

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

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

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

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

BinaryQuery 类

BinaryQuery 类也是一个抽象基类，它保存操作两个运算对象的查询类型所需的数据：

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(const Query &l, const Query &r, std::string s):
        lhs(l), rhs(r), opSym(s) { }
    // 抽象类: BinaryQuery 不定义 eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                + opSym + " "
                                + rhs.rep() + ")"; }
    Query lhs, rhs;           // 左侧和右侧运算对象
    std::string opSym;        // 运算符的名字
};
```

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

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



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

AndQuery 类、OrQuery 类及相应的运算符

AndQuery 类和 OrQuery 类以及它们的运算符都非常相似：

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "&") { }
    // 具体的类: AndQuery 继承了 rep 并且定义了其他纯虚函数
    QueryResult eval(const TextQuery&) const;
```

```

};

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new AndQuery(lhs, rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
        BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};

inline Query operator|(const Query &lhs, const Query &rhs)
{
    return std::shared_ptr<Query_base>(new OrQuery(lhs, rhs));
}

```

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

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

15.9.3 节练习

< 645

练习 15.34: 针对图 15.3（第 565 页）构建的表达式：

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

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

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

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

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

```

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

```

15.9.4 eval 函数

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

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



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

OrQuery::eval

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

646

```
// 返回运算对象查询结果 set 的并集
QueryResult
OrQuery::eval(const TextQuery& text) const
{
    // 通过 Query 成员 lhs 和 rhs 进行的虚调用
    // 调用 eval 返回每个运算对象的 QueryResult
    auto right = rhs.eval(text), left = lhs.eval(text);
    // 将左侧运算对象的行号拷贝到结果 set 中
    auto ret_lines =
        make_shared<set<line_no>>(left.begin(), left.end());
    // 插入右侧运算对象所得的行号
    ret_lines->insert(right.begin(), right.end());
    // 返回一个新的 QueryResult，它表示 lhs 和 rhs 的并集
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

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

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

AndQuery::eval

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

```
// 返回运算对象查询结果 set 的交集
QueryResult
AndQuery::eval(const TextQuery& text) const
{
```

```

// 通过 Query 运算对象进行的虚调用，以获得运算对象的查询结果 set
auto left = lhs.eval(text), right = rhs.eval(text);
// 保存 left 和 right 交集的 set
auto ret_lines = make_shared<set<line_no>>();
// 将两个范围的交集写入一个目的迭代器中
// 本次调用的目的迭代器向 ret 添加元素
set_intersection(left.begin(), left.end(),
                 right.begin(), right.end(),
                 inserter(*ret_lines, ret_lines->begin()));
return QueryResult(rep(), ret_lines, left.get_file());
}

```

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

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

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

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

NotQuery::eval

`NotQuery` 查找运算对象没有出现的文本行：

```

// 返回运算对象的结果 set 中不存在的行
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // 通过 Query 运算对象对 eval 进行虚调用
    auto result = query.eval(text);
    // 开始时结果 set 为空
    auto ret_lines = make_shared<set<line_no>>();
    // 我们必须在运算对象出现的所有行中进行迭代
    auto beg = result.begin(), end = result.end();
    // 对于输入文件的每一行，如果该行不在 result 当中，则将其添加到 ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // 如果我们还没有处理完 result 的所有行
        // 检查当前行是否存在
        if (beg == end || *beg != n)
            ret_lines->insert(n);      // 如果不在 result 当中，添加这一行
        else if (beg != end)
            ++beg;                  // 否则继续获取 result 的下一行（如果说有的话）
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}

```

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

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

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

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

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

15.9.4 节练习

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

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

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

练习 15.42：从下面的几种改进中选择一种，设计并实现它：

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

小结

< 649

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

在 C++ 语言中，动态绑定只作用于虚函数，并且需要通过指针或引用调用。

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

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

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

术语表

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

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

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

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

派生类（derived class） 从其他类派生而

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

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

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

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

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

< 650

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

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

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

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

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

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

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

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

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

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

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

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

运行时绑定 (run-time binding) 参见“动态绑定”。

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

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

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

第 16 章

模板与泛型编程

内容

16.1 定义模板	578
16.2 模板实参推断	600
16.3 重载与模板	614
16.4 可变参数模板	618
16.5 模板特例化	624
小结	630
术语表	630

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

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

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

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

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

16.1 定义模板

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

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

这两个函数几乎是相同的，唯一的差异是参数的类型，函数体则完全一样。

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



16.1.1 函数模板

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

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

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



在模板定义中，模板参数列表不能为空。

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

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

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

实例化函数模板

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

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

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

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

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

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

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

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

模板类型参数

654

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

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

类型参数前必须使用关键字 `class` 或 `typename`:

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

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

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

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

非类型模板参数

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

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

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

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

当我们调用这个版本的 `compare` 时：

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

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

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

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

在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用

非类型参数，例如，指定数组大小。



非类型模板参数的模板实参必须是常量表达式。

inline 和 constexpr 的函数模板

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

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

// 错误: inline 说明符的位置不正确
inline template <typename T> T min(const T&, const T&);
```

编写类型无关的代码



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

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

通过将函数参数设定为 `const` 的引用，我们保证了函数可以用于不能拷贝的类型。大多 656 数类型，包括内置类型和我们已经用过的标准库类型（除 `unique_ptr` 和 `IO` 类型之外），都是允许拷贝的。但是，不允许拷贝的类类型也是存在的。通过将参数设定为 `const` 的引用，保证了这些类型可以用我们的 `compare` 函数来处理。而且，如果 `compare` 用于处理大对象，这种设计策略还能使函数运行得更快。

你可能认为既使用<运算符又使用>运算符来进行比较操作会更为自然：

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

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

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

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

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



模板程序应该尽量减少对实参类型的要求。



模板编译

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

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

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

657



函数模板和类模板成员函数的定义通常放在头文件中。

关键概念：模板和头文件

模板包含两种名字：

- 那些不依赖于模板参数的名字
- 那些依赖于模板参数的名字

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

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

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

大多数编译错误在实例化期间报告

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

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

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

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

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

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

```
return 0; // 返回 int; 不依赖于 T
```

当编译器处理此模板时，它不能验证 `if` 语句中的条件是否合法。如果传递给 `compare` <658 的实参定义了`<`运算符，则代码就是正确的，否则就是错误的。例如，

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

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



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

16.1.1 节练习

练习 16.1: 给出实例化的定义。

练习 16.2: 编写并测试你自己版本的 `compare` 函数。

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

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

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

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

练习 16.7: 编写一个 `constexpr` 模板，返回给定数组的大小。

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

16.1.2 类模板



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

<659

定义类模板

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

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

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

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

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

660 实例化类模板

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

例如，为了用我们的 `Blob` 模板定义一个类型，必须提供元素类型：

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

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

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

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

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

对我们指定的每一种元素类型，编译器都生成一个不同的类：

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

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



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

在模板作用域中引用模板类型



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

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

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

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

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

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

类模板的成员函数

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

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

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

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

对应的 Blob 的成员应该是这样的：

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

check 和元素访问成员

我们首先定义 check 成员，它检查一个给定的索引：

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

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

下标运算符和 back 函数用模板参数指出返回类型，其他未变：

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

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

`pop_back` 函数与原 StrBlob 的成员几乎相同：

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

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

Blob 构造函数

与其他任何定义在类模板外的成员一样，构造函数的定义要以模板参数开始：

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

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

函数（参见 12.1.1 节，第 405 页），此构造函数分配一个空 vector，并将指向 vector 的指针保存在 data 中。如前所述，我们将类模板自己的类型参数作为 vector 的模板实参来分配 vector。

类似的，接受一个 `initializer_list` 参数的构造函数将其类型参数 `T` 作为 `initializer_list` 参数的元素类型：

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

类似默认构造函数，此构造函数分配一个新的 vector。在本例中，我们用参数 `il` 来初始化此 vector。

为了使用这个构造函数，我们必须传递给它一个 `initializer_list`，其中的元素必须与 Blob 的元素类型兼容：

```
Blob<string> articles = {"a", "an", "the"};
```

这条语句中，构造函数的参数类型为 `initializer_list<string>`。列表中的每个字符串字面量隐式地转换为一个 `string`。

类模板成员函数的实例化

663

默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化。例如，下面代码

```
// 实例化 Blob<int> 和接受 initializer_list<int> 的构造函数
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
// 实例化 Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; // 实例化 Blob<int>::operator[](size_t)
```

实例化了 `Blob<int>` 类和它的三个成员函数：`operator[]`、`size` 和接受 `initializer_list<int>` 的构造函数。

如果一个成员函数没有被使用，则它不会被实例化。成员函数只有在被用到时才进行实例化，这一特性使得即使某种类型不能完全符合模板操作的要求（参见 9.2 节，第 294 页），我们仍然能用该类型实例化类。



默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被实例化。

在类代码内简化模板类名的使用

当我们使用一个类模板类型时必须提供模板实参，但这一规则有一个例外。在类模板自己的作用域中，我们可以直接使用模板名而不提供实参：

```
// 若试图访问一个不存在的元素，BlobPtr 抛出一个异常
template <typename T> class BlobPtr {
public:
    BlobPtr(): curr(0) {}
    BlobPtr(Blob<T> &a, size_t sz = 0):
        wptr(a.data), curr(sz) {}
    T& operator*() const
    { auto p = check(curr, "dereference past end");
        return (*p)[curr]; // (*p) 为本对象指向的 vector
```

```

    }
    // 递增和递减
    BlobPtr& operator++(); // 前置运算符
    BlobPtr& operator--();
private:
    // 若检查成功, check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr, 表示底层 vector 可能被销毁
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr; // 数组中的当前位置
};

```

细心的读者可能已经注意到, BlobPtr 的前置递增和递减成员返回 BlobPtr&, 而不是 BlobPtr<T>&。当我们处于一个类模板的作用域中时, 编译器处理模板自身引用时就好像我们已经提供了与模板参数匹配的实参一样。即, 就好像我们这样编写代码一样:

```

BlobPtr<T>& operator++();
BlobPtr<T>& operator--();

```

在类模板外使用类模板名

当我们在类模板外定义其成员时, 必须记住, 我们并不在类的作用域中, 直到遇到类名才表示进入类的作用域 (参见 7.4 节, 第 253 页):

```

// 后置: 递增/递减对象但返回原值
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    // 此处无须检查; 调用前置递增时会进行检查
    BlobPtr ret = *this; // 保存当前值
    ++*this; // 推进一个元素; 前置++检查递增是否合法
    return ret; // 返回保存的状态
}

```

由于返回类型位于类的作用域之外, 我们必须指出返回类型是一个实例化的 BlobPtr, 它所用类型与类实例化所用类型一致。在函数体内, 我们已经进入类的作用域, 因此在定义 ret 时无须重复模板实参。如果不提供模板实参, 则编译器将假定我们使用的类型与成员实例化所用类型一致。因此, ret 的定义与如下代码等价:

```

BlobPtr<T> ret = *this;

```



在一个类模板的作用域内, 我们可以直接使用模板名而不必指定模板实参。

类模板和友元

当一个类包含一个友元声明 (参见 7.2.1 节, 第 241 页) 时, 类与友元各自是否是模板是相互无关的。如果一个类模板包含一个非模板友元, 则友元被授权可以访问所有模板实例。如果友元自身是模板, 类可以授权给所有友元模板实例, 也可以只授权给特定实例。

一对一封好关系

类模板与另一个 (类或函数) 模板间友好关系的最常见的形式是建立对应实例及其友元间的友好关系。例如, 我们的 Blob 类应该将 BlobPtr 类和一个模板版本的 Blob 相

等运算符（最初是在 14.3.1 节（第 498 页）练习中为 StrBlob 定义的）定义为友元。

为了引用（类或函数）模板的一个特定实例，我们必须首先声明模板自身。一个模板声明包括模板参数列表：

```
// 前置声明，在 Blob 中声明友元所需要的
template <typename> class BlobPtr;
template <typename> class Blob; // 运算符==中的参数所需要的
template <typename T>
    bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {
    // 每个 Blob 实例将访问权限授予用相同类型实例化的 BlobPtr 和相等运算符
    friend class BlobPtr<T>;
    friend bool operator==(const Blob<T>&, const Blob<T>&);
    // 其他成员定义，与 12.1.1（第 405 页）相同
};
```

< 665

我们首先将 Blob、BlobPtr 和 operator== 声明为模板。这些声明是 operator== 函数的参数声明以及 Blob 中的友元声明所需要的。

友元的声明用 Blob 的模板形参作为它们自己的模板实参。因此，友好关系被限定在用相同类型实例化的 Blob 与 BlobPtr 相等运算符之间：

```
Blob<char> ca; // BlobPtr<char>和 operator==<char>都是本对象的友元
Blob<int> ia; // BlobPtr<int>和 operator==<int>都是本对象的友元
```

BlobPtr<char> 的成员可以访问 ca（或任何其他 Blob<char> 对象）的非 public 部分，但 ca 对 ia（或任何其他 Blob<int> 对象）或 Blob 的任何其他实例都没有特殊访问权限。

通用和特定的模板友好关系

一个类也可以将另一个模板的每个实例都声明为自己的友元，或者限定特定的实例为友元：

```
// 前置声明，在将模板的一个特定实例声明为友元时要用到
template <typename T> class Pal;
class C { // C 是一个普通的非模板类
    friend class Pal<C>; // 用类 C 实例化的 Pal 是 C 的一个友元
    // Pal2 的所有实例都是 C 的友元；这种情况无须前置声明
    template <typename T> friend class Pal2;
};

template <typename T> class C2 { // C2 本身是一个类模板
    // C2 的每个实例将相同实例化的 Pal 声明为友元
    friend class Pal<T>; // Pal 的模板声明必须在作用域之内
    // Pal2 的所有实例都是 C2 的每个实例的友元，不需要前置声明
    template <typename X> friend class Pal2;
    // Pal3 是一个非模板类，它是 C2 所有实例的友元
    friend class Pal3; // 不需要 Pal3 的前置声明
};
```

为了让所有实例成为友元，友元声明中必须使用与类模板本身不同的模板参数。

666 令模板自己的类型参数成为友元

C++ 11 在新标准中，我们可以将模板类型参数声明为友元：

```
template <typename Type> class Bar {
    friend Type; // 将访问权限授予用来实例化 Bar 的类型
    //...
};
```

此处我们将用来实例化 Bar 的类型声明为友元。因此，对于某个类型名 Foo，Foo 将成为 Bar<Foo>的友元，Sales_data 将成为 Bar<Sales_data>的友元，依此类推。

值得注意的是，虽然友元通常来说应该是一个类或是一个函数，但我们完全可以用一个内置类型来实例化 Bar。这种与内置类型的友好关系是允许的，以便我们能用内置类型来实例化 Bar 这样的类。

模板类型别名

类模板的一个实例定义了一个类类型，与任何其他类类型一样，我们可以定义一个 `typedef`（参见 2.5.1 节，第 60 页）来引用实例化的类：

```
typedef Blob<string> StrBlob;
```

这条 `typedef` 语句允许我们运行在 12.1.1 节（第 405 页）中编写的代码，而使用的却是用 `string` 实例化的模板版本的 `Blob`。由于模板不是一个类型，我们不能定义一个 `typedef` 引用一个模板。即，无法定义一个 `typedef` 引用 `Blob<T>`。

C++ 11 但是，新标准允许我们为类模板定义一个类型别名：

```
template<typename T> using twin = pair<T, T>;
twin<string> authors; // authors 是一个 pair<string, string>
```

在这段代码中，我们将 `twin` 定义为成员类型相同的 `pair` 的别名。这样，`twin` 的用户只需指定一次类型。

一个模板类型别名是一族类的别名：

```
twin<int> win_loss; // win_loss 是一个 pair<int, int>
twin<double> area; // area 是一个 pair<double, double>
```

就像使用类模板一样，当我们使用 `twin` 时，需要指出希望使用哪种特定类型的 `twin`。

当我们定义一个模板类型别名时，可以固定一个或多个模板参数：

```
template <typename T> using partNo = pair<T, unsigned>;
partNo<string> books; // books 是一个 pair<string, unsigned>
partNo<Vehicle> cars; // cars 是一个 pair<Vehicle, unsigned>
partNo<Student> kids; // kids 是一个 pair<Student, unsigned>
```

这段代码中我们将 `partNo` 定义为一族类型的别名，这族类型是 `second` 成员为 `unsigned` 的 `pair`。`partNo` 的用户需要指出 `pair` 的 `first` 成员的类型，但不能指定 `second` 成员的类型。

667 类模板的 static 成员

与任何其他类相同，类模板可以声明 `static` 成员（参见 7.6 节，第 269 页）：

```
template <typename T> class Foo {
public:
```

```
static std::size_t count() { return ctr; }
// 其他接口成员
private:
    static std::size_t ctr;
    // 其他实现成员
};
```

在这段代码中，`Foo` 是一个类模板，它有一个名为 `count` 的 `public static` 成员函数和一个名为 `ctr` 的 `private static` 数据成员。每个 `Foo` 的实例都有其自己的 `static` 成员实例。即，对任意给定类型 `X`，都有一个 `Foo<X>::ctr` 和一个 `Foo<X>::count` 成员。所有 `Foo<X>` 类型的对象共享相同的 `ctr` 对象和 `count` 函数。例如，

```
// 实例化 static 成员 Foo<string>::ctr 和 Foo<string>::count
Foo<string> fs;
// 所有三个对象共享相同的 Foo<int>::ctr 和 Foo<int>::count 成员
Foo<int> fi, fi2, fi3;
```

与任何其他 `static` 数据成员相同，模板类的每个 `static` 数据成员必须有且仅有一个定义。但是，类模板的每个实例都有一个独有的 `static` 对象。因此，与定义模板的成员函数类似，我们将 `static` 数据成员也定义为模板：

```
template <typename T>
size_t Foo<T>::ctr = 0; // 定义并初始化 ctr
```

与类模板的其他任何成员类似，定义的开始部分是模板参数列表，随后是我们定义的成员的类型和名字。与往常一样，成员名包括成员的类名，对于从模板生成的类来说，类名包括模板实参。因此，当使用一个特定的模板实参类型实例化 `Foo` 时，将会为该类类型实例化一个独立的 `ctr`，并将其初始化为 0。

与非模板类的静态成员相同，我们可以通过类类型对象来访问一个类模板的 `static` 成员，也可以使用作用域运算符直接访问成员。当然，为了通过类来直接访问 `static` 成员，我们必须引用一个特定的实例：

```
Foo<int> fi; // 实例化 Foo<int> 类和 static 数据成员 ctr
auto ct = Foo<int>::count(); // 实例化 Foo<int>::count
ct = fi.count(); // 使用 Foo<int>::count
ct = Foo::count(); // 错误：使用哪个模板实例的 count？
```

类似任何其他成员函数，一个 `static` 成员函数只有在使用时才会实例化。

16.1.2 节练习

668

练习 16.9：什么是函数模板？什么是类模板？

练习 16.10：当一个类模板被实例化时，会发生什么？

练习 16.11：下面 `List` 的定义是错误的。应如何修正它？

```
template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
```

```

    void insert(ListItem *ptr, elemType value);
private:
    ListItem *front, *end;
};

```

练习 16.12: 编写你自己版本的 Blob 和 BlobPtr 模板，包含书中未定义的多个 const 成员。

练习 16.13: 解释你为 BlobPtr 的相等和关系运算符选择哪种类型的友好关系？

练习 16.14: 编写 Screen 类模板，用非类型参数定义 Screen 的高和宽。

练习 16.15: 为你的 Screen 模板实现输入和输出运算符。Screen 类需要哪些友元（如果需要的话）来令输入和输出运算符正确工作？解释每个友元声明（如果有的话）为什么是必要的。

练习 16.16: 将 StrVec 类（参见 13.5 节，第 465 页）重写为模板，命名为 Vec。



16.1.3 模板参数

类似函数参数的名字，一个模板参数的名字也没有什么内在含义。我们通常将类型参数命名为 T，但实际上我们可以使用任何名字：

```

template <typename Foo> Foo calc(const Foo& a, const Foo& b)
{
    Foo tmp = a; // tmp 的类型与参数和返回类型一样
    //...
    return tmp; // 返回类型和参数类型一样
}

```

模板参数与作用域

模板参数遵循普通的作用域规则。一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前。与任何其他名字一样，模板参数会隐藏外层作用域中声明的相同名字。但是，与大多数其他上下文不同，在模板内不能重用模板参数名：

```

typedef double A;
template <typename A, typename B> void f(A a, B b)
{
    A tmp = a; // tmp 的类型为模板参数 A 的类型，而非 double
    double B; // 错误：重声明模板参数 B
}

```

正常的名字隐藏规则决定了 A 的 `typedef` 被类型参数 A 隐藏。因此，tmp 不是一个 `double`，其类型是使用 `f` 时绑定到类型参数 A 的类型。由于我们不能重用模板参数名，声明名字为 B 的变量是错误的。

由于参数名不能重用，所以一个模板参数名在一个特定模板参数列表中只能出现一次：

```

// 错误：非法重用模板参数名 V
template <typename V, typename V> //...

```

模板声明

模板声明必须包含模板参数：

```
// 声明但不定义 compare 和 Blob
template <typename T> int compare(const T&, const T&);
template <typename T> class Blob;
```

与函数参数相同，声明中的模板参数的名字不必与定义中相同：

```
// 3个 calc 都指向相同的函数模板
template <typename T> T calc(const T&, const T&); // 声明
template <typename U> U calc(const U&, const U&); // 声明
// 模板的定义
template <typename Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

当然，一个给定模板的每个声明和定义必须有相同数量和种类（即，类型或非类型）的参数。



一个特定文件所需的所有模板的声明通常一起放置在文件开始位置，出现于任何使用这些模板的代码之前，原因我们将在 16.3 节（第 617 页）中解释。

使用类的类型成员

回忆一下，我们用作用域运算符 (::) 来访问 static 成员和类型成员（参见 7.4 节，第 253 页和 7.6 节，第 269 页）。在普通（非模板）代码中，编译器掌握类的定义。因此，它知道通过作用域运算符访问的名字是类型还是 static 成员。例如，如果我们写下 string::size_type，编译器有 string 的定义，从而知道 size_type 是一个类型。

670

但对于模板代码就存在困难。例如，假定 T 是一个模板类型参数，当编译器遇到类似 T::mem 这样的代码时，它不会知道 mem 是一个类型成员还是一个 static 数据成员，直至实例化时才会知道。但是，为了处理模板，编译器必须知道名字是否表示一个类型。例如，假定 T 是一个类型参数的名字，当编译器遇到如下形式的语句时：

```
T::size_type * p;
```

它需要知道我们是正在定义一个名为 p 的变量还是将一个名为 size_type 的 static 数据成员与名为 p 的变量相乘。

默认情况下，C++ 语言假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显式告诉编译器该名字是一个类型。我们通过使用关键字 typename 来实现这一点：

```
template <typename T>
typename T::value_type top(const T& c)
{
    if (!c.empty())
        return c.back();
    else
        return typename T::value_type();
}
```

我们的 top 函数期待一个容器类型的实参，它使用 typename 指明其返回类型并在 c 中没有元素时生成一个值初始化的元素（参见 7.5.3 节，第 262 页）返回给调用者。



当我们希望通知编译器一个名字表示类型时，必须使用关键字 typename，而不能使用 class。

默认模板实参

C++
11

就像我们能为函数参数提供默认实参一样（参见 6.5.1 节，第 211 页），我们也可以提供默认模板实参（default template argument）。在新标准中，我们可以为函数和类模板提供默认实参。而更早的 C++ 标准只允许为类模板提供默认实参。

例如，我们重写 `compare`，默认使用标准库的 `less` 函数对象模板（参见 14.8.2 节，第 509 页）：

```
// compare 有一个默认模板实参 less<T> 和一个默认函数实参 F()
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

671 在这段代码中，我们为模板添加了第二个类型参数，名为 `F`，表示可调用对象（参见 10.3.2 节，第 346 页）的类型；并定义了一个新的函数参数 `f`，绑定到一个可调用对象上。

我们为此模板参数提供了默认实参，并为其对应的函数参数也提供了默认实参。默认模板实参指出 `compare` 将使用标准库的 `less` 函数对象类，它是使用与 `compare` 一样的类型参数实例化的。默认函数实参指出 `f` 将是类型 `F` 的一个默认初始化的对象。

当用户调用这个版本的 `compare` 时，可以提供自己的比较操作，但这并不是必需的：

```
bool i = compare(0, 42); // 使用 less; i 为 -1
// 结果依赖于 item1 和 item2 中的 isbn
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
```

第一个调用使用默认函数实参，即，类型 `less<T>` 的一个默认初始化对象。在此调用中，`T` 为 `int`，因此可调用对象的类型为 `less<int>`。`compare` 的这个实例化版本将使用 `less<int>` 进行比较操作。

在第二个调用中，我们传递给 `compare` 三个实参：`compareIsbn`（参见 11.2.2 节，第 379 页）和两个 `Sales_data` 类型的对象。当传递给 `compare` 三个实参时，第三个实参的类型必须是一个可调用对象，该可调用对象的返回类型必须能转换为 `bool` 值，且接受的实参类型必须与 `compare` 的前两个实参的类型兼容。与往常一样，模板参数的类型从它们对应的函数实参推断而来。在此调用中，`T` 的类型被推断为 `Sales_data`，`F` 被推断为 `compareIsbn` 的类型。

与函数默认实参一样，对于一个模板参数，只有当它右侧的所有参数都有默认实参时，它才可以有默认实参。

模板默认实参与类模板

无论何时使用一个类模板，我们都必须在模板名之后接上尖括号。尖括号指出类必须从一个模板实例化而来。特别是，如果一个类模板为所有模板参数都提供了默认实参，且我们希望使用这些默认实参，就必须在模板名之后跟一个空尖括号对：

```
template <class T = int> class Numbers { // T 默认为 int
public:
    Numbers(T v = 0) : val(v) {}
```

```

    // 对数值的各种操作
private:
    T val;
};

Numbers<long double> lots_of_precision;
Numbers<> average_precision; // 空<>表示我们希望使用默认类型

```

此例中我们实例化了两个 Numbers 版本：average_precision 是用 int 替代 T 实例化得到的；lots_of_precision 是用 long double 替代 T 实例化而得到的。

16.1.3 节练习

<672

练习 16.17：声明为 typename 的类型参数和声明为 class 的类型参数有什么不同（如果有的话）？什么时候必须使用 typename？

练习 16.18：解释下面每个函数模板声明并指出它们是否非法。更正你发现的每个错误。

- (a) template <typename T, U, typename V> void f1(T, U, V);
- (b) template <typename T> T f2(int &T);
- (c) inline template <typename T> T foo(T, unsigned int*);
- (d) template <typename T> f4(T, T);
- (e) typedef char Ctype;
template <typename Ctype> Ctype f5(Ctype a);

练习 16.19：编写函数，接受一个容器的引用，打印容器中的元素。使用容器的 size_type 和 size 成员来控制打印元素的循环。

练习 16.20：重写上一题的函数，使用 begin 和 end 返回的迭代器来控制循环。

16.1.4 成员模板

一个类（无论是普通类还是类模板）可以包含本身是模板的成员函数。这种成员被称为成员模板（member template）。成员模板不能是虚函数。

普通（非模板）类的成员模板

作为普通类包含成员模板的例子，我们定义一个类，类似 unique_ptr 所使用的默认删除器类型（参见 12.1.5 节，第 418 页）。类似默认删除器，我们的类将包含一个重载的函数调用运算符（参见 14.8 节，第 506 页），它接受一个指针并对此指针执行 delete。与默认删除器不同，我们的类还将在删除器被执行时打印一条信息。由于希望删除器适用于任何类型，所以我们将调用运算符定义为一个模板：

```

// 函数对象类，对给定指针执行 delete
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr): os(s) { }
    // 与任何函数模板相同，T 的类型由编译器推断
    template <typename T> void operator()(T *p) const
        { os << "deleting unique_ptr" << std::endl; delete p; }
private:
    std::ostream &os;
};

```

673 与任何其他模板相同，成员模板也是以模板参数列表开始的。每个 `DebugDelete` 对象都有一个 `ostream` 成员，用于写入数据；还包含一个自身是模板的成员函数。我们可以用这个类代替 `delete`：

```
double* p = new double;
DebugDelete d; // 可像 delete 表达式一样使用的对象
d(p); // 调用 DebugDelete::operator()(double*)，释放 p
int* ip = new int;
// 在一个临时 DebugDelete 对象上调用 operator()(int*)
DebugDelete()(ip);
```

由于调用一个 `DebugDelete` 对象会 `delete` 其给定的指针，我们也可以将 `DebugDelete` 用作 `unique_ptr` 的删除器。为了重载 `unique_ptr` 的删除器，我们在尖括号内给出删除器类型，并提供一个这种类型的对象给 `unique_ptr` 的构造函数（参见 12.1.5 节，第 418 页）：

```
// 销毁 p 指向的对象
// 实例化 DebugDelete::operator()(int*)(int *)
unique_ptr<int, DebugDelete> p(new int, DebugDelete());
// 销毁 sp 指向的对象
// 实例化 DebugDelete::operator()(string*)(string*)
unique_ptr<string, DebugDelete> sp(new string, DebugDelete());
```

在本例中，我们声明 `p` 的删除器的类型为 `DebugDelete`，并在 `p` 的构造函数中提供了该类型的一个未命名对象。

`unique_ptr` 的析构函数会调用 `DebugDelete` 的调用运算符。因此，无论何时 `unique_ptr` 的析构函数实例化时，`DebugDelete` 的调用运算符都会实例化：因此，上述定义会这样实例化。

```
// DebugDelete 的成员模板实例化样例
void DebugDelete::operator()(int *p) const { delete p; }
void DebugDelete::operator()(string *p) const { delete p; }
```

类模板的成员模板

对于类模板，我们也可以为其定义成员模板。在此情况下，类和成员各自有自己的、独立的模板参数。

例如，我们将为 `Blob` 类定义一个构造函数，它接受两个迭代器，表示要拷贝的元素范围。由于我们希望支持不同类型序列的迭代器，因此将构造函数定义为模板：

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
    //...
};
```

此构造函数有自己的模板类型参数 `It`，作为它的两个函数参数的类型。

与类模板的普通函数成员不同，成员模板是函数模板。当我们在类模板外定义一个成员模板时，必须同时为类模板和成员模板提供模板参数列表。类模板的参数列表在前，后跟成员自己的模板参数列表：

```
template <typename T> // 类的类型参数
template <typename It> // 构造函数的类型参数
Blob<T>::Blob(It b, It e);
```

```
data(std::make_shared<std::vector<T>>(b, e)) { }
```

在此例中，我们定义了一个类模板的成员，类模板有一个模板类型参数，命名为 `T`。而成员自身是一个函数模板，它有一个名为 `It` 的类型参数。

实例化与成员模板

为了实例化一个类模板的成员模板，我们必须同时提供类和函数模板的实参。与往常一样，我们在哪个对象上调用成员模板，编译器就根据该对象的类型来推断类模板参数的实参。与普通函数模板相同，编译器通常根据传递给成员模板的函数实参来推断它的模板实参（参见 16.1.1 节，第 579 页）：

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
// 实例化 Blob<int>类及其接受两个 int*参数的构造函数
Blob<int> a1(begin(ia), end(ia));
// 实例化 Blob<int>类的接受两个 vector<long>::iterator 的构造函数
Blob<int> a2(vi.begin(), vi.end());
// 实例化 Blob<string>及其接受两个 list<const char*>::iterator 参数的构造函数
Blob<string> a3(w.begin(), w.end());
```

当我们定义 `a1` 时，显式地指出编译器应该实例化一个 `int` 版本的 `Blob`。构造函数自己的类型参数则通过 `begin(ia)` 和 `end(ia)` 的类型来推断，结果为 `int*`。因此，`a1` 的定义实例化了如下版本：

```
Blob<int>::Blob(int*, int*);
```

`a2` 的定义使用了已经实例化了的 `Blob<int>` 类，并用 `vector<short>::iterator` 替换 `It` 来实例化构造函数。`a3` 的定义（显式地）实例化了一个 `string` 版本的 `Blob`，并（隐式地）实例化了该类的成员模板构造函数，其模板参数被绑定到 `list<const char*>`。

16.1.4 节练习

675

练习 16.21：编写你自己的 `DebugDelete` 版本。

练习 16.22：修改 12.3 节（第 430 页）中你的 `TextQuery` 程序，令 `shared_ptr` 成员使用 `DebugDelete` 作为它们的删除器（参见 12.1.4 节，第 415 页）。

练习 16.23：预测在你的查询主程序中何时会执行调用运算符。如果你的预测和实际不符，确认你理解了原因。

练习 16.24：为你的 `Blob` 模板添加一个构造函数，它接受两个迭代器。

16.1.5 控制实例化



当模板被使用时才会进行实例化（参见 16.1.1 节，第 582 页）这一特性意味着，相同的实例可能出现在多个对象文件中。当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例。

C++
11

在大系统中，在多个文件中实例化相同模板的额外开销可能非常严重。在新标准中，我们可以通过显式实例化（`explicit instantiation`）来避免这种开销。一个显式实例化有如下

形式：

```
extern template declaration;      // 实例化声明
template declaration;           // 实例化定义
```

declaration 是一个类或函数声明，其中所有模板参数已被替换为模板实参。例如，

```
// 实例化声明与定义
extern template class Blob<string>;           // 声明
template int compare(const int&, const int&); // 定义
```

当编译器遇到 `extern` 模板声明时，它不会在本文件中生成实例化代码。将一个实例化声明为 `extern` 就表示承诺在程序其他位置有该实例化的一个非 `extern` 声明（定义）。对于一个给定的实例化版本，可能有多个 `extern` 声明，但必须只有一个定义。

由于编译器在使用一个模板时自动对其实例化，因此 `extern` 声明必须出现在任何使用此实例化版本的代码之前：

```
// Application.cc
// 这些模板类型必须在程序其他位置进行实例化
extern template class Blob<string>;
extern template int compare(const int&, const int&);
Blob<string> sal, sa2; // 实例化会出现在其他位置
// Blob<int>及其接受 initializer_list 的构造函数在本文件中实例化
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); // 拷贝构造函数在本文件中实例化
int i = compare(a1[0], a2[0]); // 实例化出现在其他位置
```

676 文件 `Application.o` 将包含 `Blob<int>` 的实例及其接受 `initializer_list` 参数的构造函数和拷贝构造函数的实例。而 `compare<int>` 函数和 `Blob<string>` 类将不在本文件中进行实例化。这些模板的定义必须出现在程序的其他文件中：

```
// templateBuild.cc
// 实例化文件必须为每个在其他文件中声明为 extern 的类型和函数提供一个（非 extern）
// 的定义
template int compare(const int&, const int&);
template class Blob<string>; // 实例化类模板的所有成员
```

当编译器遇到一个实例化定义（与声明相对）时，它为其生成代码。因此，文件 `templateBuild.o` 将会包含 `compare` 的 `int` 实例化版本的定义和 `Blob<string>` 类的定义。当我们编译此应用程序时，必须将 `templateBuild.o` 和 `Application.o` 链接到一起。



对每个实例化声明，在程序中某个位置必须有其显式的实例化定义。

实例化定义会实例化所有成员

一个类模板的实例化定义会实例化该模板的所有成员，包括内联的成员函数。当编译器遇到一个实例化定义时，它不了解程序使用哪些成员函数。因此，与处理类模板的普通实例化不同，编译器会实例化该类的所有成员。即使我们不使用某个成员，它也会被实例化。因此，我们用来显式实例化一个类模板的类型，必须能用于模板的所有成员。

Note

在一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数。

16.1.5 节练习

练习 16.25：解释下面这些声明的含义：

```
extern template class vector<string>;
template class vector<Sales_data>;
```

练习 16.26：假设 `NoDefault` 是一个没有默认构造函数的类，我们可以显式实例化 `vector<NoDefault>` 吗？如果不可以，解释为什么。

练习 16.27：对下面每条带标签的语句，解释发生了什么样的实例化（如果有的话）。如果一个模板被实例化，解释为什么；如果未实例化，解释为什么没有。

```
template <typename T> class Stack { };
void f1(Stack<char>); // (a)
class Exercise {
    Stack<double> &rsd; // (b)
    Stack<int> si; // (c)
};
int main() {
    Stack<char> *sc; // (d)
    f1(*sc); // (e)
    int iObj = sizeof(Stack< string >); // (f)
}
```

16.1.6 效率与灵活性

对模板设计者所面对的设计选择，标准库智能指针类型（参见 12.1 节，第 400 页）给出了一个很好的展示。

`shared_ptr` 和 `unique_ptr` 之间的明显不同是它们管理所保存的指针的策略——前者给予我们共享指针所有权的能力；后者则独占指针。这一差异对两个类的功能来说是至关重要的。

这两个类的另一个差异是它们允许用户重载默认删除器的方式。我们可以很容易地重载一个 `shared_ptr` 的删除器，只要在创建或 `reset` 指针时传递给它一个可调用对象即可。与之相反，删除器的类型是一个 `unique_ptr` 对象的类型的一部分。用户必须在定义 `unique_ptr` 时以显式模板实参的形式提供删除器的类型。因此，对于 `unique_ptr` 的用户来说，提供自己的删除器就更为复杂。

如何处理删除器的差异实际上就是这两个类功能的差异。但是，如我们将要看到的，这一实现策略上的差异可能对性能有重要影响。

在运行时绑定删除器

虽然我们不知道标准库类型是如何实现的，但可以推断出，`shared_ptr` 必须能直接访问其删除器。即，删除器必须保存为一个指针或一个封装了指针的类（如 `function`，参见 14.8.3 节，第 512 页）。

我们可以确定 `shared_ptr` 不是将删除器直接保存为一个成员，因为删除器的类型

直到运行时才会知道。实际上，在一个 `shared_ptr` 的生存期中，我们可以随时改变其删除器的类型。我们可以使用一种类型的删除器构造一个 `shared_ptr`，随后使用 `reset` 赋予此 `shared_ptr` 另一种类型的删除器。通常，类成员的类型在运行时是不能改变的。因此，不能直接保存删除器。

为了考察删除器是如何正确工作的，让我们假定 `shared_ptr` 将它管理的指针保存在一个成员 `p` 中，且删除器是通过一个名为 `del` 的成员来访问的。则 `shared_ptr` 的析构函数必须包含类似下面这样的语句：

```
// del 的值只有在运行时才知道；通过一个指针来调用它
del ? del(p) : delete p; // del(p) 需要运行时跳转到 del 的地址
```

678> 由于删除器是间接保存的，调用 `del(p)` 需要一次运行时的跳转操作，转到 `del` 中保存的地址来执行对应的代码。

在编译时绑定删除器

现在，让我们来考察 `unique_ptr` 可能的工作方式。在这个类中，删除器的类型是类类型的一部分。即，`unique_ptr` 有两个模板参数，一个表示它所管理的指针，另一个表示删除器的类型。由于删除器的类型是 `unique_ptr` 类型的一部分，因此删除器成员的类型在编译时是知道的，从而删除器可以直接保存在 `unique_ptr` 对象中。

`unique_ptr` 的析构函数与 `shared_ptr` 的析构函数类似，也是对其保存的指针调用用户提供的删除器或执行 `delete`：

```
// del 在编译时绑定；直接调用实例化的删除器
del(p); // 无运行时额外开销
```

`del` 的类型或者是默认删除器类型，或者是用户提供的类型。到底是哪种情况没有关系，应该执行的代码在编译时肯定会知道。实际上，如果删除器是类似 `DebugDelete`（参见 16.1.4 节，第 595 页）之类的东西，这个调用甚至可能被编译为内联形式。

通过在编译时绑定删除器，`unique_ptr` 避免了间接调用删除器的运行时开销。通过在运行时绑定删除器，`shared_ptr` 使用用户重载删除器更为方便。

16.1.6 节练习

练习 16.28：编写你自己版本的 `shared_ptr` 和 `unique_ptr`。

练习 16.29：修改你的 `Blob` 类，用你自己的 `shared_ptr` 代替标准库中的版本。

练习 16.30：重新运行你的一些程序，验证你的 `shared_ptr` 类和修改后的 `Blob` 类。（注意：实现 `weak_ptr` 类型超出了本书范围，因此你不能将 `BlobPtr` 类与你修改后的 `Blob` 一起使用。）

练习 16.31：如果我们将 `DebugDelete` 与 `unique_ptr` 一起使用，解释编译器将删除器处理为内联形式的可能方式。

16.2 模板实参推断

我们已经看到，对于函数模板，编译器利用调用中的函数实参来确定其模板参数。从函数实参来确定模板实参的过程被称为模板实参推断（template argument deduction）。在模

板实参推断过程中，编译器使用函数调用中的实参类型来寻找模板实参，用这些模板实参生成的函数版本与给定的函数调用最为匹配。

16.2.1 类型转换与模板类型参数



与非模板函数一样，我们在一次调用中传递给函数模板的实参被用来初始化函数的形参。如果一个函数形参的类型使用了模板类型参数，那么它采用特殊的初始化规则。只有很有限的几种类型转换会自动地应用于这些实参。编译器通常不是对实参进行类型转换，而是生成一个新的模板实例。

与往常一样，顶层 `const`（参见 2.4.3 节，第 57 页）无论是在形参中还是在实参中，都会被忽略。在其他类型转换中，能在调用中应用于函数模板的包括如下两项。

- `const` 转换：可以将一个非 `const` 对象的引用（或指针）传递给一个 `const` 的引用（或指针）形参（参见 4.11.2 节，第 144 页）。
- 数组或函数指针转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的指针。类似的，一个函数实参可以转换为一个该函数类型的指针（参见 4.11.2 节，第 143 页）。

其他类型转换，如算术转换（参见 4.11.1 节，第 142 页）、派生类向基类的转换（参见 15.2.2 节，第 530 页）以及用户定义的转换（参见 7.5.4 节，第 263 页和 14.9 节，第 514 页），都不能应用于函数模板。

作为一个例子，考虑对函数 `fobj` 和 `oref` 的调用。`fobj` 函数拷贝它的参数，而 `oref` 的参数是引用类型：

```
template <typename T> T fobj(T, T); // 实参被拷贝
template <typename T> T oref(const T&, const T&); // 引用
string s1("a value");
const string s2("another value");
fobj(s1, s2); // 调用 fobj(string, string); const 被忽略
oref(s1, s2); // 调用 oref(const string&, const string&)
               // 将 s1 转换为 const 是允许的
int a[10], b[42];
fobj(a, b); // 调用 f(int*, int*)
oref(a, b); // 错误：数组类型不匹配
```

在第一对调用中，我们传递了一个 `string` 和一个 `const string`。虽然这些类型不严格匹配，但两个调用都是合法的。在 `fobj` 调用中，实参被拷贝，因此原对象是否是 `const` 没有关系。在 `oref` 调用中，参数类型是 `const` 的引用。对于一个引用参数来说，转换为 `const` 是允许的，因此这个调用也是合法的。

在下一对调用中，我们传递了数组实参，两个数组大小不同，因此是不同类型。在 `fobj` 调用中，数组大小不同无关紧要。两个数组都被转换为指针。`fobj` 中的模板类型为 `int*`。但是，`oref` 调用是不合法的。如果形参是一个引用，则数组不会转换为指针（参见 6.2.4 节，第 195 页）。`a` 和 `b` 的类型是不匹配的，因此调用是错误的。



将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有 `const` 转换及数组或函数到指针的转换。



使用相同模板参数类型的函数形参

一个模板类型参数可以用作多个函数形参的类型。由于只允许有限的几种类型转换，因此传递给这些形参的实参必须具有相同的类型。如果推断出的类型不匹配，则调用就是错误的。例如，我们的 `compare` 函数（参见 16.1.1 节，第 578 页）接受两个 `const T&` 参数，其实参必须是相同类型：

```
long lng;
compare(lng, 1024); // 错误：不能实例化 compare(long, int)
```

此调用是错误的，因为传递给 `compare` 的实参类型不同。从第一个函数实参推断出的模板实参为 `long`，从第二个函数实参推断出的模板实参为 `int`。这些类型不匹配，因此模板实参推断失败。

如果希望允许对函数实参进行正常的类型转换，我们可以将函数模板定义为两个类型参数：

```
// 实参类型可以不同，但必须兼容
template <typename A, typename B>
int flexibleCompare(const A& v1, const B& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

现在用户可以提供不同类型的实参了：

```
long lng;
flexibleCompare(lng, 1024); // 正确：调用 flexibleCompare(long, int)
```

当然，必须定义了能比较这些类型的值的`<`运算符。

正常类型转换应用于普通函数实参

函数模板可以有普通类型定义的参数，即，不涉及模板类型参数的类型。这种函数实参不进行特殊处理；它们正常转换为对应形参的类型（参见 6.1 节，第 183 页）。例如，考虑下面的模板：

```
template <typename T> ostream &print(ostream &os, const T &obj)
{
    return os << obj;
}
```

第一个函数参数是一个已知类型 `ostream&`。第二个参数 `obj` 则是模板参数类型。由于 `os` 的类型是固定的，因此当调用 `print` 时，传递给它的实参会进行正常的类型转换：

```
681 print(cout, 42); // 实例化 print(ostream&, int)
                    ofstream f("output");
                    print(f, 10); // 使用 print(ostream&, int); 将 f 转换为 ostream&
```

在第一个调用中，第一个实参的类型严格匹配第一个参数的类型。此调用会实例化接受一个 `ostream&` 和一个 `int` 的 `print` 版本。在第二个调用中，第一个实参是一个 `ofstream`，它可以转换为 `ostream&`（参见 8.2.1 节，第 284 页）。由于此参数的类型不依赖于模板参数，因此编译器会将 `f` 隐式转换为 `ostream&`。



如果函数参数类型不是模板参数，则对实参进行正常的类型转换。

16.2.1 节练习

练习 16.32: 在模板实参推断过程中发生了什么？

练习 16.33: 指出在模板实参推断过程中允许对函数实参进行的两种类型转换。

练习 16.34: 对下面的代码解释每个调用是否合法。如果合法，`T` 的类型是什么？如果不合法，为什么？

```
template <class T> int compare(const T&, const T&);
(a) compare("hi", "world"); (b) compare("bye", "dad");
```

练习 16.35: 下面调用中哪些是错误的（如果有的话）？如果调用合法，`T` 的类型是什么？如果调用不合法，问题何在？

```
template <typename T> T calc(T, int);
template <typename T> T fcn(T, T);
double d; float f; char c;
(a) calc(c, 'c'); (b) calc(d, f);
(c) fcn(c, 'c'); (d) fcn(d, f);
```

练习 16.36: 进行下面的调用会发生什么：

```
template <typename T> f1(T, T);
template <typename T1, typename T2> f2(T1, T2);
int i = 0, j = 42, *p1 = &i, *p2 = &j;
const int *cp1 = &i, *cp2 = &j;
(a) f1(p1, p2); (b) f2(p1, p2); (c) f1(cp1, cp2);
(d) f2(cp1, cp2); (e) f1(p1, cp1); (f) f2(p1, cp1);
```

16.2.2 函数模板显式实参

在某些情况下，编译器无法推断出模板实参的类型。其他一些情况下，我们希望允许用户控制模板实例化。当函数返回类型与参数列表中任何类型都不相同时，这两种情况最常出现。

< 682

指定显式模板实参

作为一个允许用户指定使用类型的例子，我们将定义一个名为 `sum` 的函数模板，它接受两个不同类型的参数。我们希望允许用户指定结果的类型。这样，用户就可以选择合适的精度。

我们可以定义表示返回类型的第三个模板参数，从而允许用户控制返回类型：

```
// 编译器无法推断 T1，它未出现在函数参数列表中
template <typename T1, typename T2, typename T3>
T1 sum(T2, T3);
```

在本例中，没有任何函数实参的类型可用来推断 `T1` 的类型。每次调用 `sum` 时调用者都必须为 `T1` 提供一个显式模板实参（explicit template argument）。

我们提供显式模板实参的方式与定义类模板实例的方式相同。显式模板实参在尖括号中给出，位于函数名之后，实参列表之前：

```
// T1 是显式指定的, T2 和 T3 是从函数实参类型推断而来的
auto val3 = sum<long long>(i, lng); // long long sum(int, long)
```

此调用显式指定 T1 的类型。而 T2 和 T3 的类型则由编译器从 i 和 lng 的类型推断出来。

显式模板实参按由左至右的顺序与对应的模板参数匹配；第一个模板实参与第一个模板参数匹配，第二个实参与第二个参数匹配，依此类推。只有尾部（最右）参数的显式模板实参才可以忽略，而且前提是它们可以从函数参数推断出来。如果我们的 sum 函数按照如下形式编写：

```
// 糟糕的设计：用户必须指定所有三个模板参数
template <typename T1, typename T2, typename T3>
T3 alternative_sum(T2, T1);
```

则我们总是必须为所有三个形参指定实参：

```
// 错误：不能推断前几个模板参数
auto val3 = alternative_sum<long long>(i, lng);
// 正确：显式指定了所有三个参数
auto val2 = alternative_sum<long long, int, long>(i, lng);
```

正常类型转换应用于显式指定的实参

对于用普通类型定义的函数参数，允许进行正常的类型转换（参见 16.2.1 节，第 602 页），出于同样的原因，对于模板类型参数已经显式指定了的函数实参，也进行正常的类型转换：

683	long lng;	
	compare(lng, 1024);	// 错误：模板参数不匹配
	compare<long>(lng, 1024);	// 正确：实例化 compare(long, long)
	compare<int>(lng, 1024);	// 正确：实例化 compare(int, int)

如我们所见，第一个调用是错误的，因为传递给 compare 的实参必须具有相同的类型。如果我们显式指定模板类型参数，就可以进行正常类型转换了。因此，调用 compare<long> 等价于调用一个接受两个 const long& 参数的函数。int 类型的参数被自动转化为 long。在第三个调用中，T 被显式指定为 int，因此 lng 被转换为 int。

16.2.2 节练习

练习 16.37：标准库 max 函数有两个参数，它返回实参中的较大者。此函数有一个模板类型参数。你能在调用 max 时传递给它一个 int 和一个 double 吗？如果可以，如何做？如果不可以，为什么？

练习 16.38：当我们调用 make_share（参见 12.1.1 节，第 401 页）时，必须提供一个显式模板实参。解释为什么需要显式模板实参以及它是如何使用的。

练习 16.39：对 16.1.1 节（第 578 页）中的原始版本的 compare 函数，使用一个显式模板实参，使得可以向函数传递两个字符串字面常量。



16.2.3 尾置返回类型与类型转换

当我们希望用户确定返回类型时，用显式模板实参表示模板函数的返回类型是很有效的。但在其他情况下，要求显式指定模板实参会给用户增添额外负担，而且不会带来什么好处。例如，我们可能希望编写一个函数，接受表示序列的一对迭代器和返回序列中一个

元素的引用:

```
template <typename It>
??? &fcn(It beg, It end)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的引用
}
```

我们并不知道返回结果的准确类型，但知道所需类型是所处理的序列的元素类型:

```
vector<int> vi = {1,2,3,4,5};
Blob<string> ca = { "hi", "bye" };
auto &i = fcn(vi.begin(), vi.end()); // fcn 应该返回 int&
auto &s = fcn(ca.begin(), ca.end()); // fcn 应该返回 string&
```

此例中，我们知道函数应该返回`*beg`，而且知道我们可以用`decltype(*beg)`来获取此表达式的类型。但是，在编译器遇到函数的参数列表之前，`beg`都是不存在的。为了定义此函数，我们必须使用尾置返回类型（参见 6.3.3 节，第 206 页）。由于尾置返回出现在参数列表之后，它可以使用函数的参数:

```
// 尾置返回允许我们在参数列表之后声明返回类型
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的引用
}
```

此例中我们通知编译器`fcn`的返回类型与解引用`beg`参数的结果类型相同。解引用运算符返回一个左值（参见 4.1.1 节，第 121 页），因此通过`decltype`推断的类型为`beg`表示的元素的类型的引用。因此，如果对一个`string`序列调用`fcn`，返回类型将是`string&`。如果是`int`序列，则返回类型是`int&`。

进行类型转换的标准库模板类

有时我们无法直接获得所需要的类型。例如，我们可能希望编写一个类似`fcn`的函数，但返回一个元素的值（参见 6.3.2 节，第 201 页）而非引用。

在编写这个函数的过程中，我们面临一个问题：对于传递的参数的类型，我们几乎一无所知。在此函数中，我们知道唯一可以使用的操作是迭代器操作，而所有迭代器操作都不会生成元素，只能生成元素的引用。

为了获得元素类型，我们可以使用标准库的**类型转换**（type transformation）模板。这些模板定义在头文件`type_traits`中。这个头文件中的类通常用于所谓的模板元程序设计，这一主题已超出本书的范围。但是，类型转换模板在普通编程中也很有用。表 16.1 列出了这些模板，我们将在 16.5 节（第 624 页）中看到它们是如何实现的。

在本例中，我们可以使用`remove_reference`来获得元素类型。`remove_reference`模板有一个模板类型参数和一个名为`type`的（public）类型成员。如果我们用一个引用类型实例化`remove_reference`，则`type`将表示被引用的类型。例如，如果我们实例化`remove_reference<int&>`，则`type`成员将是`int`。类似的，如果我们实例化`remove_reference<string&>`，则`type`成员将是`string`，依此类推。更一般的，给定一个迭代器`beg`:

<684

C++
11

```
remove_reference<decltype(*beg)>::type
```

将获得 beg 引用的元素的类型: decltype(*beg) 返回元素类型的引用类型。remove_reference::type 脱去引用, 剩下元素类型本身。

组合使用 remove_reference、尾置返回及 decltype, 我们就可以在函数中返回元素值的拷贝:

685 // 为了使用模板参数的成员, 必须用 typename, 参见 16.1.3 节(第 593 页)

```
template <typename It>
auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

注意, type 是一个类的成员, 而该类依赖于一个模板参数。因此, 我们必须在返回类型的声明中使用 typename 来告知编译器, type 表示一个类型(参见 16.1.3 节, 第 593 页)

表 16.1: 标准类型转换模板

对 Mod<T>, 其中 Mod 为	若 T 为	则 Mod<T>::type 为
remove_reference	X&或 X&&	X
	否则	T
add_const	X&、const X 或函数	T
	否则	const T
add_lvalue_reference	X&	T
	X&&	X&
	否则	T&
add_rvalue_reference	X&或 X&&	T
	否则	T&&
remove_pointer	X*	X
	否则	T
add_pointer	X&或 X&&	X*
	否则	T*
make_signed	unsigned X	X
	否则	T
make_unsigned	带符号类型	unsigned X
	否则	T
remove_extent	X[n]	X
	否则	T
remove_all_extents	X[n1][n2]...	X
	否则	T

表 16.1 中描述的每个类型转换模板的工作方式都与 remove_reference 类似。每个模板都有一个名为 type 的 public 成员, 表示一个类型。此类型与模板自身的模板类型参数相关, 其关系如模板名所示。如果不可能(或者不必要)转换模板参数, 则 type 成员就是模板参数类型本身。例如, 如果 T 是一个指针类型, 则 remove_pointer<T>::type 是 T 指向的类型。如果 T 不是一个指针, 则无须进行任何

转换，从而 `type` 具有与 `T` 相同的类型。

16.2.3 节练习

686

练习 16.40：下面的函数是否合法？如果不合法，为什么？如果合法，对可以传递的实参类型有什么限制（如果有的话）？返回类型是什么？

```
template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0)
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

练习 16.41：编写一个新的 `sum` 版本，它的返回类型保证足够大，足以容纳加法结果。

16.2.4 函数指针和实参推断



当我们用一个函数模板初始化一个函数指针或为一个函数指针赋值（参见 6.7 节，第 221 页）时，编译器使用指针的类型来推断模板实参。

例如，假定我们有一个函数指针，它指向的函数返回 `int`，接受两个参数，每个参数都是指向 `const int` 的引用。我们可以使用该指针指向 `compare` 的一个实例：

```
template <typename T> int compare(const T&, const T&);
// pf1 指向实例 int compare(const int&, const int&)
int (*pf1)(const int&, const int&) = compare;
```

`pf1` 中参数的类型决定了 `T` 的模板实参的类型。在本例中，`T` 的模板实参类型为 `int`。指针 `pf1` 指向 `compare` 的 `int` 版本实例。如果不能从函数指针类型确定模板实参，则产生错误：

```
// func 的重载版本；每个版本接受一个不同的函数指针类型
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));
func(compare); // 错误：使用 compare 的哪个实例？
```

这段代码的问题在于，通过 `func` 的参数类型无法确定模板实参的唯一类型。对 `func` 的调用既可以实例化接受 `int` 的 `compare` 版本，也可以实例化接受 `string` 的版本。由于不能确定 `func` 的实参的唯一实例化版本，此调用将编译失败。

我们可以通过使用显式模板实参来消除 `func` 调用的歧义：

```
// 正确：显式指出实例化哪个 compare 版本
func(compare<int>); // 传递 compare(const int&, const int&)
```

此表达式调用的 `func` 版本接受一个函数指针，该指针指向的函数接受两个 `const int&` 参数。



当参数是一个函数模板实例的地址时，程序上下文必须满足：对每个模板参数，能唯一确定其类型或值。

687

16.2.5 模板实参推断和引用

为了理解如何从函数调用进行类型推断，考虑下面的例子：

```
template <typename T> void f(T &p);
```

其中函数参数 p 是一个模板类型参数 T 的引用，非常重要的是记住两点：编译器会应用正常的引用绑定规则；const 是底层的，不是顶层的。

从左值引用函数参数推断类型

当一个函数参数是模板类型参数的一个普通（左值）引用时（即，形如 T&），绑定规则告诉我们，只能传递给它一个左值（如，一个变量或一个返回引用类型的表达式）。实参可以是 const 类型，也可以不是。如果实参是 const 的，则 T 将被推断为 const 类型：

```
template <typename T> void f1(T&); // 实参必须是一个左值
// 对 f1 的调用使用实参所引用的类型作为模板参数类型
f1(i); // i 是一个 int；模板参数类型 T 是 int
f1(ci); // ci 是一个 const int；模板参数 T 是 const int
f1(5); // 错误：传递给一个&参数的实参必须是一个左值
```

如果一个函数参数的类型是 const T&，正常的绑定规则告诉我们可以传递给它任何类型的实参——一个对象（const 或非 const）、一个临时对象或是一个字面常量值。当函数参数本身是 const 时，T 的类型推断的结果不会是一个 const 类型。const 已经是函数参数类型的一部分；因此，它不会也是模板参数类型的一部分：

```
template <typename T> void f2(const T&); // 可以接受一个右值
// f2 中的参数是 const &；实参中的 const 是无关的
// 在每个调用中，f2 的函数参数都被推断为 const int&
f2(i); // i 是一个 int；模板参数 T 是 int
f2(ci); // ci 是一个 const int，但模板参数 T 是 int
f2(5); // 一个 const &参数可以绑定到一个右值；T 是 int
```

从右值引用函数参数推断类型

当一个函数参数是一个右值引用（参见 13.6.1 节，第 471 页）（即，形如 T&&）时，正常绑定规则告诉我们可以传递给它一个右值。当我们这样做时，类型推断过程类似普通左值引用函数参数的推断过程。推断出的 T 的类型是该右值实参的类型：

```
template <typename T> void f3(T&&);
f3(42); // 实参是一个 int 类型的右值；模板参数 T 是 int
```

688> 引用折叠和右值引用参数

假定 i 是一个 int 对象，我们可能认为像 f3(i) 这样的调用是不合法的。毕竟，i 是一个左值，而通常我们不能将一个右值引用绑定到一个左值上。但是，C++ 语言在正常绑定规则之外定义了两个例外规则，允许这种绑定。这两个例外规则是 move 这种标准库设施正确工作的基础。

第一个例外规则影响右值引用参数的推断如何进行。当我们将一个左值（如 i）传递给函数的右值引用参数，且此右值引用指向模板类型参数（如 T&&）时，编译器推断模板类型参数为实参的左值引用类型。因此，当我们调用 f3(i) 时，编译器推断 T 的类型为 int&，而非 int。

T 被推断为 int& 看起来好像意味着 f3 的函数参数应该是一个类型 int& 的右值引用。

通常，我们不能（直接）定义一个引用的引用（参见 2.3.1 节，第 46 页）。但是，通过类型别名（参见 2.5.1 节，第 60 页）或通过模板类型参数间接定义是可以的。

在这种情况下，我们可以使用第二个例外绑定规则：如果我们间接创建一个引用的引用，则这些引用形成了“折叠”。在所有情况下（除了一个例外），引用会折叠成一个普通的左值引用类型。在新标准中，折叠规则扩展到右值引用。只有一种特殊情况下引用会折叠成右值引用：右值引用的右值引用。即，对于一个给定类型 X ：

- $X\& \&$ 、 $X\& \&&$ 和 $X\&\& \&$ 都折叠成类型 $X\&$
- 类型 $X\&\& \&$ 折叠成 $X\&\&$



引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数。

如果将引用折叠规则和右值引用的特殊类型推断规则组合在一起，则意味着我们可以对一个左值调用 $f3$ 。当我们把一个左值传递给 $f3$ 的（右值引用）函数参数时，编译器推断 T 为一个左值引用类型：

```
f3(i); // 实参是一个左值；模板参数 T 是 int&
f3(ci); // 实参是一个左值；模板参数 T 是一个 const int&
```

当一个模板参数 T 被推断为引用类型时，折叠规则告诉我们函数参数 $T\&\&$ 折叠为一个左值引用类型。例如， $f3(i)$ 的实例化结果可能像下面这样：

```
// 无效代码，只是用于演示目的
void f3<int&>(int& &&); // 当 T 是 int& 时，函数参数为 int& &&
```

$f3$ 的函数参数是 $T\&\&$ 且 T 是 $int\&$ ，因此 $T\&\&$ 是 $int\& \&\&$ ，会折叠成 $int\&$ 。因此，即使 $f3$ 的函数参数形式是一个右值引用（即， $T\&\&$ ），此调用也会用一个左值引用类型（即， $int\&$ ）实例化 $f3$ ：

```
void f3<int&>(int&); // 当 T 是 int& 时，函数参数折叠为 int&
```

这两个规则导致了两个重要结果：

- 如果一个函数参数是一个指向模板类型参数的右值引用（如， $T\&\&$ ），则它可以被绑定到一个左值；且
- 如果实参是一个左值，则推断出的模板实参类型将是一个左值引用，且函数参数将被实例化为一个（普通）左值引用参数（ $T\&$ ）

另外值得注意的是，这两个规则暗示，我们可以将任意类型的实参传递给 $T\&\&$ 类型的函数参数。对于这种类型的参数，（显然）可以传递给它右值，而如我们刚刚看到的，也可以传递给它左值。



如果一个函数参数是指向模板参数类型的右值引用（如， $T\&\&$ ），则可以传递给它任意类型的实参。如果将一个左值传递给这样的参数，则函数参数被实例化为一个普通的左值引用（ $T\&$ ）。

编写接受右值引用参数的模板函数

模板参数可以推断为一个引用类型，这一特性对模板内的代码可能有令人惊讶的影响：

```
template <typename T> void f3(T&& val)
{
    T t = val; // 拷贝还是绑定一个引用？
```

```
t = fcn(t); // 赋值只改变 t 还是既改变 t 又改变 val?
if (val == t) { /* ... */ } // 若 T 是引用类型，则一直为 true
}
```

当我们对一个右值调用 `f3` 时，例如字面常量 42，`T` 为 `int`。在此情况下，局部变量 `t` 的类型为 `int`，且通过拷贝参数 `val` 的值被初始化。当我们对 `t` 赋值时，参数 `val` 保持不变。

另一方面，当我们对一个左值 `i` 调用 `f3` 时，则 `T` 为 `int&`。当我们定义并初始化局部变量 `t` 时，赋予它类型 `int&`。因此，对 `t` 的初始化将其绑定到 `val`。当我们对 `t` 赋值时，也同时改变了 `val` 的值。在 `f3` 的这个实例化版本中，`if` 判断永远得到 `true`。

当代码中涉及的类型可能是普通（非引用）类型，也可能是引用类型时，编写正确的代码就变得异常困难（虽然 `remove_reference` 这样的类型转换类可能会有帮助（参见 16.2.3 节，第 605 页））。

在实际中，右值引用通常用于两种情况：模板转发其实参或模板被重载。我们将在 16.2.7 节（第 612 页）中介绍实参转发，在 16.3 节（第 614 页）中介绍模板重载。

目前应该注意的是，使用右值引用的函数模板通常使用我们在 13.6.3 节（第 481 页）中看到的方式来进行重载：

```
template <typename T> void f(T&&);           // 绑定到非 const 右值
template <typename T> void f(const T&);        // 左值和 const 右值
```

与非模板函数一样，第一个版本将绑定到可修改的右值，而第二个版本将绑定到左值或 `const` 右值。

690

16.2.5 节练习

练习 16.42: 对下面每个调用，确定 `T` 和 `val` 的类型：

```
template <typename T> void g(T&& val);
int i = 0; const int ci = i;
(a) g(i); (b) g(ci); (c) g(i * ci);
```

练习 16.43: 使用上一题定义的函数，如果我们调用 `g(i = ci)`，`g` 的模板参数将是什么？

练习 16.44: 使用与第一题中相同的三个调用，如果 `g` 的函数参数声明为 `T`（而不是 `T&&`），确定 `T` 的类型。如果 `g` 的函数参数是 `const T&` 呢？

练习 16.45: 给定下面的模板，如果我们对一个像 42 这样的字面常量调用 `g`，解释会发什么？如果我们对一个 `int` 类型的变量调用 `g` 呢？

```
template <typename T> void g(T&& val) { vector<T> v; }
```



16.2.6 理解 `std::move`

标准库 `move` 函数（参见 13.6.1 节，第 472 页）是使用右值引用的模板的一个很好的例子。幸运的是，我们不必理解 `move` 所使用的模板机制也可以直接使用它。但是，研究 `move` 是如何工作的可以帮助我们巩固对模板的理解和使用。

在 13.6.2 节（第 473 页）中我们注意到，虽然不能直接将一个右值引用绑定到一个左值上，但可以用 `move` 获得一个绑定到左值上的右值引用。由于 `move` 本质上可以接受任

何类型的实参，因此我们不会惊讶于它是一个函数模板。

std::move 是如何定义的

标准库是这样定义 move 的：

```
// 在返回类型和类型转换中也要用到 typename，参见 16.1.3 节（第 593 页）
// remove_reference 是在 16.2.3 节（第 605 页）中介绍的
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    // static_cast 是在 4.11.3 节（第 145 页）中介绍的
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

这段代码很短，但其中有些微妙之处。首先，move 的函数参数 `T&&` 是一个指向模板类型参数的右值引用。通过引用折叠，此参数可以与任何类型的实参匹配。特别是，我们既可以传递给 move 一个左值，也可以传递给它一个右值：

```
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // 正确：从一个右值移动数据
s2 = std::move(s1); // 正确：但在赋值之后，s1 的值是不确定的
```

std::move 是如何工作的

在第一个赋值中，传递给 move 的实参是 string 的构造函数的右值结果——`string("bye!")`。如我们已经见到过的，当向一个右值引用函数参数传递一个右值时，由实参推断出的类型为被引用的类型（参见 16.2.5 节，第 608 页）。因此，在 `std::move(string("bye!"))` 中：

- 推断出的 `T` 的类型为 `string`。
- 因此，`remove_reference` 用 `string` 进行实例化。
- `remove_reference<string>` 的 `type` 成员是 `string`。
- `move` 的返回类型是 `string&&`。
- `move` 的函数参数 `t` 的类型为 `string&&`。

因此，这个调用实例化 `move<string>`，即函数

```
string&& move(string &t)
```

函数体返回 `static_cast<string&&>(t)`。`t` 的类型已经是 `string&&`，于是类型转换什么都不做。因此，此调用的结果就是它所接受的右值引用。

现在考虑第二个赋值，它调用了 `std::move()`。在此调用中，传递给 move 的实参是一个左值。这样：

- 推断出的 `T` 的类型为 `string&` (`string` 的引用，而非普通 `string`)。
- 因此，`remove_reference` 用 `string&` 进行实例化。
- `remove_reference<string&>` 的 `type` 成员是 `string`。
- `move` 的返回类型仍是 `string&&`。
- `move` 的函数参数 `t` 实例化为 `string& &&`，会折叠为 `string&`。

因此，这个调用实例化 `move<string&>`，即

```
string&& move(string &t)
```



691

这正是我们所寻求的——我们希望将一个右值引用绑定到一个左值。这个实例的函数体返回 `static_cast<string&&>(t)`。在此情况下，`t` 的类型为 `string&`，`cast` 将其转换为 `string&&`。

从一个左值 `static_cast` 到一个右值引用是允许的

C++ 11 通常情况下，`static_cast` 只能用于其他合法的类型转换（参见 4.11.3 节，第 145 页）。但是，这里又有一条针对右值引用的特许规则：虽然不能隐式地将一个左值转换为右值引用，但我们可以用 `static_cast` 显式地将一个左值转换为一个右值引用。

692 对于操作右值引用的代码来说，将一个右值引用绑定到一个左值的特性允许它们截断左值。有时候，例如在我们的 `StrVec` 类的 `reallocate` 函数（参见 13.6.1 节，第 469 页）中，我们知道截断一个左值是安全的。一方面，通过允许进行这样的转换，C++ 语言认可了这种用法。但另一方面，通过强制使用 `static_cast`，C++ 语言试图阻止我们意外地进行这种转换。

最后，虽然我们可以直接编写这种类型转换代码，但使用标准库 `move` 函数是容易得多的方式。而且，统一使用 `std::move` 使得我们在程序中查找潜在的截断左值的代码变得很容易。

16.2.6 节练习

练习 16.46：解释下面的循环，它来自 13.5 节（第 469 页）中的 `StrVec::reallocate`：

```
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
```

16.2.7 转发

某些函数需要将其一个或多个实参连同类型不变地转发给其他函数。在此情况下，我们需要保持被转发实参的所有性质，包括实参类型是否是 `const` 的以及实参是左值还是右值。

作为一个例子，我们将编写一个函数，它接受一个可调用表达式和两个额外实参。我们的函数将调用给定的可调用对象，将两个额外参数逆序传递给它。下面是我们的翻转函数的初步模样：

```
// 接受一个可调用对象和另外两个参数的模板
// 对“翻转”的参数调用给定的可调用对象
// flip1 是一个不完整的实现：顶层 const 和引用丢失了
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}
```

这个函数一般情况下工作得很好，但当我们希望用它调用一个接受引用参数的函数时就会出现问题：

```
void f(int v1, int &v2) // 注意 v2 是一个引用
{
    cout << v1 << " " << ++v2 << endl;
}
```

在这段代码中，`f` 改变了绑定到 `v2` 的实参的值。但是，如果我们通过 `flip1` 调用 `f`，`f` 所做的改变就不会影响实参：

```
f(42, i);           // f 改变了实参 i
flip1(f, j, 42); // 通过 flip1 调用 f 不会改变 j
```

问题在于 `j` 被传递给 `flip1` 的参数 `t1`。此参数是一个普通的、非引用的类型 `int`，而非 `int&`。因此，这个 `flip1` 调用会实例化为

```
void flip1(void(*fcn)(int, int&), int t1, int t2);
```

`j` 的值被拷贝到 `t1` 中。`f` 中的引用参数被绑定到 `t1`，而非 `j`，从而其改变不会影响 `j`。

定义能保持类型信息的函数参数

为了通过翻转函数传递一个引用，我们需要重写函数，使其参数能保持给定实参的“左值性”。更进一步，可以想到我们也希望保持参数的 `const` 属性。

通过将一个函数参数定义为一个指向模板类型参数的右值引用，我们可以保持其对应实参的所有类型信息。而使用引用参数（无论是左值还是右值）使得我们可以保持 `const` 属性，因为在引用类型中的 `const` 是底层的。如果我们将函数参数定义为 `T1&&` 和 `T2&&`，通过引用折叠（参见 16.2.5 节，第 608 页）就可以保持翻转实参的左值/右值属性（参见 16.2.5 节，第 608 页）：

```
template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}
```

与较早的版本一样，如果我们调用 `flip2(f, j, 42)`，将传递给参数 `t1` 一个左值 `j`。但是，在 `flip2` 中，推断出的 `T1` 的类型为 `int&`，这意味着 `t1` 的类型会折叠为 `int&`。由于是引用类型，`t1` 被绑定到 `j` 上。当 `flip2` 调用 `f` 时，`f` 中的引用参数 `v2` 被绑定到 `t1`，也就是被绑定到 `j`。当 `f` 递增 `v2` 时，它也同时改变了 `j` 的值。



如果一个函数参数是指向模板类型参数的右值引用（如 `T&&`），它对应的实参的 `const` 属性和左值/右值属性将得到保持。

这个版本的 `flip2` 解决了一半问题。它对于接受一个左值引用的函数工作得很好，但不能用于接受右值引用参数的函数。例如：

```
void g(int &&i, int& j)
{
    cout << i << " " << j << endl;
}
```

如果我们试图通过 `flip2` 调用 `g`，则参数 `t2` 将被传递给 `g` 的右值引用参数。即使我们传递一个右值给 `flip2`：

```
flip2(g, i, 42); // 错误：不能从一个左值实例化 int&
```

传递给 `g` 的将是 `flip2` 中名为 `t2` 的参数。函数参数与其他任何变量一样，都是左值表达式（参见 13.6.1 节，第 471 页）。因此，`flip2` 中对 `g` 的调用将传递给 `g` 的右值引用参数一个左值。

在调用中使用 std::forward 保持类型信息

694 >

我们可以使用一个名为 `forward` 的新标准库设施来传递 `flip2` 的参数，它能保持原始实参的类型。类似 `move`, `forward` 定义在头文件 `utility` 中。与 `move` 不同，`forward` 必须通过显式模板实参来调用（参见 16.2.2 节，第 603 页）。`forward` 返回该显式实参类型的右值引用。即，`forward<T>` 的返回类型是 `T&&`。

 11

通常情况下，我们使用 `forward` 传递那些定义为模板类型参数的右值引用的函数参数。通过其返回类型上的引用折叠，`forward` 可以保持给定实参的左值/右值属性：

```
template <typename Type> intermediary(Type &&arg)
{
    finalFcn(std::forward<Type>(arg));
    // ...
}
```

本例中我们使用 `Type` 作为 `forward` 的显式模板实参类型，它是从 `arg` 推断出来的。由于 `arg` 是一个模板类型参数的右值引用，`Type` 将表示传递给 `arg` 的实参的所有类型信息。如果实参是一个右值，则 `Type` 是一个普通（非引用）类型，`forward<Type>` 将返回 `Type&&`。如果实参是一个左值，则通过引用折叠，`Type` 本身是一个左值引用类型。在此情况下，返回类型是一个指向左值引用类型的右值引用。再次对 `forward<Type>` 的返回类型进行引用折叠，将返回一个左值引用类型。



当用于一个指向模板参数类型的右值引用函数参数 (`T&&`) 时，`forward` 会保持实参类型的所有细节。

使用 `forward`，我们可以再次重写翻转函数：

```
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}
```

如果我们调用 `flip(g, i, 42)`，`i` 将以 `int&` 类型传递给 `g`，`42` 将以 `int&&` 类型传递给 `g`。



与 `std::move` 相同，对 `std::forward` 不使用 `using` 声明是一个好主意。我们将在 18.2.3 节（第 706 页）中解释原因。

16.2.7 节练习

练习 16.47： 编写你自己版本的翻转函数，通过调用接受左值和右值引用参数的函数来测试它。



16.3 重载与模板

函数模板可以被另一个模板或一个普通非模板函数重载。与往常一样，名字相同的函数必须具有不同数量或类型的参数。

695 >

如果涉及函数模板，则函数匹配规则（参见 6.4 节，第 209 页）会在以下几方面受到

影响：

- 对于一个调用，其候选函数包括所有模板实参推断（参见 16.2 节，第 600 页）成功的函数模板实例。
- 候选的函数模板总是可行的，因为模板实参推断会排除任何不可行的模板。
- 与往常一样，可行函数（模板与非模板）按类型转换（如果对此调用需要的话）来排序。当然，可以用于函数模板调用的类型转换是非常有限的（参见 16.2.1 节，第 601 页）。
- 与往常一样，如果恰有一个函数提供比任何其他函数都更好的匹配，则选择此函数。但是，如果有多个函数提供同样好的匹配，则：
 - 如果同样好的函数中只有一个是非模板函数，则选择此函数。
 - 如果同样好的函数中没有非模板函数，而有多个函数模板，且其中一个模板比其他模板更特例化，则选择此模板。
 - 否则，此调用有歧义。



正确定义一组重载的函数模板需要对类型间的关系及模板函数允许的有限的实参类型转换有深刻的理解。

编写重载模板

作为一个例子，我们将构造一组函数，它们在调试中可能很有用。我们将这些调试函数命名为 `debug_rep`，每个函数都返回一个给定对象的 `string` 表示。我们首先编写此函数的最通用版本，将它定义为一个模板，接受一个 `const` 对象的引用：

```
// 打印任何我们不能处理的类型
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret; // 参见 8.3 节 (第 287 页)
    ret << t; // 使用 T 的输出运算符打印 t 的一个表示形式
    return ret.str(); // 返回 ret 绑定的 string 的一个副本
}
```

此函数可以用来生成一个对象对应的 `string` 表示，该对象可以是任意具备输出运算符的类型。◀ 696

接下来，我们将定义打印指针的 `debug_rep` 版本：

```
// 打印指针的值，后跟指针指向的对象
// 注意：此函数不能用于 char*；参见 16.3 节 (第 617 页)
template <typename T> string debug_rep(T *p)
{
    ostringstream ret;
    ret << "pointer: " << p; // 打印指针本身的价值
    if (p)
        ret << " " << debug_rep(*p); // 打印 p 指向的值
    else
        ret << " null pointer"; // 或指出 p 为空
    return ret.str(); // 返回 ret 绑定的 string 的一个副本
}
```

此版本生成一个 `string`，包含指针本身的价值和调用 `debug_rep` 获得的指针指向的值。注意此函数不能用于打印字符指针，因为 IO 库为 `char*` 值定义了一个 `<<` 版本。此 `<<` 版本假定指针表示一个空字符结尾的字符数组，并打印数组的内容而非地址值。我们将在 16.3

节（第 617 页）介绍如何处理字符指针。

我们可以这样使用这些函数：

```
string s("hi");
cout << debug_rep(s) << endl;
```

对于这个调用，只有第一个版本的 `debug_rep` 是可行的。第二个 `debug_rep` 版本要求一个指针参数，但在此调用中我们传递的是一个非指针对象。因此编译器无法从一个非指针实参实例化一个期望指针类型参数的函数模板，因此实参推断失败。由于只有一个可行函数，所以此函数被调用。

如果我们用一个指针调用 `debug_rep`：

```
cout << debug_rep(&s) << endl;
```

两个函数都生成可行的实例：

- `debug_rep(const string*&)`，由第一个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `string*`。
- `debug_rep(string*)`，由第二个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `string`。

第二个版本的 `debug_rep` 的实例是此调用的精确匹配。第一个版本的实例需要进行普通指针到 `const` 指针的转换。正常函数匹配规则告诉我们应该选择第二个模板，实际上编译器确实选择了这个版本。

697 多个可行模板

作为另外一个例子，考虑下面的调用：

```
const string *sp = &s;
cout << debug_rep(sp) << endl;
```

此例中的两个模板都是可行的，而且两个都是精确匹配：

- `debug_rep(const string*&)`，由第一个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `string*`。
- `debug_rep(const string*)`，由第二个版本的 `debug_rep` 实例化而来，`T` 被绑定到 `const string`。

在此情况下，正常函数匹配规则无法区分这两个函数。我们可能觉得这个调用将是有歧义的。但是，根据重载函数模板的特殊规则，此调用被解析为 `debug_rep(T*)`，即，更特例化的版本。

设计这条规则的原因是，没有它，将无法对一个 `const` 的指针调用指针版本的 `debug_rep`。问题在于模板 `debug_rep(const T&)` 本质上可以用于任何类型，包括指针类型。此模板比 `debug_rep(T*)` 更通用，后者只能用于指针类型。没有这条规则，传递 `const` 的指针的调用永远是有歧义的。



当有多个重载模板对一个调用提供同样好的匹配时，应选择最特例化的版本。

非模板和模板重载

作为下一个例子，我们将定义一个普通非模板版本的 `debug_rep` 来打印双引号包围

的 string:

```
// 打印双引号包围的 string
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
```

现在, 当我们对一个 string 调用 debug_rep 时:

```
string s("hi");
cout << debug_rep(s) << endl;
```

有两个同样好的可行函数:

- `debug_rep<string>(const string&)`, 第一个模板, T 被绑定到 `string*`。
- `debug_rep(const string&)`, 普通非模板函数。

在本例中, 两个函数具有相同的参数列表, 因此显然两者提供同样好的匹配。但是, 编译器会选择非模板版本。698 当存在多个同样好的函数模板时, 编译器选择最特例化的版本, 出于相同的原因, 一个非模板函数比一个函数模板更好。



对于一个调用, 如果一个非函数模板与一个函数模板提供同样好的匹配, 则选择非模板版本。

重载模板和类型转换

还有一种情况我们到目前为止尚未讨论: C 风格字符串指针和字符串字面常量。现在有了一个接受 string 的 debug_rep 版本, 我们可能期望一个传递字符串的调用会匹配这个版本。但是, 考虑这个调用:

```
cout << debug_rep("hi world!") << endl; // 调用 debug_rep(T*)
```

本例中所有三个 debug_rep 版本都是可行的:

- `debug_rep(const T&)`, T 被绑定到 `char[10]`。
- `debug_rep(T*)`, T 被绑定到 `const char`。
- `debug_rep(const string&)`, 要求从 `const char*` 到 `string` 的类型转换。

对给定实参来说, 两个模板都提供精确匹配——第二个模板需要进行一次(许可的)数组到指针的转换, 而对于函数匹配来说, 这种转换被认为是精确匹配(参见 6.6.1 节, 第 219 页)。非模板版本是可行的, 但需要进行一次用户定义的类型转换, 因此它没有精确匹配那么好, 所以两个模板成为可能调用的函数。与之前一样, T* 版本更加特例化, 编译器会选择它。

如果我们希望将字符指针按 string 处理, 可以定义另外两个非模板重载版本:

```
// 将字符指针转换为 string, 并调用 string 版本的 debug_rep
string debug_rep(char *p)
{
    return debug_rep(string(p));
}
string debug_rep(const char *p)
{
    return debug_rep(string(p));
}
```

缺少声明可能导致程序行为异常

值得注意的是，为了使 `char*` 版本的 `debug_rep` 正确工作，在定义此版本时，`debug_rep(const string&)` 的声明必须在作用域中。否则，就可能调用错误的 `debug_rep` 版本：

```
699> template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
// 为了使 debug_rep(char*) 的定义正确工作，下面的声明必须在作用域中
string debug_rep(const string &);
string debug_rep(char *p)
{
    // 如果接受一个 const string& 的版本的声明不在作用域中，
    // 返回语句将调用 debug_rep(const T&) 的 T 实例化为 string 的版本
    return debug_rep(string(p));
}
```

通常，如果使用了一个忘记声明的函数，代码将编译失败。但对于重载函数模板的函数而言，则不是这样。如果编译器可以从模板实例化出与调用匹配的版本，则缺少的声明就不重要了。在本例中，如果忘记了声明接受 `string` 参数的 `debug_rep` 版本，编译器会默默地实例化接受 `const T&` 的模板版本。



Tip 在定义任何函数之前，记得声明所有重载的函数版本。这样就不必担心编译器由于未遇到你希望调用的函数而实例化一个并非你所需的版本。

16.3 节练习

练习 16.48：编写你自己版本的 `debug_rep` 函数。

练习 16.49：解释下面每个调用会发生什么：

```
template <typename T> void f(T);
template <typename T> void f(const T* );
template <typename T> void g(T);
template <typename T> void g(T* );
int i = 42, *p = &i;
const int ci = 0, *p2 = &ci;
g(42); g(p); g(ci); g(p2);
f(42); f(p); f(ci); f(p2);
```

练习 16.50：定义上一个练习中的函数，令它们打印一条身份信息。运行该练习中的代码。如果函数调用的行为与你预期不符，确定你理解了原因。

16.4 可变参数模板



一个可变参数模板（variadic template）就是一个接受可变数目参数的模板函数或模板类。可变数目的参数被称为参数包（parameter packet）。存在两种参数包：**模板参数包**（template parameter packet），表示零个或多个模板参数；**函数参数包**（function parameter packet），表示零个或多个函数参数。

我们用一个省略号来指出一个模板参数或函数参数表示一个包。在一个模板参数列表

中，`class...`或`typename...`指出接下来的参数表示零个或多个类型的列表；一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表。在函数参数列表中，如果一个参数的类型是一个模板参数包，则此参数也是一个函数参数包。例如：

```
// Args 是一个模板参数包；rest 是一个函数参数包
// Args 表示零个或多个模板类型参数
// rest 表示零个或多个函数参数
template <typename T, typename... Args>
void foo(const T &t, const Args& ... rest);
```

声明了`foo`是一个可变参数函数模板，它有一个名为`T`的类型参数，和一个名为`Args`的模板参数包。这个包表示零个或多个额外的类型参数。`foo`的函数参数列表包含一个`const &`类型的参数，指向`T`的类型，还包含一个名为`rest`的函数参数包，此包表示零个或多个函数参数。

与往常一样，编译器从函数的实参推断模板参数类型。对于一个可变参数模板，编译器还会推断包中参数的数目。例如，给定下面的调用：

```
int i = 0; double d = 3.14; string s = "how now brown cow";
foo(i, s, 42, d);      // 包中有三个参数
foo(s, 42, "hi");      // 包中有两个参数
foo(d, s);              // 包中有一个参数
foo("hi");              // 空包
```

编译器会为`foo`实例化出四个不同的版本：

```
void foo(const int&, const string&, const int&, const double&);
void foo(const string&, const int&, const char[3]&);
void foo(const double&, const string&);
void foo(const char[3]&);
```

在每个实例中，`T`的类型都是从第一个实参的类型推断出来的。剩下的实参（如果有的话）提供函数额外实参的数目和类型。

sizeof...运算符

当我们需要知道包中有多少元素时，可以使用`sizeof...`运算符。类似`sizeof`（参见4.9节，第139页），`sizeof...`也返回一个常量表达式（参见2.4.4节，第58页），而且不会对其实参求值：

```
template<typename ... Args> void g(Args ... args) {
    cout << sizeof...(Args) << endl; // 类型参数的数目
    cout << sizeof...(args) << endl; // 函数参数的数目
}
```

16.4 节练习

C++
11

701

练习 16.51：调用本节中的每个`foo`，确定`sizeof...(Args)`和`sizeof...(rest)`分别返回什么。

练习 16.52：编写一个程序验证上一题的答案。

16.4.1 编写可变参数函数模板

如 6.2.6 节（第 198 页）所述，我们可以使用一个 `initializer_list` 来定义一个可接受可变数目实参的函数。但是，所有实参必须具有相同的类型（或它们的类型可以转换为同一个公共类型）。当我们既不知道想要处理的实参的数目也不知道它们的类型时，可变参数函数是很有用的。作为一个例子，我们将定义一个函数，它类似较早的 `error_msg` 函数，差别仅在于新函数实参的类型也是可变的。我们首先定义一个名为 `print` 的函数，它在一个给定流上打印给定实参列表的内容。

可变参数函数通常是递归的（参见 6.3.2 节，第 204 页）。第一步调用处理包中的第一个实参，然后用剩余实参调用自身。我们的 `print` 函数也是这样的模式，每次递归调用将第二个实参打印到第一个实参表示的流中。为了终止递归，我们还需要定义一个非可变参数的 `print` 函数，它接受一个流和一个对象：

```
// 用来终止递归并打印最后一个元素的函数
// 此函数必须在可变参数版本的 print 定义之前声明
template<typename T>
ostream &print(ostream &os, const T &t)
{
    return os << t; // 包中最后一个元素之后不打印分隔符
}
// 包中除了最后一个元素之外的其他元素都会调用这个版本的 print
template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
    os << t << ", "; // 打印第一个实参
    return print(os, rest...); // 递归调用，打印其他实参
}
```

第一个版本的 `print` 负责终止递归并打印初始调用中的最后一个实参。第二个版本的 `print` 是可变参数版本，它打印绑定到 `t` 的实参，并调用自身来打印函数参数包中的剩余值。

这段程序的关键部分是可变参数函数中对 `print` 的调用：

```
return print(os, rest...); // 递归调用，打印其他实参
```

我们的可变参数版本的 `print` 函数接受三个参数：一个 `ostream&`，一个 `const T&` 和一个参数包。而此调用只传递了两个实参。其结果是 `rest` 中的第一个实参被绑定到 `t`，剩余实参形成下一个 `print` 调用的参数包。因此，在每个调用中，包中的第一个实参被移除，成为绑定到 `t` 的实参。即，给定：

```
print(cout, i, s, 42); // 包中有两个参数
```

递归会执行如下：

调用	t	rest...
print(cout, i, s, 42)	i	s, 42
print(cout, s, 42)	s	42
print(cout, 42) 调用非可变参数版本的 print		

前两个调用只能与可变参数版本的 `print` 匹配，非可变参数版本是不可行的，因为这两个调用分别传递四个和三个实参，而非可变参数 `print` 只接受两个实参。

对于最后一次递归调用 `print(cout, 42)`，两个 `print` 版本都是可行的。这个调用传递两个实参，第一个实参的类型为 `ostream&`。因此，可变参数版本的 `print` 可以实例化为只接受两个参数：一个是 `ostream&` 参数，另一个是 `const T&` 参数。

对于最后一个调用，两个函数提供同样好的匹配。但是，非可变参数模板比可变参数模板更特例化，因此编译器选择非可变参数版本（参见 16.3 节，第 615 页）。



当定义可变参数版本的 `print` 时，非可变参数版本的声明必须在作用域中。
否则，可变参数版本会无限递归。

16.4.1 节练习

练习 16.53：编写你自己版本的 `print` 函数，并打印一个、两个及五个实参来测试它，要打印的每个实参都应有不同的类型。

练习 16.54：如果我们对一个没有<<运算符的类型调用 `print`，会发生什么？

练习 16.55：如果我们的可变参数版本 `print` 的定义之后声明非可变参数版本，解释可变参数的版本会如何执行。

16.4.2 包扩展



对于一个参数包，除了获取其大小外，我们能对它做的唯一的事情就是 **扩展** (expand) 它。当扩展一个包时，我们还要提供用于每个扩展元素的**模式** (pattern)。扩展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表。我们通过在模式右边放一个省略号 (...) 来触发扩展操作。

<703

例如，我们的 `print` 函数包含两个扩展：

```
template <typename T, typename... Args>
ostream &
print(ostream &os, const T &t, const Args&... rest)    // 扩展 Args
{
    os << t << ", ";
    return print(os, rest...);                                // 扩展 rest
}
```

第一个扩展操作扩展模板参数包，为 `print` 生成函数参数列表。第二个扩展操作出现在对 `print` 的调用中。此模式为 `print` 调用生成实参列表。

对 `Args` 的扩展中，编译器将模式 `const Arg&` 应用到模板参数包 `Args` 中的每个元素。因此，此模式的扩展结果是一个逗号分隔的零个或多个类型的列表，每个类型都形如 `const type&`。例如：

```
print(cout, i, s, 42); // 包中有两个参数
```

最后两个实参的类型和模式一起确定了尾置参数的类型。此调用被实例化为：

```
ostream&
print(ostream&, const int&, const string&, const int&);
```

第二个扩展发生在对 `print` 的（递归）调用中。在此情况下，模式是函数参数包的名字（即 `rest`）。此模式扩展出一个由包中元素组成的、逗号分隔的列表。因此，这个调

用等价于：

```
print(os, s, 42);
```

理解包扩展

`print` 中的函数参数包扩展仅仅将包扩展为其构成元素，C++语言还允许更复杂的扩展模式。例如，我们可以编写第二个可变参数函数，对其每个实参调用 `debug_rep`（参见 16.3 节，第 615 页），然后调用 `print` 打印结果 `string`：

```
// 在 print 调用中对每个实参调用 debug_rep
template <typename... Args>
ostream &errorMsg(ostream &os, const Args&... rest)
{
    // print(os, debug_rep(a1), debug_rep(a2), ..., debug_rep(an))
    return print(os, debug_rep(rest)...);
}
```

这个 `print` 调用使用了模式 `debug_reg(rest)`。此模式表示我们希望对函数参数包 `rest` 中的每个元素调用 `debug_rep`。扩展结果将是一个逗号分隔的 `debug_rep` 调用列表。即，下面调用：

```
errorMsg(cerr, fcnName, code.num(), otherData, "other", item);
```

就好像我们这样编写代码一样

```
print(cerr, debug_rep(fcnName), debug_rep(code.num()),
      debug_rep(otherData), debug_rep("otherData"),
      debug_rep(item));
```

与之相对，下面的模式会编译失败

```
// 将包传递给 debug_rep; print(os, debug_rep(a1, a2, ..., an))
print(os, debug_rep(rest...)); // 错误：此调用无匹配函数
```

这段代码的问题是我们在 `debug_rep` 调用中扩展了 `rest`，它等价于

```
print(cerr, debug_rep(fcnName, code.num(),
                      otherData, "otherData", item));
```

在这个扩展中，我们试图用一个五个实参的列表来调用 `debug_rep`，但并不存在与此调用匹配的 `debug_rep` 版本。`debug_rep` 函数不是可变参数的，而且没有哪个 `debug_rep` 版本接受五个参数。



扩展中的模式会独立地应用于包中的每个元素。

16.4.2 节练习

练习 16.56：编写并测试可变参数版本的 `errorMsg`。

练习 16.57：比较你的可变参数版本的 `errorMsg` 和 6.2.6 节（第 198 页）中的 `error_msg` 函数。两种方法的优点和缺点各是什么？



16.4.3 转发参数包

在新标准下，我们可以组合使用可变参数模板与 `forward` 机制来编写函数，实现将

其实参不变地传递给其他函数。作为例子，我们将为 `StrVec` 类（参见 13.5 节，第 465 页）添加一个 `emplace_back` 成员。标准库容器的 `emplace_back` 成员是一个可变参数成员模板（参见 16.1.4 节，第 596 页），它用其实参在容器管理的内存空间中直接构造一个元素。

我们为 `StrVec` 设计的 `emplace_back` 版本也应该是可变参数的，因为 `string` 有多个构造函数，参数各不相同。由于我们希望能使用 `string` 的移动构造函数，因此还需要保持传递给 `emplace_back` 的实参的所有类型信息。705

如我们所见，保持类型信息是一个两阶段的过程。首先，为了保持实参中的类型信息，必须将 `emplace_back` 的函数参数定义为模板类型参数的右值引用（参见 16.2.7 节，第 613 页）：

```
class StrVec {
public:
    template <class... Args> void emplace_back(Args&&...);
    // 其他成员的定义，同 13.5 节（第 465 页）
};
```

模板参数包扩展中的模式是`&&`，意味着每个函数参数将是一个指向其对应实参的右值引用。

其次，当 `emplace_back` 将这些实参传递给 `construct` 时，我们必须使用 `forward` 来保持实参的原始类型（参见 16.2.7 节，第 614 页）：

```
template <class... Args>
inline
void StrVec::emplace_back(Args&&... args)
{
    chk_n_alloc(); // 如果需要的话重新分配 StrVec 内存空间
    alloc.construct(first_free++, std::forward<Args>(args)...);
}
```

`emplace_back` 的函数体调用了 `chk_n_alloc`（参见 13.5 节，第 465 页）来确保有足够的空间容纳一个新元素，然后调用了 `construct` 在 `first_free` 指向的位置中创建了一个元素。`construct` 调用中的扩展为

```
std::forward<Args>(args)...
```

它既扩展了模板参数包 `Args`，也扩展了函数参数包 `args`。此模式生成如下形式的元素

```
std::forward<Ti>(ti)
```

其中 T_i 表示模板参数包中第 i 个元素的类型， t_i 表示函数参数包中第 i 个元素。例如，假定 `svec` 是一个 `StrVec`，如果我们调用

```
svec.emplace_back(10, 'c'); // 将 ccccccccc 添加为新的尾元素
```

`construct` 调用中的模式会扩展出

```
std::forward<int>(10), std::forward<char>(c)
```

通过在此调用中使用 `forward`，我们保证如果用一个右值调用 `emplace_back`，则 `construct` 也会得到一个右值。例如，在下面的调用中：

```
svec.emplace_back(s1 + s2); // 使用移动构造函数
```

传递给 `emplace_back` 的实参是一个右值，它将以如下形式传递给 `construct`