

```
protected:
    using Base::n;
    // ...
};
```

正如可以使用 using 声明 (3.1 节) 从命名空间使用名字, 也可以使用 using 声明访问基类中的名字, 除了在作用域操作符左边用类名字代替命名空间名字之外, 使用形式是相同的。

3. 默认继承保护级别

在 2.8 节介绍过用 struct 和 class 保留字定义类具有不同的默认访问级别, 同样, 默认继承访问级别根据使用哪个保留字定义派生类也不相同。使用 class 保留字定义的派生类默认具有 private 继承, 而用 struct 保留字定义类默认具有 public 继承:

```
class Base { /* ... */ };
struct D1 : Base { /* ... */ }; // public inheritance by default
class D2 : Base { /* ... */ }; // private inheritance by default
```

有一种常见的误解认为用 struct 保留字定义类与用 class 定义类有更大的区别。唯一的不同只是默认的成员保护级别和默认的派生保护级别, 没有其他区别:

```
class D3 : public Base {
public:
    /* ... */
};
// equivalent definition of D3
struct D3 : Base { // inheritance public by default
    /* ... */ // initial member access public by default
};
struct D4 : private Base {
private:
    /* ... */
};
// equivalent definition of D4
class D4 : Base { // inheritance private by default
    /* ... */ // initial member access private by default
};
```

574



最佳
实践

尽管私有继承在使用 class 保留字时是默认情况, 但这在实践中相对罕见。因为私有继承是如此罕见, 通常显式指定 private 是比依赖于默认更好的办法。显式指定可清楚指出想要私有继承而不是一时疏忽。

习题

习题 15.10 在 15.2.1 节的习题中编写了一个表示图书馆借阅政策的基类。假定图书馆提供下列种类的借阅资料, 每一种有自己的检查和登记政策。将这些项目组织成一个继承层次:

book	audio book	record
children's puppet	sega video game	video
cdrom book	nintendo video game	rental book
sony playstation video game		

习题 15.11 在下列包含一族类型的一般抽象中选择一种 (或者自己选择一个), 将这些类型组织成

一个继承层次。

(a) 图像文件格式 (如 gif, tiff, jpeg, bmp)

(b) 几何图元 (如矩形, 圆, 球形, 锥形)

(c) C++语言的类型 (如类, 函数, 成员函数)

习题 15.12 对上题中选择的类, 标出可能的虚函数以及 public 和 protected 成员。

15.2.6 友元关系与继承

像其他类一样, 基类或派生类可以使其他类或函数成为友元 (12.5 节)。友元可以访问类的 private 和 protected 数据。



友元关系不能继承。基类的友元对派生类的成员没有特殊访问权限。如果基类被授予友元关系, 则只有基类具有特殊访问权限, 该基类的派生类不能访问授予友元关系的类。

每个类控制对自己的成员的友元关系:

```
class Base {
    friend class Frnd;
protected:
    int i;
};
// Frnd has no access to members in D1
class D1 : public Base {
protected:
    int j;
};
class Frnd {
public:
    int mem(Base b) { return b.i; } // ok: Frnd is friend to Base
    int mem(D1 d) { return d.i; }   // error: friendship doesn't inherit
};
// D2 has no access to members in Base
class D2 : public Frnd {
public:
    int mem(Base b) { return b.i; } // error: friendship doesn't inherit
};
```

575

如果派生类想要将自己成员的访问权授予其基类的友元, 派生类必须显式地这样做: 基类的友元对从该基类派生的类型没有特殊访问权限。同样, 如果基类和派生类都需要访问另一个类, 那个类必须特地将访问权限授予基类和每一个派生类。

15.2.7 继承与静态成员

如果基类定义了 static 成员 (12.6 节), 则整个继承层次中只有一个这样的成员。无论从基类派生出多少个派生类, 每个 static 成员只有一个实例。

static 成员遵循常规访问控制: 如果成员在基类中为 private, 则派生类不能访问它。假

定可以访问成员，则既可以通过基类访问 static 成员，也可以通过派生类访问 static 成员。一般而言，既可以使用作用域操作符也可以使用点或箭头成员访问操作符。

```
struct Base {
    static void statmem(); // public by default
};
struct Derived : Base {
    void f(const Derived&);
};
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();      // ok: Base defines statmem
    Derived::statmem();   // ok: Derived inherits statmem
    // ok: derived objects can be used to access static from base
    derived_obj.statmem(); // accessed through Derived object
    statmem();            // accessed through this class
}
```

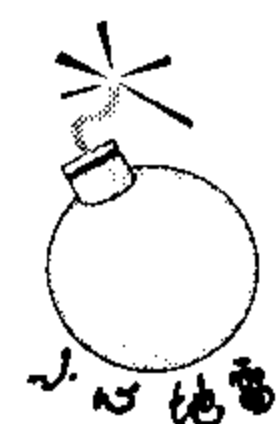
576

习题

习题 15.13 对于下面的类，列出 C1 中的成员函数访问 ConcreteBase 的 static 成员的所有方式，列出 C2 类型的对象访问这些成员的所有方式。

```
struct ConcreteBase {
    static std::size_t object_count();
protected:
    static std::size_t obj_count;
};
struct C1 : public ConcreteBase { /* . . . */ };
struct C2 : public ConcreteBase { /* . . . */ };
```

15.3 转换与继承



理解基类类型和派生类型之间的转换，对于理解面向对象编程在 C++ 中如何工作非常关键。

我们已经看到，每个派生类对象包含一个基类部分，这意味着可以像使用基类对象一样在派生类对象上执行操作。因为派生类对象也是基类对象，所以存在从派生类型引用到基类类型引用的自动转换，即，可以将派生类对象的引用转换为基类子对象的引用，对指针也类似。

基类类型对象既可以作为独立对象存在，也可以作为派生类对象的一部分而存在，因此，一个基类对象可能是也可能不是一个派生类对象的部分，结果，没有从基类引用（或基类指针）到派生类引用（或派生类指针）的（自动）转换。

相对于引用或指针而言，对象转换的情况更为复杂。虽然一般可以使用派生类型的对象对基类类型的对象进行初始化或赋值，但，没有从派生类型对象到基类类型对象的直接转换。

15.3.1 派生类到基类的转换

如果有一个派生类型的对象，则可以使用它的地址对基类类型的指针进行赋值或初始化。同

样，可以使用派生类型的引用或对象初始化基类类型的引用。严格说来，对对象没有类似转换。编译器不会自动将派生类型对象转换为基类类型对象。

577

但是，一般可以使用派生类型对象对基类对象进行赋值或初始化。对对象进行初始化和/或赋值以及可以自动转换引用或指针，这之间的区别是微妙的，必须好好理解。

1. 引用转换不同于转换对象

我们已经看到，可以将派生类型的对象传给希望接受基类引用的函数。也许会因此认为对象进行转换，但是，事实并非如此。将对象传给希望接受引用的函数时，引用直接绑定到该对象，虽然看起来在传递对象，实际上实参是该对象的引用，对象本身未被复制，并且，转换不会在任何方面改变派生类型对象，该对象仍是派生类型对象。

将派生类对象传给希望接受基类类型对象（而不是引用）的函数时，情况完全不同。在这种情况下，形参的类型是固定的——在编译时和运行时形参都是基类类型对象。如果用派生类型对象调用这样的函数，则该派生类对象的基类部分被复制到形参。

一个是将派生类对象转换为基类类型引用，一个是用派生类对象对基类对象进行初始化或赋值，理解它们之间的区别很重要。

2. 用派生类对象对基类对象进行初始化或赋值

对基类对象进行初始化或赋值，实际上是在调用函数：初始化时调用构造函数，赋值时调用赋值操作符。

用派生类对象对基类对象进行初始化或赋值时，有两种可能性。第一种（虽然不太可能的）可能性是，基类可能显式定义了将派生类型对象复制或赋值给基类对象的含义，这可以通过定义适当的构造函数或赋值操作符实现：

```
class Derived;
class Base {
public:
    Base(const Derived&); // create a new Base from a Derived
    Base &operator=(const Derived&); // assign from a Derived
    // ...
};
```

在这种情况下，这些成员的定义将控制用 Drived 对象对 Base 对象进行初始化或赋值时会发生什么。

然而，类显式定义怎样用派生类型对象对基类类型进行初始化或赋值并不常见，相反，基类一般（显式或隐式地）定义自己的复制构造函数和赋值操作符（第 13 章），这些成员接受一个形参，该形参是基类类型的（const）引用。因为存在从派生类引用到基类引用的转换，这些复制控制成员可用于从派生类对象对基类对象进行初始化或赋值：

578

```
Item_base item; // object of base type
Bulk_item bulk; // object of derived type
// ok: uses Item_base::Item_base(const Item_base&) constructor
Item_base item(bulk); // bulk is "sliced down" to its Item_base portion
// ok: calls Item_base::operator=(const Item_base&)
item = bulk; // bulk is "sliced down" to its Item_base portion
```

用 Bulk_item 类型的对象调用 Item_base 类的复制构造函数或赋值操作符时，将发生下列

步骤:

- 将 Bulk_item 对象转换为 Item_base 引用, 这仅仅意味着将一个 Item_base 引用绑定到 Bulk_item 对象。
- 将该引用作为实参传给复制构造函数或赋值操作符。
- 那些操作符使用 Bulk_item 的 Item_base 部分分别对调用构造函数或赋值的 Item_base 对象的成员进行初始化或赋值。
- 一旦操作符执行完毕, 对象即为 Item_base。它包含 Bulk_item 的 Item_base 部分的副本, 但实参的 Bulk_item 部分被忽略。

在这种情况下, 我们说 bulk 的 Bulk_item 部分在对 item 进行初始化或赋值时被“切掉”了。Item_base 对象只包含基类中定义的成员, 不包含由任意派生类型定义的成员, Item_base 对象中没有派生类成员的存储空间。

3. 派生类到基类转换的可访问性

像继承的成员函数一样, 从派生类到基类的转换可能是也可能不是可访问的。转换是否可访问取决于在派生类的派生列表中指定的访问标号。



要确定到基类的转换是否可访问, 可以考虑基类的 public 成员是否可访问, 如果可以, 转换是可访问的, 否则, 转换是不可访问的。

如果是 public 继承, 则用户代码和后代类都可以使用派生类到基类的转换。如果类是使用 private 或 protected 继承派生的, 则用户代码不能将派生类型对象转换为基类对象。如果是 private 继承, 则从 private 继承类派生的类不能转换为基类。如果是 protected 继承, 则后续派生类的成员可以转换为基类类型。

无论是什么派生访问标号, 派生类本身都可以访问基类的 public 成员, 因此, 派生类本身的成员和友元总是可以访问派生类到基类的转换。

579

15.3.2 基类到派生类的转换

从基类到派生类的自动转换是不存在的。需要派生类对象时不能使用基类对象:

```
Item_base base;
Bulk_item* bulkP = &base;    // error: can't convert base to derived
Bulk_item& bulkRef = base;    // error: can't convert base to derived
Bulk_item bulk = base;        // error: can't convert base to derived
```

没有从基类类型到派生类型的(自动)转换, 原因在于基类对象只能是基类对象, 它不能包含派生类型的成员。如果允许用基类对象给派生类型对象赋值, 那么就可以试图使用该派生类对象访问不存在的成员。

有时更令人惊讶的是, 甚至当基类指针或引用实际绑定到派生类对象时, 从基类到派生类的转换也存在限制:

```
Bulk_item bulk;
Item_base *itemP = &bulk;    // ok: dynamic type is Bulk_item
Bulk_item *bulkP = itemP;    // error: can't convert base to derived
```

编译器在编译时无法知道特定转换在运行时实际上是安全的。编译器确定转换是否合法，只看指针或引用的静态类型。

在这些情况下，如果知道从基类到派生类的转换是安全的，就可以使用 `static_cast` (5.12.4 节) 强制编译器进行转换。或者，可以用 `dynamic_cast` 申请在运行时进行检查，18.2.1 节将介绍 `dynamic_cast`。

15.4 构造函数和复制控制

每个派生类对象由派生类中定义的（非 `static`）成员加上一个或多个基类子对象构成，这一事实影响着派生类型对象的构造、复制、赋值和撤销。当构造、复制、赋值和撤销派生类型对象时，也会构造、复制、赋值和撤销这些基类子对象。

构造函数和复制控制成员不能继承，每个类定义自己的构造函数和复制控制成员。像任何类一样，如果类不定义自己的默认构造函数和复制控制成员，就将使用合成版本。

15.4.1 基类构造函数和复制控制

本身不是派生类的基类，其构造函数和复制控制基本上不受继承影响。构造函数看起来像已经见过的许多构造函数一样：

580

```
Item_base(const std::string &book = "",
          double sales_price = 0.0):
    isbn(book), price(sales_price) { }
```

继承对基类构造函数的唯一影响是，在确定提供哪些构造函数时，必须考虑一类新用户。像任意其他成员一样，构造函数可以为 `protected` 或 `private`，某些类需要只希望派生类使用的特殊构造函数，这样的构造函数应定义为 `protected`。

15.4.2 派生类构造函数

派生类的构造函数受继承关系的影响，每个派生类构造函数除了初始化自己的数据成员之外，还要初始化基类。

1. 合成的派生类默认构造函数

派生类的合成默认构造函数（12.4.3 节）与非派生的构造函数只有一点不同：除了初始化派生类的数据成员之外，它还初始化派生类对象的基类部分。基类部分由基类的默认构造函数初始化。

对于 `Bulk_item` 类，合成的默认构造函数会这样执行：

- (1) 调用 `Item_base` 的默认构造函数，将 `isbn` 成员初始化为空串，将 `price` 成员初始化为 0。
- (2) 用常规变量初始化规则初始化 `Bulk_item` 的成员，也就是说，`qty` 和 `discount` 成员会是未初始化的。

2. 定义默认构造函数

因为 `Bulk_item` 具有内置类型成员，所以应定义自己的默认构造函数：

```
class Bulk_item : public Item_base {
public:
    Bulk_item(): min_qty(0), discount(0.0) { }
    // as before
};
```

这个构造函数使用构造函数初始化列表 (7.7.3 节) 初始化 `min_qty` 和 `discount` 成员, 该构造函数还隐式调用 `Item_base` 的默认构造函数初始化对象的基类部分。

运行这个构造函数的效果是, 首先使用 `Item_base` 的默认构造函数初始化 `Item_base` 部分, 那个构造函数将 `isbn` 置为空串并将 `price` 置为 0。 `Item_base` 的构造函数执行完毕后, 再初始化 `Bulk_item` 部分的成员, 并执行构造函数的函数体 (函数体为空)。

581

3. 向基类构造函数传递实参

除了默认构造函数之外, `Item_base` 类还使用户能够初始化 `isbn` 和 `price` 成员, 我们希望支持同样的 `Bulk_item` 对象的初始化, 事实上, 我们希望用户能够指定整个 `Bulk_item` 的值, 包括折扣率和数量。

派生类构造函数的初始化列表只能初始化派生类的成员, 不能直接初始化继承成员。相反, 派生类构造函数通过将基类包含在构造函数初始化列表中来间接初始化继承成员。

```
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book, double sales_price,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        min_qty(qty), discount(disc_rate) { }
    // as before
};
```

这个构造函数使用有两个形参的 `Item_base` 构造函数初始化基类子对象, 它将自己的 `book` 和 `sales_price` 实参传递给该构造函数。这个构造函数可以这样使用:

```
// arguments are the isbn, price, minimum quantity, and discount
Bulk_item bulk("0-201-82470-1", 50, 5, .19);
```

要建立 `bulk`, 首先运行 `Item_base` 构造函数, 该构造函数使用从 `Bulk_item` 构造函数初始化列表传来的实参初始化 `isbn` 和 `price`。 `Item_base` 构造函数执行完毕之后, 再初始化 `Bulk_item` 的成员。最后, 运行 `Bulk_item` 构造函数的 (空) 函数体。



构造函数初始化列表为类的基类和成员提供初始值, 它并不指定初始化的执行次序。首先初始化基类, 然后根据声明次序初始化派生类的成员。

4. 在派生类构造函数中使用默认实参

当然, 也可以将这两个 `Bulk_item` 构造函数编写为一个接受默认实参的构造函数:

```
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book, double sales_price,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
```

582

```

        min_qty(qty), discount(disc_rate) { }
    // as before
};

```

这里为每个形参提供了默认值，因此，可以用 0 至 4 个实参使用该构造函数。

5. 只能初始化直接基类

一个类只能初始化自己的直接基类。直接基类就是在派生列表中指定的类。如果类 C 从类 B 派生，类 B 从类 A 派生，则 B 是 C 的直接基类。虽然每个 C 类对象包含一个 A 类部分，但 C 的构造函数不能直接初始化 A 部分。相反，需要类 C 初始化类 B，而类 B 的构造函数再初始化类 A。这一限制的原因是，类 B 的作者已经指定了怎样构造和初始化 B 类型的对象。像类 B 的任何用户一样，类 C 的作者无权改变这个规约。

作为更具体的例子，书店可以有几种折扣策略。除了批量折扣外，还可以为购买某个数量打折，此后按全价销售，或者，购买量超过一定限度的可以打折，在该限度之内不打折。

这些折扣策略都需要一个数量和一个折扣量，可以定义名为 Disc_item 的新类存储数量和折扣量，以支持这些不同的折扣策略。Disc_item 类可以不定义 net_price 函数，但可以作为定义不同折扣策略的其他类（如 Bulk_item 类）的基类。

关键概念：重构

将 Disc_item 加到 Item_base 层次是重构（refactoring）的一个例子。重构包括重新定义类层次，将操作和/或数据从一个类移到另一个类。为了适应应用程序的需要而重新设计类以便增加新函数或处理其他改变时，最有可能需要进行重构。

重构常见在面向对象应用程序中非常常见。值得注意的是，虽然改变了继承层次，使用 Bulk_item 类或 Item_base 类的代码不需要改变。然而，对类进行重构，或以任意其他方式改变类，使用这些类的任意代码都必须重新编译。

要实现这个设计，首先需要定义 Disc_item 类：

```

// class to hold discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_item : public Item_base {
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        quantity(qty), discount(disc_rate) { }

protected:
    std::size_t quantity; // purchase size for discount to apply
    double discount;      // fractional discount to apply
};

```

583

这个类继承 Item_base 类并定义了自己的 discount 和 quantity 成员。它唯一的成员函数是构造函数，用以初始化基类和 Disc_item 定义的成员。

其次，可以重新实现 Bulk_item 以继承 Disc_item，而不再直接继承 Item_base：

```
// discount kicks in when a specified number of copies of same book are sold
// the discount is expressed as a fraction to use to reduce the normal price
class Bulk_item : public Disc_item {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) {}
    // redefines base version so as to implement bulk purchase discount policy
    double net_price(std::size_t) const;
};
```

Bulk_item 类现在有一个直接基类 Disc_item, 还有一个间接基类 Item_base。每个 Bulk_item 对象有三个子对象: 一个 (空的) Bulk_item 部分和一个 Disc_item 子对象, Disc_item 子对象又有一个 Item_base 基类子对象。

虽然 Bulk_item 没有自己的数据成员, 但为了获取值用来初始化其继承成员, 它定义了一个构造函数。

派生类构造函数只能初始化自己的直接基类, 在 Bulk_item 类的构造函数初始化列表中指定 Item_base 是一个错误。

关键概念: 尊重基类接口

构造函数只能初始化其直接基类的原因是每个类都定义了自己的接口。定义 Disc_item 时, 通过定义它的构造函数指定了怎样初始化 Disc_item 对象。一旦类定义了自己的接口, 与该类对象的所有交互都应该通过该接口, 即使对象是派生类对象的一部分也不例外。

同样, 派生类构造函数不能初始化基类的成员且不应该对基类成员赋值。如果那些成员为 public 或 protected, 派生构造函数可以在构造函数函数体中给基类成员赋值, 但是, 这样做会违反基类的接口。派生类应通过使用基类构造函数尊重基类的初始化意图, 而不是在派生类构造函数函数体中对这些成员赋值。

习题

习题 15.14 重新定义 Bulk_item 和 Item_base 类, 使每个类只需定义一个构造函数。

习题 15.15 对于 15.2.5 节习题第一题中描述的图书馆类层次, 识别基类和派生类构造函数。

习题 15.16 对于下面的基类定义:

```
struct Base {
    Base(int val): id(val) {}
protected:
    int id;
};
```

解释为什么下述每个构造函数是非法的。

- (a) struct C1 : public Base {
 C1(int val): id(val) {}
 };
 (b) struct C2 : public C1 {
 C2(int val): Base(val), C1(val) {}
 }

```

};
(c) struct C3 : public C1 {
    C3(int val): Base(val) { }
};
(d) struct C4 : public Base {
    C4(int val) : Base(id + val){ }
};
(e) struct C5 : public Base {
    C5() { }
};

```

15.4.3 复制控制和继承

像任意其他类一样，派生类也可以使用第 13 章所介绍的合成复制控制成员。合成操作对对象的基类部分连同派生部分的成员一起进行复制、赋值或撤销，使用基类的复制构造函数、赋值操作符或析构函数对基类部分进行复制、赋值或撤销。

584

类是否需要定义复制控制成员完全取决于类自身的直接成员。基类可以定义自己的复制控制而派生类使用合成版本，反之亦然。

只包含类类型或内置类型数据成员、不含指针的类一般可以使用合成操作，复制、赋值或撤销这样的成员不需要特殊控制。具有指针成员的类一般需要定义自己的复制控制来管理这些成员。

Item_base 类及其派生类可以使用复制控制操作的合成版本。复制 Bulk_item 对象时，调用（合成的）Item_base 复制构造函数复制 isbn 和 price 成员。使用 string 复制构造函数复制 isbn，直接复制 price 成员。一旦复制了基类部分，就复制派生部分。Bulk_item 的两个成员都是 double 型，直接复制这些成员。赋值操作符和析构函数类似处理。

585

1. 定义派生类复制构造函数



如果派生类显式定义自己的复制构造函数或赋值操作符，则该定义将完全覆盖默认定义。被继承类的复制构造函数和赋值操作符负责对基类成分以及类自己的成员进行复制或赋值。

如果派生类定义了自己的复制构造函数，该复制构造函数一般应显式使用基类复制构造函数初始化对象的基类部分：

```

class Base { /* ... */ };
class Derived: public Base {
public:
    // Base::Base(const Base&) not invoked automatically
    Derived(const Derived& d):
        Base(d) /* other member initialization */ { /*... */ }
};

```

初始化函数 Base(d) 将派生类对象 d 转换（15.3 节）为它的基类部分的引用，并调用基类复制构造函数。如果省略基类初始化函数，如下代码：

```

// probably incorrect definition of the Derived copy constructor
Derived(const Derived& d) /* derived member initializations */
{ /* ... */ }

```

效果是运行 Base 的默认构造函数初始化对象的基类部分。假定 Derived 成员的初始化从 d 复制对应成员，则新构造的对象将具有奇怪的配置：它的 Base 部分将保存默认值，而它的 Derived 成员是另一对象的副本。

2. 派生类赋值操作符

赋值操作符通常与复制构造函数类似：如果派生类定义了自己的赋值操作符，则该操作符必须对基类部分进行显式赋值。

```
// Base::operator=(const Base&) not invoked automatically
Derived &Derived::operator=(const Derived &rhs)
{
    if (this != &rhs) {
        Base::operator=(rhs); // assigns the base part
        // do whatever needed to clean up the old value in the derived part
        // assign the members from the derived
    }
    return *this;
}
```

586

赋值操作符必须防止自身赋值。假定左右操作数不同，则调用 Base 类的赋值操作符给基类部分赋值。该操作符可以由类定义，也可以是合成赋值操作符，这没什么关系——我们可以直接调用它。基类操作符将释放左操作数中基类部分的值，并赋以来自 rhs 的新值。该操作符执行完毕后，接着要做的是为派生类中的成员赋值。

3. 派生类析构函数

析构函数的工作与复制构造函数和赋值操作符不同：派生类析构函数不负责撤销基类对象的成员。编译器总是显式调用派生类对象基类部分的析构函数。每个析构函数只负责清除自己的成员：

```
class Derived: public Base {
public:
    // Base::~Base invoked automatically
    ~Derived() { /* do what it takes to clean up derived members */ }
};
```

对象的撤销顺序与构造顺序相反：首先运行派生类析构函数，然后按继承层次依次向上调用各基类析构函数。

15.4.4 虚析构函数

自动调用基类部分的析构函数对基类的设计有重要影响。

删除指向动态分配对象的指针时，需要运行析构函数在释放对象的内存之前清除对象。处理继承层次中的对象时，指针的静态类型可能与被删除对象的动态类型不同，可能会删除实际指向派生类对象的基类类型指针。

如果删除基类指针，则需要运行基类析构函数并清除基类的成员，如果对象实际是派生类型的，则没有定义该行为。要保证运行适当的析构函数，基类中的析构函数必须为虚函数：

```
class Item_base {
```

```
public:
    // no work, but virtual destructor needed
    // if base pointer that points to a derived object is ever deleted
    virtual ~Item_base() { }
};
```

587

如果析构函数为虚函数，那么通过指针调用时，运行哪个析构函数将因指针所指对象类型的不同而不同：

```
Item_base *itemP = new Item_base; // same static and dynamic type
delete itemP;                      // ok: destructor for Item_base called
itemP = new Bulk_item;             // ok: static and dynamic types differ
delete itemP;                      // ok: destructor for Bulk_item called
```

像其他虚函数一样，析构函数的虚函数性质都将继承。因此，如果层次中根类的析构函数为虚函数，则派生类析构函数也将是虚函数，无论派生类显式定义析构函数还是使用合成析构函数，派生类析构函数都是虚函数。

基类析构函数是三法则（13.3 节）的一个重要例外。三法则指出，如果类需要析构函数，则类几乎也确实需要其他复制控制成员。基类几乎总是需要构造函数，从而可以将析构函数设为虚函数。如果基类为了将析构函数设为虚函数而具有空析构函数，那么，类具有析构函数并不表示也需要赋值操作符或复制构造函数。



最佳
实践

即使析构函数没有工作要做，继承层次的根类也应该定义一个虚析构函数。

构造函数和赋值操作符不是虚函数

在复制控制成员中，只有析构函数应定义为虚函数，构造函数不能定义为虚函数。构造函数是在对象完全构造之前运行的，在构造函数运行的时候，对象的动态类型还不完整。

虽然可以在基类中将成员函数 `operator=` 定义为虚函数，但这样做并不影响派生类中使用的赋值操作符。每个类有自己的赋值操作符，派生类中的赋值操作符有一个与类本身类型相同的形参，该类型必须不同于继承层次中任意其他类的赋值操作符的形参类型。

将赋值操作符设为虚函数可能会令人混淆，因为虚函数必须在基类和派生类中具有同样的形参。基类赋值操作符有一个形参是自身类类型的引用，如果该操作符为虚函数，则每个类都将得到一个虚函数成员，该成员定义了参数为一个基类对象的 `operator=`。但是，对派生类而言，这个操作符与赋值操作符是不同的。



588

将类的赋值操作符设为虚函数很可能会令人混淆，而且不会有什么用处。

习题

习题 15.17 说明在什么情况下类应该具有虚析构函数。

习题 15.18 虚析构函数必须执行什么操作？

习题 15.19 如果这个类定义有错，可能是什么错？

```
class AbstractObject {
public:
    virtual void doit();
    // other members not including any of the copy-control functions
};
```

习题 15.20 回忆在 13.3 节习题中编写的类，该类的复制控制成员打印一条消息，为 `Item_base` 和 `Bulk_item` 类的构造函数增加打印语句。定义复制控制成员，使之完成与合成版本相同的工作外，还打印一条消息。应用使用了 `Item_base` 类型的那些对象和函数编写一些程序，在每种情况下，预测将会创建和撤销什么对象，并将你的预测与程序所产生的结果进行比较。继续实验，直至你能够正确地预测对于给定的代码片段，会执行哪些复制控制成员。

15.4.5 构造函数和析构函数中的虚函数

构造派生类对象时首先运行基类构造函数初始化对象的基类部分。在执行基类构造函数时，对象的派生类部分是未初始化的。实际上，此时对象还不是一个派生类对象。

撤销派生类对象时，首先撤销它的派生类部分，然后按照与构造顺序的逆序撤销它的基类部分。

在这两种情况下，运行构造函数或析构函数的时候，对象都是不完整的。为了适应这种不完整，编译器将对象的类型视为在构造或析构期间发生了变化。在基类构造函数或析构函数中，将派生类对象当作基类类型对象对待。

构造或析构期间的对象类型对虚函数的绑定有影响。



如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本。

无论由构造函数（或析构函数）直接调用虚函数，或者从构造函数（或析构函数）所调用的函数间接调用虚函数，都应用这种绑定。

要理解这种行为，考虑如果从基类构造函数（或析构函数）调用虚函数的派生类版本会怎么样。虚函数的派生类版本很可能会访问派生类对象的成员，毕竟，如果派生类版本不需要使用派生类对象的成员，派生类多半能够使用基类中的定义。但是，对象的派生部分的成员不会在基类构造函数运行期间初始化，实际上，如果允许这样的访问，程序很可能会崩溃。

589

15.5 继承情况下的类作用域

每个类都保持着自己的作用域（12.3 节），在该作用域中定义了成员的名字。在继承情况下，派生类的作用域嵌套在基类作用域中。如果不能在派生类作用域中确定名字，就在外围基类作用域中查找该名字的定义。

正是这种类作用域的层次嵌套使我们能够直接访问基类的成员，就好象这些成员是派生类成员一样。如果编写如下代码：

```
Bulk_item bulk;
cout << bulk.book();
```

名字 book 的使用将这样确定：

- (1) bulk 是 Bulk_item 类对象，在 Bulk_item 类中查找，找不到名字 book。
- (2) 因为从 Item_base 派生 Bulk_item，所以接着在 Item_base 类中查找，找到名字 book，引用成功地确定了。

15.5.1 名字查找在编译时发生

对象、引用或指针的静态类型决定了对象能够完成的行为。甚至当静态类型和动态类型可能不同的时候，就像使用基类类型的引用或指针时可能会发生的，静态类型仍然决定着可以使用什么成员。例如，可以给 Disc_item 类增加一个成员，该成员返回一个保存最小（或最大）数量和折扣价格的 pair 对象：

```
class Disc_item : public Item_base {
public:
    std::pair<size_t, double> discount_policy() const
    { return std::make_pair(quantity, discount); }
    // other members as before
};
```

590 只能通过 Disc_item 类型或 Disc_item 派生类型的对象、指针或引用访问 discount_policy：

```
Bulk_item bulk;
Bulk_item *bulkP = &bulk; // ok: static and dynamic types are the same
Item_base *itemP = &bulk; // ok: static and dynamic types differ
bulkP->discount_policy(); // ok: bulkP has type Bulk_item*
itemP->discount_policy(); // error: itemP has type Item_base*
```

通过 itemP 的访问是错误的，因为基类类型的指针（引用或对象）只能访问对象的基类部分，而在基类中没有定义 discount_policy 成员。

习题

习题 15.21 重新定义 Item_base 层次以包含 Disc_item 类。

习题 15.22 重新定义 Bulk_item 和你在 15.2.3 节习题中实现的那个继承 Disc_item 的表示有限折扣策略的类。

15.5.2 名字冲突与继承

虽然可以直接访问基类成员，就像它是派生类成员一样，但是成员保留了它的基类成员资格。一般我们并不关心是哪个实际类包含成员，通常只在基类和派生类共享同一名字时才需要注意。



与基类成员同名的派生类成员将屏蔽对基类成员的直接访问。

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
```

```
};
struct Derived : Base {
    Derived(int i): mem(i) { } // initializes Derived::mem
    int get_mem() { return mem; } // returns Derived::mem
protected:
    int mem; // hides mem in the base
};
```

get_mem 中对 mem 的引用被确定为使用 Derived 中的名字。如果编写如下代码:

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

则输出将是 42。

591

使用作用域操作符访问被屏蔽成员

可以使用作用域操作符访问被屏蔽的基类成员:

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

作用域操作符指示编译器在 Base 中查找 mem。



最佳
实践

设计派生类时, 只要可能, 最好避免与基类成员的名字冲突。

习题

习题 15.23 对于下面的基类和派生类定义:

```
struct Base {
    foo(int);
protected:
    int bar;
    double foo_bar;
};

struct Derived : public Base {
    foo(string);
    bool bar(Base *pb);
    void foobar();
protected:
    string bar;
};
```

找出下述每个例子中的错误并说明怎样改正:

- (a) `Derived d; d.foo(1024);`
- (b) `void Derived::foobar() { bar = 1024; }`
- (c) `bool Derived::bar(Base *pb)`
`{ return foo_bar == pb->foo_bar; }`

15.5.3 作用域与成员函数

在基类和派生类中使用同一名字的成员函数, 其行为与数据成员一样: 在派生类作用域中派生类成员将屏蔽基类成员。即使函数原型不同, 基类成员也会被屏蔽:

592

```

struct Base {
    int memfcn();
};
struct Derived : Base {
    int memfcn(int); // hides memfcn in the base
};
Derived d; Base b;
b.memfcn();          // calls Base::memfcn
d.memfcn(10);         // calls Derived::memfcn
d.memfcn();           // error: memfcn with no arguments is hidden
d.Base::memfcn();     // ok: calls Base::memfcn

```

Derived 中的 memfcn 声明隐藏了 Base 中的声明。这并不奇怪，第一个调用通过 Base 对象 b 调用基类中的版本，同样，第二个调用通过 d 调用 Derived 中的版本。可能比较奇怪的是第三个调用：

```
d.memfcn(); // error: Derived has no memfcn that takes no arguments
```

要确定这个调用，编译器需要查找名字 memfcn，并在 Derived 类中找到。一旦找到了名字，编译器就不再继续查找了。这个调用与 Derived 中的 memfcn 定义不匹配，该定义希望接受 int 实参，而这个函数调用没有提供那样的实参，因此出错。



回忆一下，局部作用域中声明的函数不会重载全局作用域中定义的函数(7.8.1 节)，同样，派生类中定义的函数也不重载基类中定义的成员。通过派生类对象调用函数时，实参必须与派生类中定义的版本相匹配，只有在派生类根本没有定义该函数时，才考虑基类函数。

重载函数

像其他任意函数一样，成员函数（无论虚还是非虚）也可以重载。派生类可以重定义所继承的 0 个或多个版本。



如果派生类重定义了重载成员，则通过派生类型只能访问派生类中重定义的那些成员。

如果派生类想通过自身类型使用所有的重载版本，则派生类必须要么重定义所有重载版本，要么一个也不重定义。

有时类需要仅仅重定义一个重载集中某些版本的行为，并且想要继承其他版本的含义，在这种情况下，为了重定义需要特化的某个版本而不得不重定义每一个基类版本，可能会令人厌烦。

派生类不用重定义所继承的每一个基类版本，它可以为重载成员提供 using 声明(15.2.5 节)。一个 using 声明只能指定一个名字，不能指定形参表，因此，为基类成员函数名称而作的 using 声明将该函数的所有重载实例加到派生类的作用域。将所有名字加入作用域之后，派生类只需要重定义本类型确实必须定义的那些函数，对其他版本可以使用继承的定义。

593

15.5.4 虚函数与作用域

还记得吗，要获得动态绑定，必须通过基类的引用或指针调用虚成员。当我们这样做时，编

译器将在基类中查找函数。假定找到了名字，编译器就检查实参是否与形参匹配。

现在可以理解虚函数为什么必须在基类和派生类中拥有同一原型了。如果基类成员与派生类成员接受的实参不同，就没有办法通过基类类型的引用或指针调用派生类函数。考虑如下（人为的）类集合：

```
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    int fcn(int); // parameter list differs from fcn in Base
    // D1 inherits definition of Base::fcn()
};
class D2 : public D1 {
public:
    int fcn(int); // nonvirtual function hides D1::fcn(int)
    int fcn();    // redefines virtual fcn from Base
};
```

D1 中的 fcn 版本没有重定义 Base 的虚函数 fcn，相反，它屏蔽了基类的 fcn。结果 D1 有两个名为 fcn 的函数：类从 Base 继承了一个名为 fcn 的虚函数，类又定义了自己的名为 fcn 的非虚成员函数，该函数接受一个 int 形参。但是，从 Base 继承的虚函数不能通过 D1 对象（或 D1 的引用或指针）调用，因为该函数被 fcn(int) 的定义屏蔽了。

类 D2 重定义了它继承的两个函数，它重定义了 Base 中定义的 fcn 的原始版本并重定义了 D1 中定义的非虚版本。

通过基类调用被屏蔽的虚函数

通过基类类型的引用或指针调用函数时，编译器将在基类中查找该函数而忽略派生类：

```
Base bobj; D1 d1obj; D2 d2obj;
Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
bp1->fcn(); // ok: virtual call, will call Base::fcn at run time
bp2->fcn(); // ok: virtual call, will call Base::fcn at run time
bp3->fcn(); // ok: virtual call, will call D2::fcn at run time
```

594

三个指针都是基类类型的指针，因此通过在 Base 中查找 fcn 来确定这三个调用，所以这些调用是合法的。另外，因为 fcn 是虚函数，所以编译器会生成代码，在运行时基于引用或指针所绑定的对象的实际类型进行调用。在 bp2 的情况，基本对象是 D1 类的，D1 类没有重定义不接受实参的虚函数版本，通过 bp2 的函数调用（在运行时）调用 Base 中定义的版本。

关键概念：名字查找与继承

理解 C++ 中继承层次的关键在于理解如何确定函数调用。确定函数调用遵循以下四个步骤：

- (1) 首先确定进行函数调用的对象、引用或指针的静态类型。
- (2) 在该类中查找函数，如果找不到，就在直接基类中查找，如此循着类的继承链往上找，直到找到该函数或者查找完最后一个类。如果不能在类或其相关基类中找到该名字，则调用是

错误的。

(3) 一旦找到了该名字, 就进行常规类型检查 (7.1.2 节), 查看如果给定找到的定义, 该函数调用是否合法。

(4) 假定函数调用合法, 编译器就生成代码。如果函数是虚函数且通过引用或指针调用, 则编译器生成代码以确定根据对象的动态类型运行哪个函数版本, 否则, 编译器生成代码直接调用函数。

习题

习题 15.24 对于如下代码:

```
Bulk_item bulk;
Item_base item(bulk);
Item_base *p = &bulk;
```

为什么表达式

```
p->net_price(10);
```

调用 net_price 的 Bulk_item 实例, 而表达式

```
item.net_price(10);
```

调用 Item_base 实例?

习题 15.25 假定 Derived 继承 Base, 并且 Base 将下面的函数定义为虚函数, 假定 Derived 打算定义自己的这个虚函数的版本, 确定在 Derived 中哪个声明是错误的, 并指出为什么错。

- (a) Base* Base::copy(Base*);
Base* Derived::copy(Derived*);
- (b) Base* Base::copy(Base*);
Derived* Derived::copy(Base*);
- (c) ostream& Base::print(int, ostream&=cout);
ostream& Derived::print(int, ostream&);
- (d) void Base::eval() const;
void Derived::eval();

15.6 纯虚函数

在 15.4.2 节所编写的 Disc_item 类提出了一个有趣的问题: 该类从 Item_base 继承了 net_price 函数但没有重定义该函数。因为对 Disc_item 类而言没有可以给予该函数的意义, 所以没有重定义该函数。在我们的应用程序中, Disc_item 不对应任何折扣策略, 这个类的存在只是为了让其他类继承。

我们不想让用户定义 Disc_item 对象, 相反, Disc_item 对象只应该作为 Disc_item 派生类型的对象的一部分而存在。但是, 正如已定义的, 没有办法防止用户定义一个普通的 Disc_item 对象。这带来一个问题: 如果用户创建一个 Disc_item 对象并调用该对象的 net_price 函数, 会发生什么呢? 从前面章节的讨论中了解到, 结果将是调用从 Item_base 继承而来的 net_price 函数, 该函数产生的是不打折的价格。

595

很难说用户可能期望调用 Disc_item 的 net_price 会有什么样的行为。真正的问题在于, 我们宁愿用户根本不能创建这样的对象。可以使 net_price 成为纯虚函数 (pure virtual function),

强制实现这一设计意图并正确指出 `Disc_item` 的 `net_price` 版本没有意义。在函数形参表后面写上 `=0` 以指定纯虚函数：

```
class Disc_item : public Item_base {
public:
    double net_price(std::size_t) const = 0;
};
```

将函数定义为纯虚能够说明，该函数为后代类型提供了可以覆盖的接口，但是这个类中的版本决不会调用。重要的是，用户将不能创建 `Disc_item` 类型的对象。

试图创建抽象基类的对象将发生编译时错误：

```
// Disc_item declares pure virtual functions
Disc_item discounted; // error: can't define a Disc_item object
Bulk_item bulk;       // ok: Disc_item subobject within Bulk_item
```

596



含有（或继承）一个或多个纯虚函数的类是**抽象基类**（abstract base class）。除了作为抽象基类的派生类的对象的组成部分，不能创建抽象类型的对象。

习题

习题 15.26 使你的 `Disc_item` 类版本成为抽象类。

习题 15.27 试试定义 `Disc_item` 类型的一个对象，看看会从编译器得到什么错误。

15.7 容器与继承

我们希望使用容器（或内置数组）保存因继承而相关联的对象。但是，对象不是多态的（15.3.1 节），这一事实对将容器用于继承层次中的类型有影响。

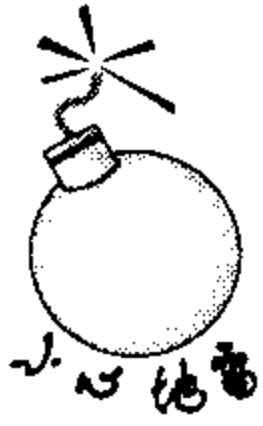
例如，书店应用程序中可能有购物篮，购物篮代表顾客正在购买的书籍。我们希望能够将 `multiset`（10.5 节）中存储购买物，要定义 `multiset`，必须指定容器将保存的对象的类型。将对象放进容器时，复制元素（9.3.3 节）。

如果定义 `multiset` 保存基类类型的对象：

```
multiset<Item_base> basket;
Item_base base;
Bulk_item bulk;
basket.insert(base); // ok: add copy of base to basket
basket.insert(bulk); // ok: but bulk sliced down to its base part
```

则加入派生类型的对象时，只将对象的基类部分保存在容器中。记住，将派生类对象复制到基类对象时，派生类对象将被切掉（15.3.1 节）。

容器中的元素是 `Item_base` 对象，无论元素是否作为 `Bulk_item` 对象的副本而建立，当计算元素的 `net_price` 时，元素将按不打折定价。一旦对象放入了 `multiset`，它就不再是派生类对象了。



因为派生类对象在赋值给基类对象时会被“切掉”，所以容器与通过继承相关的类型不能很好地融合。

597

不能通过定义容器保存派生类对象来解决这个问题。在这种情况下，不能将 `Item_base` 对象放入容器——没有从基类类型到派生类型的标准转换。可以显式地将基类对象强制转换为派生类对象并将结果对象加入容器，但是，如果这样做，当试图使用这样的元素时，会产生大问题：在这种情况下，元素可以当作派生类对象对待，但派生类部分的成员将是未初始化的。

唯一可行的选择可能是使用容器保存对象的指针。这个策略可行，但代价是需要用户面对管理对象和指针的问题，用户必须保证只要容器存在，被指向的对象就存在。如果对象是动态分配的，用户必须保证在容器消失时适当地释放对象。下一节将介绍对这个问题更好更通用的解决方案。

习题

习题 15.28 定义一个 `vector` 保存 `Item_base` 类型的对象，并将一些 `Bulk_item` 类型对象复制到 `vector` 中。遍历并计算容器中元素的总和。

习题 15.29 重复程序，但这次存储 `Item_base` 类型对象的指针。比较结果总和。

习题 15.30 解释上两题程序所产生总和的差异。如果没有差异，解释为什么没有。

15.8 句柄类与继承

C++中面向对象编程的一个颇具讽刺意味的地方是，不能使用对象支持面向对象编程，相反，必须使用指针或引用。例如，下面的代码段中：

```
void get_prices(Item_base object,
               const Item_base *pointer,
               const Item_base &reference)
{
    // which version of net_price is called is determined at run time
    cout << pointer->net_price(1) << endl;
    cout << reference.net_price(1) << endl;

    // always invokes Item_base::net_price
    cout << object.net_price(1) << endl;
}
```

通过 `pointer` 和 `reference` 进行的调用在运行时根据它们所绑定对象的动态类型而确定。

598

但是，使用指针或引用会加重类用户的负担。在前一节中讨论继承类型对象与容器的相互作用时，已经碰到了一种这样的负担。

C++中一个通用的技术是定义包装（`cover`）类或句柄（`handle`）类。句柄类存储和管理基类指针。指针所指对象的类型可以变化，它既可以指向基类类型对象又可以指向派生类型对象。用户通过句柄类访问继承层次的操作。因为句柄类使用指针执行操作，虚成员的行为将在运行时根据句柄实际绑定的对象的类型而变化。因此，句柄的用户可以获得动态行为但无须操心指针的

管理。

包装了继承层次的句柄有两个重要的设计考虑因素：

- 像对任何保存指针（13.5 节）的类一样，必须确定对复制控制做些什么。包装了继承层次的句柄通常表现得像一个智能指针（13.5.1 节）或者像一个值（13.5.2 节）。
- 句柄类决定句柄接口屏蔽还是不屏蔽继承层次，如果不屏蔽继承层次，用户必须了解和使用基本层次中的对象。

对于这些选项没有正确的选择，决定取决于继承层次的细节，以及类设计者希望程序员如何与那些类相互作用。下面两节将实现两种不同的句柄，用不同的方式解决这些设计问题。

15.8.1 指针型句柄

像第一个例子一样，我们将定义一个名为 `Sales_item` 的指针型句柄类，表示 `Item_base` 层次。`Sales_item` 的用户将像使用指针一样使用它：用户将 `Sales_item` 绑定到 `Item_base` 类型的对象并使用 `*` 和 `->` 操作符执行 `Item_base` 的操作：

```
// bind a handle to a Bulk_item object
Sales_item item(Bulk_item("0-201-82470-1", 35, 3, .20));

item->net_price(); // virtual call to net_price function
```

但是，用户不必管理句柄指向的对象，`Sales_item` 类将完成这部分工作。当用户通过 `Sales_item` 类对象调用函数时，将获得多态行为。

1. 定义句柄

`Sales_item` 类有三个构造函数：默认构造函数、复制构造函数和接受 `Item_base` 对象的构造函数。第三个构造函数将复制 `Item_base` 对象，并保证：只要 `Sales_item` 对象存在副本就存在。当复制 `Sales_item` 对象或给 `Sales_item` 对象赋值时，将复制指针而不是复制对象。像对其他指针型句柄类一样，将用使用计数来管理副本。

迄今为止，我们已经使用过的使用计数式类，都使用一个伙伴类来存储指针和相关的使用计数。这个例子将使用不同的设计，如图 15-2 所示。`Sales_item` 类将有两个数据成员，都是指针：一个指针将指向 `Item_base` 对象，而另一个将指向使用计数。`Item_base` 指针可以指向 `Item_base` 对象也可以指向 `Item_base` 派生类型的对象。通过指向使用计数，多个 `Sales_item` 对象可以共享同一计数器。

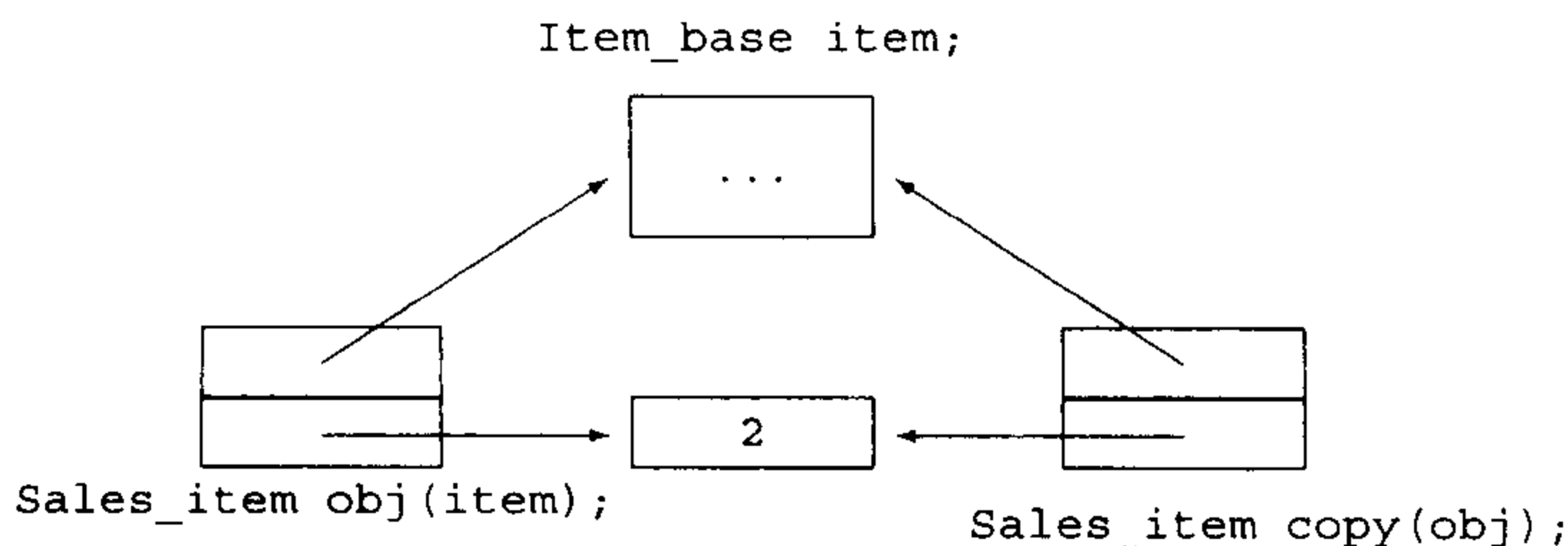


图 15-2 `Sales_item` 句柄类的使用计数策略

除了管理使用计数之外，Sales_item 类还将定义解引用操作符和箭头操作符：

```
// use counted handle class for the Item_base hierarchy
class Sales_item {
public:
    // default constructor: unbound handle
    Sales_item(): p(0), use(new std::size_t(1)) { }
    // attaches a handle to a copy of the Item_base object
    Sales_item(const Item_base&);
    // copy control members to manage the use count and pointers
    Sales_item(const Sales_item &i):
        p(i.p), use(i.use) { ++*use; }
    ~Sales_item() { decr_use(); }
    Sales_item& operator=(const Sales_item&);
    // member access operators
    const Item_base *operator->() const { if (p) return p;
        else throw std::logic_error("unbound Sales_item"); }
    const Item_base &operator*() const { if (p) return *p;
        else throw std::logic_error("unbound Sales_item"); }
private:
    Item_base *p;           // pointer to shared item
    std::size_t *use;       // pointer to shared use count
    // called by both destructor and assignment operator to free pointers
    void decr_use()
        { if (--*use == 0) { delete p; delete use; } }
};
```

600

2. 使用计数式复制控制

复制控制成员适当地操纵使用计数和 Item_base 指针。复制 Sales_item 对象包括复制两个指针和将使用计数加 1。析构函数将使用计数减 1，如果计数减至 0 就撤销指针。因为赋值操作符需要完成同样的工作，所以在名为 decr_use 的私有实用函数中实现析构函数的行为。

赋值操作符比复制构造函数复杂一点：

```
// use-counted assignment operator; use is a pointer to a shared use count
Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{
    ++*rhs.use;
    decr_use();
    p = rhs.p;
    use = rhs.use;
    return *this;
}
```

赋值操作符像复制构造函数一样，将右操作数的使用计数加 1 并复制指针；它也像析构函数一样，首先必须将左操作数的使用计数减 1，如果使用计数减至 0 就删除指针。

像通常对赋值操作符一样，必须防止自身赋值。这个操作符通过首先将右操作数的使用计数减 1 来处理自身赋值。如果左右操作数相同，则调用 decr_use 时使用计数将至少为 2。该函数将左操作数的使用计数减 1 并进行检查，如果使用计数减至 0，则 decr_use 将释放该对象中的 Item_base 对象和 use 对象。剩下的是从右操作数向左操作数复制指针，像平常一样，我们的赋值操作符返回左操作数的引用。

除了复制控制成员以外, Sales_item 定义的其他函数是操作符函数 operator* 和 operator->, 用户将通过这些操作符访问 Item_base 成员。因为这两个操作符分别返回指针和引用, 所以通过这些操作符调用的函数将进行动态绑定。

我们只定义了这些操作符的 const 版本, 因为基础 Item_base 层次中的成员都是 const 成员。

3. 构造句柄

我们的句柄有两个构造函数: 默认构造函数创建未绑定的 Sales_item 对象, 第二个构造函数接受一个对象, 将句柄与其关联。

第一个构造函数容易定义: 将 Item_base 指针置 0 以指出该句柄没有关联任何对象上。构造函数分配一个新的计数器并将它初始化为 1。

第二个构造函数难一点, 我们希望句柄的用户创建自己的对象, 在这些对象上关联句柄。构造函数将分配适当类型的新对象并将形参复制到新分配的对象中, 这样, Sales_item 类将拥有对象并能够保证在关联到该对象的最后一个 Sales_item 对象消失之前不会删除对象。

601

15.8.2 复制未知类型

要实现接受 Item_base 对象的构造函数, 必须首先解决一个问题: 我们不知道给予构造函数的对象的实际类型。我们知道它是一个 Item_base 对象或者是一个 Item_base 派生类型的对象。句柄类经常需要在不知道对象的确切类型时分配已知对象的新副本。Sales_item 构造函数是个好例子。



解决这个问题通用方法是定义虚操作进行复制, 我们称将该操作命名为 clone。

为了支持句柄类, 需要从基类开始, 在继承层次的每个类型中增加 clone, 基类必须将该函数定义为虚函数:

```
class Item_base {
public:
    virtual Item_base* clone() const
        { return new Item_base(*this); }
};
```

每个类必须重定义该虚函数。因为函数的存在是为了生成类对象的新副本, 所以定义返回类型为类本身:

```
class Bulk_item : public Item_base {
public:
    Bulk_item* clone() const
        { return new Bulk_item(*this); }
};
```

15.2.3 节介绍过, 对于派生类的返回类型必须与基类实例的返回类型完全匹配的要求, 但有一个例外。这个例外支持像这个类这样的情况。如果虚函数的基类实例返回类类型的引用或指针, 则该虚函数的派生类实例可以返回基类实例返回的类型的派生类(或者是类类型的指针或引用)。

定义句柄构造函数

一旦有了 clone 函数，就可以这样编写 Sales_item 构造函数：

```
Sales_item::Sales_item(const Item_base &item):
    p(item.clone()), use(new std::size_t(1)) { }
```

602

像默认构造函数一样，这个构造函数分配并初始化使用计数，它调用形参的 clone 产生那个对象的（虚）副本。如果实参是 Item_base 对象，则运行 Item_base 的 clone 函数；如果实参是 Bulk_item 对象，则执行 Bulk_item 的 clone 函数。

习题

习题 15.31 为 15.2.3 节的习题中实现的有限折扣类定义和实现 clone 操作。

习题 15.32 实际上，程序不太可能在第一次运行或第一次用真实数据运行时就能正确运行。在类的设计中包括调试策略经常是有用的。为 Item_base 类层次实现一个 debug 虚函数，显示各个类的数据成员。

习题 15.33 对于 Item_base 层次的包括 Disc_item 抽象基类的版本，指出 Disc_item 类是否应实现 clone 函数，为什么？

习题 15.34 修改调试函数以允许用户打开或关闭调试。用两种方式实现控制：

(a) 通过定义 debug 函数的形参。

(b) 通过定义类数据成员。该成员允许个体对象打开或关闭调试信息的显示。

15.8.3 句柄的使用

使用 Sales_item 对象可以更容易地编写书店应用程序。代码将不必管理 Item_base 对象的指针，但仍然可以获得通过 Sales_item 对象进行的调用的虚行为。

例如，可以使用 Item_base 对象解决 15.7 节提出的问题。可以使用 Sales_item 对象跟踪顾客所做的购买，在 multiset 中保存一个对象表示一次购买，当顾客完成购买时，可以计算销售总数。

1. 比较两个 Sales_item 对象

在编写函数计算销售总数之前，需要定义比较 Sales_item 对象的方法。要用 Sales_item 作为关联容器的关键字，必须能够比较它们（10.3.1 节）。关联容器默认使用关键字类型的小于操作符，但是，基于 14.3.2 节讨论过的有关原始 Sales_item 类型的同样理由，为 Sales_item 句柄类定义 operator< 可能是个坏主意：当使用 Sales_item 作关键字时，只想考虑 ISBN，但确定相等时又想要考虑所有数据成员。

603

幸好，关联容器使我们能够指定一个函数[或函数对象（14.8 节）]用作比较函数，这样做类似于 11.2.3 节中将单独函数传给 stable_sort 算法的方式。在那种情况下，只需要将附加的实参传给 stable_sort 以提供比较函数，代替<操作符的使用。覆盖关联容器的比较函数有点复杂，因为，正如我们将看到的，在定义容器对象时必须提供比较函数。

让我们从容易的部分开始，定义一个函数用于比较 Sales_item 对象：

```
// compare defines item ordering for the multiset in Basket
inline bool
compare(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs->book() < rhs->book();
}
```

我们的 `compare` 函数与小于操作符有同样的接口，它接受两个 `Sales_item` 对象的 `const` 引用，通过比较 ISBN 而比较形参，返回一个 `bool` 值。该函数使用 `Sales_item` 的 `->` 操作符，该操作符返回 `Item_base` 对象的指针，那个指针用于获取并运行成员 `book`，该成员返回 ISBN。

2. 使用带关联容器的比较器

如果考虑一下如何使用比较函数，就会认识到，它必须作为容器的部分而存储。任何在容器中增加或查找元素的操作都要使用比较函数。原则上，每个这样的操作可以接受一个可选的附加实参，表示比较函数。但是，这种策略容易导致出错：如果两个操作使用不同的比较函数，顺序可能会不一致。不可能预测实际上会发生什么。

要有效地工作，关联容器需要对每个操作使用同一比较函数。然而，期望用户每次记住比较函数是不合理的，尤其是，没有办法检查每个调用使用同一比较函数。因此，容器记住比较函数是有意义的。通过将比较器存储在容器对象中，可以保证比较元素的每个操作将一致地进行。

基于同样的理由，容器需要知道元素类型，为了存储比较器，它需要知道比较器类型。原则上，通过假定比较器是一个函数指针，该函数接受两个容器的 `key_type` 类型的对象并返回 `bool` 值，容器可以推断出这个类型。不幸的是，这个推断出的类型可能限制太大。首先，应该允许比较器是函数对象或是普通函数。即使我们愿意要求比较器为函数，这个推断出的类型也可能仍然太受限制了，毕竟，比较函数可以返回 `int` 或者其他任意可用在条件中的类型。同样，形参类型也不需要与 `key_type` 完全匹配，应该允许可以转换为 `key_type` 的任意形参类型。

所以，要使用 `Sales_item` 的比较函数，在定义 `multiset` 时必须指定比较器类型。在我们的例子中，比较器类型是接受两个 `const Sales_item` 引用并返回 `bool` 值的函数。

首先定义一个类型别名，作为该类型的同义词（7.9 节）：

```
// type of the comparison function used to order the multiset
typedef bool (*Comp)(const Sales_item&, const Sales_item&);
```

这个语句将 `Comp` 定义为函数类型指针的同义词，该函数类型与我们希望用来比较 `Sales_item` 对象的比较函数相匹配。

接着需要定义 `multiset`，保存 `Sales_item` 类型的对象并在它的比较函数中使用这个 `Comp` 类型。关联容器的每个构造函数使我们能够提供比较函数的名字。可以这样定义使用 `compare` 函数的空 `multiset`：

```
std::multiset<Sales_item, Comp> items(compare);
```

这个定义是说，`items` 是一个 `multiset`，它保存 `Sales_item` 对象并使用 `Comp` 类型的对象比较它们。`multiset` 是空的——我们没有提供任何元素，但我们的确提供了一个名为 `compare` 的比较函数。当在 `items` 中增加或查找元素时，将用 `compare` 函数对 `multiset` 进行排序。

3. 容器与句柄类

既然知道了怎样提供比较函数，我们将定义名为 `Basket` 的类，以跟踪销售并计算购买价格：

```
class Basket {
    // type of the comparison function used to order the multiset
    typedef bool (*Comp)(const Sales_item&, const Sales_item&);
public:
    // make it easier to type the type of our set
    typedef std::multiset<Sales_item, Comp> set_type;
    // typedefs modeled after corresponding container types
    typedef set_type::size_type size_type;
    typedef set_type::const_iterator const_iter;
    Basket(): items(compare) { } // initialize the comparator
    void add_item(const Sales_item &item)
        { items.insert(item); }
    size_type size(const Sales_item &i) const
        { return items.count(i); }
    double total() const; // sum of net prices for all items in the basket
private:
    std::multiset<Sales_item, Comp> items;
};
```

605

这个类在 `Sales_item` 对象的 `multiset` 中保存顾客购买的商品，用 `multiset` 使顾客能够购买同一本书的多个副本。

该类定义了一个构造函数，即 `Basket` 默认构造函数。该类需要自己的默认构造函数，以便将 `compare` 传给建立 `items` 成员的 `multiset` 构造函数。

`Basket` 类定义的操作非常简单：`add_item` 操作接受 `Sales_item` 对象引用并将该项目的副本放入 `multiset`；对于给定 ISBN，`size`¹ 操作返回购物篮中该 ISBN 的记录数。除了操作，`Basket` 还定义了三个类型别名，这样使用它的 `multiset` 成员就比较容易了。

4. 使用句柄执行虚函数

`Basket` 类唯一的复杂成员是 `total` 函数，该函数返回购物篮中所有物品的价格：

```
double Basket::total() const
{
    double sum = 0.0; // holds the running total
    /* find each set of items with the same isbn and calculate
     * the net price for that quantity of items
     * iter refers to first copy of each book in the set
     * upper_bound refers to next element with a different isbn
     */
    for (const_iter iter = items.begin();
         iter != items.end();
         iter = items.upper_bound(*iter))
    {
        // we know there's at least one element with this key in the Basket
        // virtual call to net_price applies appropriate discounts, if any
        sum += (*iter)->net_price(items.count(*iter));
    }
}
```

1. 此处原文误为 `item_count`。——译者注

```

    return sum;
}

```

total 函数有两个有趣的部分：对 net_price 函数的调用，以及 for 循环结构。我们逐一进行分析。

调用 net_price 函数时，需要告诉它某本书已经购买了多少本，net_price 函数使用这个实参确定是否打折。这个要求暗示着我们希望成批处理 multiset——处理给定标题的所有记录，然后处理下一个标题的所有记录，以此类推。幸好，multiset 非常适合处理这个问题。

for 循环开始于定义 iter 并将 iter 初始化为指向 multiset 中的第一个元素。我们使用 multiset 的 count 成员（10.3.6 节）确定 multiset 中的多少成员具有相同的键（即，相同的 isbn），并且使用该数目作为实参调用 net_price 函数。

for 循环中的“增量”表达式很有意思。与读每个元素的一般循环不同，我们推进 iter 指向下一个键。调用 upper_bound 函数以跳过与当前键匹配的所有元素，upper_bound 函数的调用返回一个迭代器，该迭代器指向与 iter 键相同的最后一个元素的下一元素，即，该迭代器指向集合的末尾或下一本书。测试 iter 的新值，如果与 items.end() 相等，则跳出 for 循环，否则，就处理下一本书。

for 循环的循环体调用 net_price 函数，阅读这个调用需要一点技巧：

```
sum += (*iter) -> net_price(items.count(*iter)) ;
```

对 iter 解引用获得基础 Sales_item 对象，对该对象应用 Sales_item 类重载的箭头操作符，该操作符返回句柄所关联的基础 Item_base 对象，用该 Item_base 对象调用 net_price 函数，传递具有相同 isbn 的图书的 count 作为实参。net_price 是虚函数，所以调用的定价函数的版本取决于基础 Item_base 对象的类型。

606

习题

习题 15.35 编写自己的 compare 函数和 Basket 类的版本并使用它们管理销售。

习题 15.36 Basket::const_iter 的基础类型是什么？

习题 15.37 为什么在 Basket 的 private 部分定义 Comp 类型别名？

习题 15.38 为什么在 Basket 中定义两个 private 部分？

15.9 再谈文本查询示例

作为继承的最后一个例子，我们来扩展 10.6 节的文本查询应用程序。使用在 10.6 节开发的类，已经能够在文本文件中查找给定单词的出现，但我们想扩展系统以支持更复杂的查询。

为了说明问题，将用下面的简单小说来运行查询：

```

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?"

```

607

系统应该支持:

- (1) 查找单个单词的查询。按升序显示所有包含该单词的行:

```

Executed Query for: Daddy
match 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?"

```

- (2) “非” 查询, 使用~操作符。显示所有不匹配的行:

```

Executed Query for: ~(Alice)
match 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.
. . .

```

- (3) “或” 查询, 使用|操作符。显示与两个查询条件中任意一个匹配的所有行:

```

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

```

- (4) “与” 查询, 使用&操作符。显示与两个查询条件都匹配的所有行:

```

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

```

而且, 可以组合这些元素, 如

```
fiery & bird | wind
```

我们的系统没有复杂到能够读这些表达式。我们将在 C++ 程序中创建它们, 因此, 将用常规 C++ 优先级规则对诸如此类的复合表达式求值。这个查询的求值结果将与出现 fiery 和 bird 的行或者出现 wind 的行相匹配, 而不会与 fiery 或 bird 单独出现的行相匹配:

```

Executing Query for: ((fiery & bird) | wind)
match 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,

```

输出将打印查询, 并使用圆括号指出解释该查询的方法。像原来的实现一样, 系统必须足够

608

聪明, 不会重复显示相同行。

15.9.1 面向对象的解决方案

可以考虑使用 10.6.2 节的 `TextQuery` 表示单词查询，然后从 `TextQuery` 类派生其他类。

但是，这个设计可能有缺陷。概念上，“非”查询不是一种单词查询，相反，非查询“有一个”查询（单词查询或其他任意种类的查询），非查询对该查询的值求反。

注意到这一点，我们将不同种类的查询建模为独立的类，它们共享一个公共基类：

```
WordQuery // Shakespeare
NotQuery  // ~Shakespeare
OrQuery   // Shakespeare | Marlowe
AndQuery  // William & Shakespeare
```

我们不继承 `TextQuery`，而是使用 `TextQuery` 类保存文件并建立相关的 `word_map`，使用查询类建立表达式，这些表达式最终对 `TextQuery` 对象中的文件运行查询。

1. 抽象接口类

已经识别出四种查询类，这些类在概念上是兄弟类。它们共享相同的抽象接口，这暗示我们定义一个抽象基类（15.6 节）以表示由查询执行的操作。将该抽象基类命名为 `Query_base`，以指出它的作用是作为查询继承层次的根。

直接从抽象基类派生 `WordQuery` 和 `NotQuery` 类，`WordQuery` 和 `NotQuery` 类具有系统中其他类所没有的一个性质：它们都有两个操作数。要为此建立模型，将在继承层次中增加另一个名为 `BinaryQuery` 的抽象类，表示带两个操作数的查询。`WordQuery` 和 `NotQuery` 类将继承 `BinaryQuery` 类，`BinaryQuery` 类继承 `Query_base` 类。这些决定得出了图 15-3 所示的类设计。

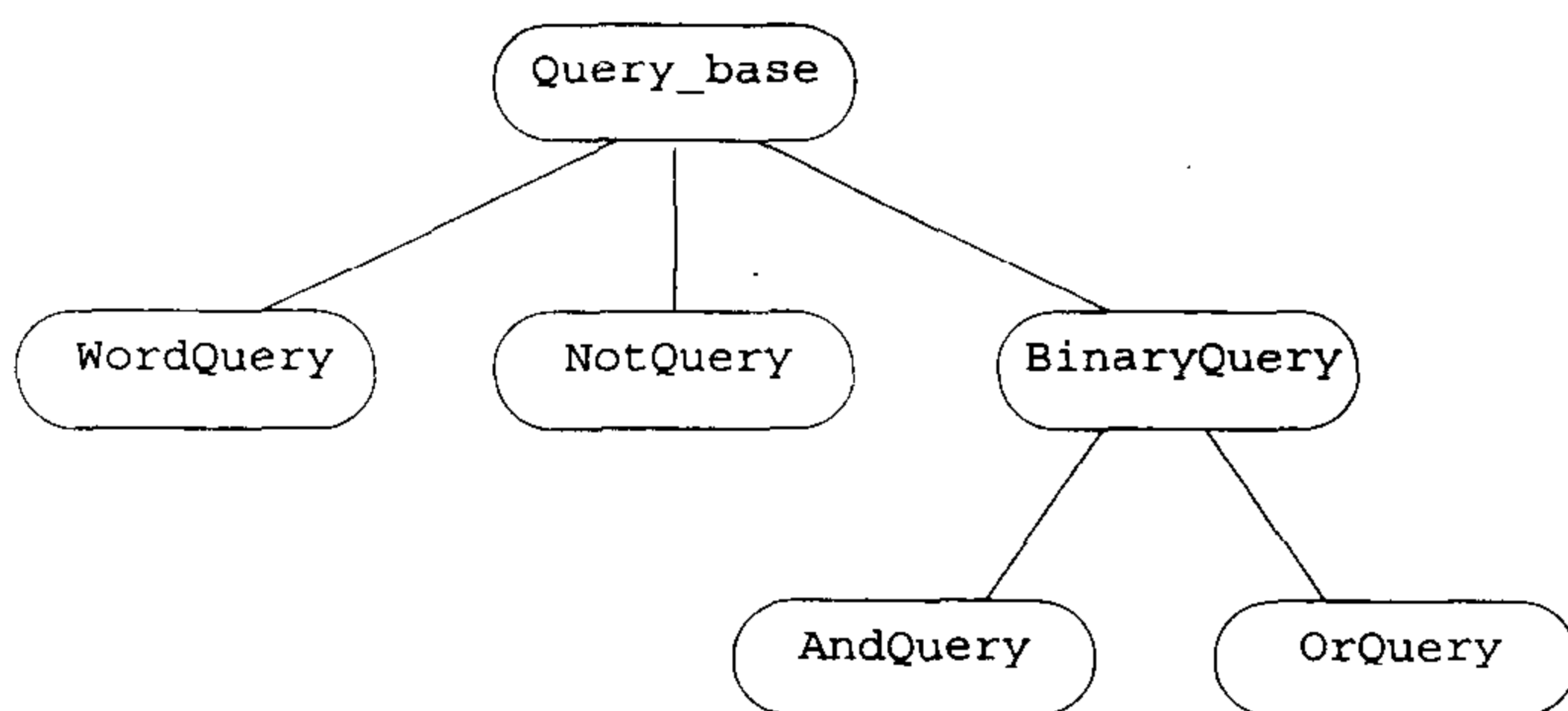


图 15-3 `Query_base` 继承层次

2. 操作

`Query_base` 类的存在主要是为了表示查询类型，不做实际工作。我们将重用 `TextQuery` 类以存储文件、建立查询以及查找每个单词。查询类型只需要两个操作：

(1) `eval` 操作，返回匹配行编号的集合。该操作接受 `TextQuery` 对象，在 `TextQuery` 对象上执行查询。

(2) `display` 操作，接受 `ostream` 引用并打印给定对象在该 `ostream` 上执行的查询。

我们将这些操作定义为 `Query_base` 中的纯虚函数（15.6 节），每个派生类都必须对这些函数定义自己的版本。

15.9.2 值型句柄

程序将处理计算查询，而不建立查询，但是，需要能够创建查询以便运行程序。最简单的办法是编写 C++表达式直接创建查询，例如，可以编写这样的代码：

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

以产生前面描述的复合查询。

这个问题描述暗示我们，用户级代码将不能直接使用我们的继承层次，相反，我们将定义一个名为 Query 的句柄类，用它隐藏继承层次。用户代码将根据句柄执行，用户代码只能间接操纵 Query_base 对象。

像 Sales_item 句柄一样，Query 句柄将保存指向继承层次中一个类型的对象的指针，Query 类还指向一个使用计数，我们用这个使用计数管理句柄指向的对象。

在这种情况下，句柄将完全屏蔽基础继承层次，用户将只能间接地通过 Query 对象的操作创建和操纵 Query_base 对象。我们将定义 Query 对象的三个重载操作符以及 Query 构造函数，Query 构造函数将动态分配新的 Query_base 对象。每个操作符将生成的对象绑定到 Query 句柄：&操作符将生成绑定到新的 AndQuery 对象的 Query 对象；|操作符将生成绑定到新的 OrQuery 对象的 Query 对象；~操作符将生成绑定到新的 NotQuery 对象的 Query 对象。给 Query 定义一个参数为 string 对象的构造函数，该构造函数将生成新的 WordQuery。

Query 类将提供与 Query_base 类同样的操作：eval 对相关查询进行计算，display 打印查询。它将定义重载输出操作符显示相关查询。

表 15-1 查询程序设计：扼要重述

操作或表达式	功 能
TextQuery	读指定文件并建立相关查找映射的类。该类提供 query_text 操作，该操作接受 string 实参并返回一个 set，保存出现实参的行的编号。
Query_base	查询类的抽象基类。
Query	用户计数的句柄类，它指向 Query_base 派生类型的对象。
WordQuery	从 Query_base 派生的类，查找给定单词。
NotQuery	从 Query_base 派生的类，返回操作数不出现的行的编号集合。
BinaryQuery	从 Query_base 派生的抽象基类类型，表示带两个 Query 操作数的查询。
OrQuery	从 BinaryQuery 派生的类，返回两个操作数出现的行编号集的并集。
AndQuery	从 BinaryQuery 派生的类，返回两个操作数出现的行编号集的交集。
q1 & q2	返回 Query 对象，该 Query 对象绑定到保存 q1 和 q2 的新 AndQuery 对象。
q1 q2	返回 Query 对象，该 Query 对象绑定到保存 q1 和 q2 的新 OrQuery 对象。
~q	返回 Query 对象，该 Query 对象绑定到保存 q 的新 NotQuery 对象。
Query q(s)	将 Query q 绑定到保存 string s 的新 WordQuery 对象。

我们的设计：扼要重述



理解设计经常是最困难的部分，尤其是刚开始设计面向对象系统时。一旦熟悉了设计，实现就是顺理成章的了。

这个应用程序的主要工作由建立对象表示用户的查询构成，认识到这一点很重要。正如图 15-4 所示，表达式

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

生成 10 个对象：5 个 Query_base 对象及其相关联的句柄。5 个 Query_base 对象分别是 3 个 WordQuery 对象，一个 OrQuery 对象和一个 AndQuery 对象。

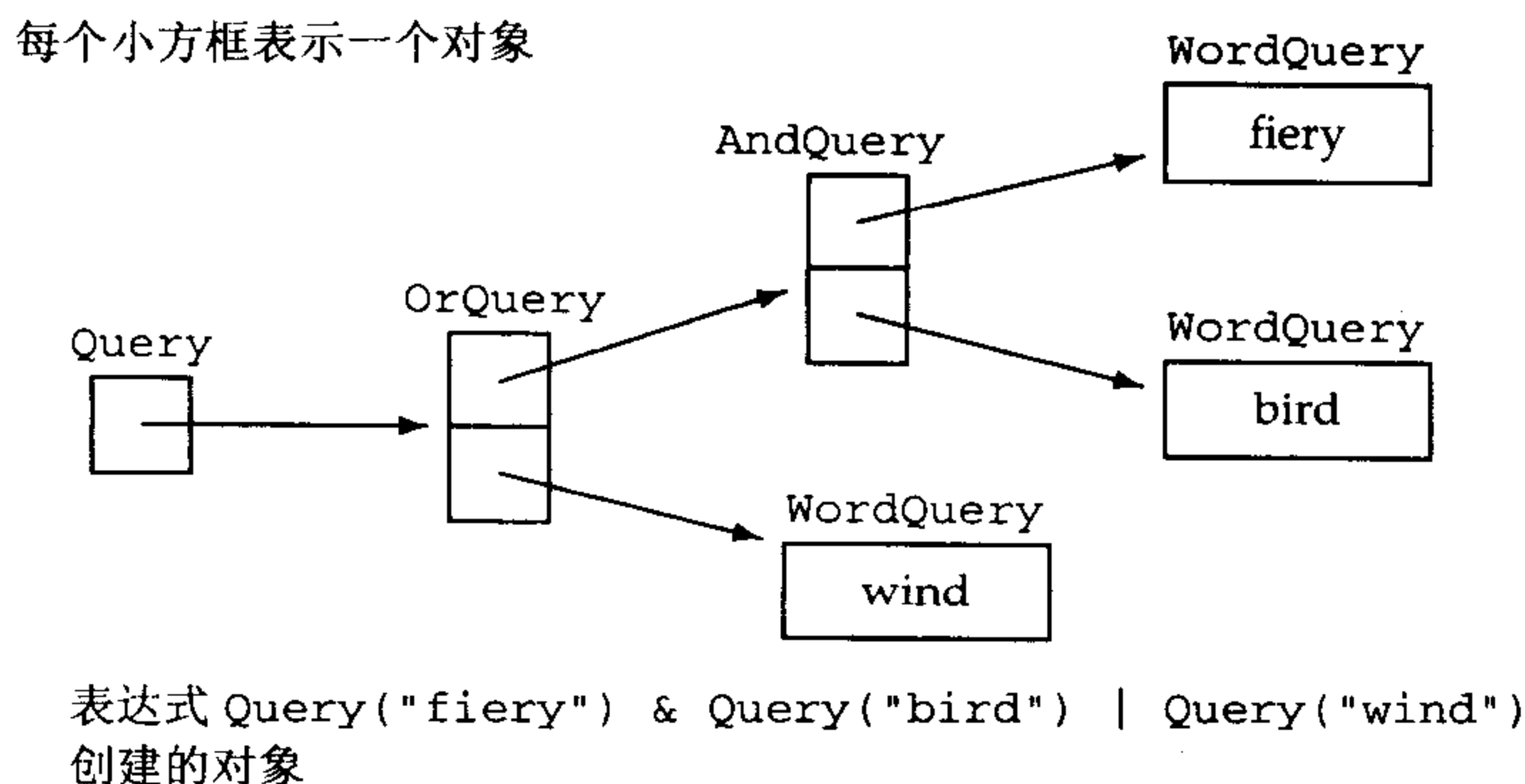


图 15-4 Query 表达式创建的对象

一旦建立了对象树，计算（或显示）给定查询基本上是沿着这些链接，要求树中每个对象计算（或显示）自己的过程，该过程由编译器管理。例如，如果调用 `q`（即，在这棵树的树根）的 `eval`，则 `eval` 将要求 `q` 指向的 `OrQuery` 对象调用 `eval` 来计算自己，计算这个 `OrQuery` 对象用两个操作数调用 `eval`，这会依次调用 `AndQuery` 对象和 `WordQuery` 对象的 `eval`，查找单词 `wind`，依此类推。

611

习题

习题 15.39 给定 `s1`、`s2`、`s3` 和 `s4` 均为 `string` 对象，确定下述 `Query` 类的使用创建什么对象：

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

15.9.3 Query_base 类

现在我们的设计已经解释清楚，该开始实现了。首先来定义 `Query_base` 类：

```
// private, abstract class acts as a base class for concrete query types
class Query_base {
    friend class Query;
protected:
    typedef TextQuery::line_no line_no;
    virtual ~Query_base() { }
private:
    // eval returns the |set| of lines that this Query matches

```

```

        virtual std::set<line_no>
            eval(const TextQuery&) const = 0;
        // display prints the query
        virtual std::ostream&
            display(std::ostream& = std::cout) const = 0;
};

```

612

这个类定义了两个接口成员：eval 和 display。两个成员都是纯虚函数（15.6 节），因此该类为抽象类，应用程序中将没有 Query_base 类型的对象。

用户和派生类将只通过 Query 句柄使用 Query_base 类，因此，将 Query_base 接口设为 private。（虚）析构函数（15.4.4 节）和类型别名为 protected，这些派生类型就能够访问这些成员，构造函数由派生类构造函数（隐式）使用，因此派生类必须能够访问构造函数。

给 Query 句柄类授予友元关系，该类的成员将调用 Query_base 中的虚函数因此必须能够访问它们。

15.9.4 Query 句柄类

Query 句柄将类似于 Sales_item 类，因为它将保存 Query_base 指针和使用计数指针。像 Sales_item 类一样，Query 的复制控制成员将管理使用计数和 Query_base 指针。

与 Sales_item 类不同的是，Query 类将只为 Query_base 继承层次提供接口。用户将不能直接访问 Query 或其派生类的任意成员，这一设计决定导致 Query 和 Sales_item 之间存在两个区别。第一个区别是，Query 类将不定义解引用操作符和箭头操作符的重载版本。Query_base 类没有 public 成员，如果 Query 句柄定义了解引用操作符和箭头操作符，它们将没有用处！使用那些操作符访问成员的任何尝试都将失败，相反，Query 类必须定义接口函数 eval 和 display 的自身版本。

另一个区别来自于我们打算怎样创建继承层次的对象。我们的设计指出将只通过 Query 句柄的操作创建 Query_base 的派生类对象，这个区别导致 Query 类需要与 Sales_item 句柄中所用的构造函数不同的构造函数。

1. Query 类

按照前面的设计，Query 类本身相当简单：

```

// handle class to manage the Query_base inheritance hierarchy
class Query {
    // these operators need access to the Query_base* constructor
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);
public:
    Query(const std::string&); // builds a new WordQuery
    // copy control to manage pointers and use counting
    Query(const Query &c): q(c.q), use(c.use) { ++*use; }
    ~Query() { decr_use(); }
    Query& operator=(const Query&);
    // interface functions: will call corresponding Query_base operations
    std::set<TextQuery::line_no>
        eval(const TextQuery &t) const { return q->eval(t); }
}

```

613

```

        std::ostream &display(std::ostream &os) const
            { return q->display(os); }
private:
    Query(Query_base *query): q(query),
                               use(new std::size_t(1)) { }

    Query_base *q;
    std::size_t *use;
    void decr_use()
    { if (--*use == 0) { delete q; delete use; } }
};

```

首先指定创建 Query 对象的操作符为友元，我们将很快看到为什么需要将这些操作符设为友元。

在 Query 类的 public 接口中，声明了但没有定义接受 string 对象的构造函数，该构造函数创建 WordQuery 对象，因此在定义 WordQuery 类之前不能定义它。

后面三个成员处理复制控制，与 Sales_item 类中的对应成员相同。

最后两个 public 成员表示对 Query_base 类的接口。每种情况下，Query 操作都使用它的 Query_base 指针调用相应 Query_base 操作。这些操作是虚函数，在运行时根据 q 指向的对象的类型确定调用的实际版本。

Query 类实现的 private 部分包括一个接受 Query_base 对象指针的构造函数，该构造函数将获得的指针存储在 q 中并分配新的使用计数，将使用计数初始化为 1。该构造函数为 private，是因为我们不希望普通用户代码定义 Query_base 对象，相反，创建 Query 对象的操作符需要这个构造函数。因为构造函数为 private，所以必须将操作符设为友元。

2. Query 重载操作符

|、&和~操作符分别创建 OrQuery、AndQuery 和 NotQuery 对象：

```

inline Query operator&(const Query &lhs, const Query &rhs)
{
    return new AndQuery(lhs, rhs);
}
inline Query operator|(const Query &lhs, const Query &rhs)
{
    return new OrQuery(lhs, rhs);
}
inline Query operator~(const Query &oper)
{
    return new NotQuery(oper);
}

```

614

每个操作动态分配 Query_base 派生类型的新对象，return 语句（隐式）使用接受 Query_base 指针的 Query 构造函数，用操作分配的 Query_base 指针创建 Query 对象。例如，~操作符中的 return 语句等价于：

```

// allocate a new NotQuery object
// convert the resulting pointer to NotQuery to a pointer to Query_base
Query_base *tmp = new NotQuery(expr);
return Query(tmp); // use Query constructor that takes a pointer to Query_base

```

没有操作符创建 WordQuery 对象，相反，为 Query 类定义一个接受 string 对象的构造函数

数，该构造函数生成 WordQuery 对象查找给定 string。

3. Query 输出操作符

我们希望用户可以用标准（重载的）输出操作符打印 Query 对象，但是，也需要打印操作是虚函数——打印 Query 对象应打印 Query 对象指向的 Query_base 对象。这里存在一个问题：只有成员函数可以为虚函数，但输出操作符不能是 Query_base 类的成员（14.2.1 节）。

要获得必要的虚函数行为，Query_base 类定义了一个虚函数成员 display，Query 输出操作符将使用它：

```
inline std::ostream&
operator<<(std::ostream &os, const Query &q)
{
    return q.display(os);
}
```

如果编写

```
Query andq = Query(sought1) & Query(sought2);
cout << "\nExecuted query: " << andq << endl;
```

将调用 Query 输出操作符，该操作符调用 q.display(os)，其中，q 引用指向该 AndQuery 对象的 Query 对象，os 绑定到 cout。如果编写

```
Query name(sought);
cout << "\nExecuted Query for: " << name << endl;
```

将调用 display 的 WordQuery 实例。更一般的，以下代码

```
Query query = some_query;
cout << query << endl;
```

615 将调用程序运行到此时与 query 所指对象相关联的 display 实例。

15.9.5 派生类

下面要实现具体的查询类。关于这些类，一个有趣的部分是如何表示它们。WordQuery 类最直接，它的工作是保存要查找的单词。

其他类操作一个或两个 Query 操作数。NotQuery 对象对别的 Query 对象的结果求反，AndQuery 类和 OrQuery 类都有两个操作数，操作数实际存储在它们的公共基类 BinaryQuery 中。

在这些类当中，操作数都可以是任意具体 Query_base 类的对象：NotQuery 对象可以应用于 WordQuery 对象、AndQuery 对象、OrQuery 对象或其他 NotQuery 对象。要允许这种灵活性，操作数必须存储为 Query_base 指针，它可以指向任意具体的 Query_base 类。

但是，我们的类不存储 Query_base 指针，而是自己使用 Query 句柄。正如使用句柄可以简化用户代码，也可以使用同样的句柄类简化类代码。将 Query 操作数设为 const，因为一旦创立了 Query_base 对象，就没有操作可以改变操作数了。

了解了这些类的设计之后，就可以实现它们了。

1. WordQuery 类

WordQuery 是一种 Query_base，它在给定的查询映射中查找指定单词：

```

class WordQuery: public Query_base {
    friend class Query; // Query uses the WordQuery constructor
    WordQuery(const std::string &s): query_word(s) { }
    // concrete class: WordQuery defines all inherited pure virtual functions
    std::set<line_no> eval(const TextQuery &t) const
        { return t.run_query(query_word); }
    std::ostream& display (std::ostream &os) const
        { return os << query_word; }
    std::string query_word; // word for which to search
};

```

像 Query_base 类一样, WordQuery 类没有 public 成员, WordQuery 类必须将 Query 类设为友元以允许 Query 访问 WordQuery 构造函数。

每个具体的查询类必须定义继承的纯虚函数。WordQuery 类的操作足够简单, 可以定义在类定义体中。eval 成员调用其 TextQuery 形参的 query_text 成员, 将用于创建该 WordQuery 对象的 string 对象传给它。要 display 一个 WordQuery 对象, 就打印 query_word 对象。

2. NotQuery 类

NotQuery 对象保存一个 const Query 对象, 对它求反:

```

class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(Query q): query(q) { }
    // concrete class: NotQuery defines all inherited pure virtual functions
    std::set<line_no> eval(const TextQuery&) const;
    std::ostream& display(std::ostream &os) const
        { return os << "~(" << query << ")"; }
    const Query query;
};

```

616

将 Query 的重载~操作符设为友元, 从而允许该操作符创建新的 NotQuery 对象。为了 display 一个 NotQuery 对象, 打印~符号, 后接基础 Query 对象, 将输出用圆括号括住以保证读者清楚优先级。

display 操作中输出操作符的使用最终是对 Query_base 对象的虚函数调用:



```

// uses the Query output operator, which calls Query::display
// that function makes a virtual call to Query_base::display
{ return os << "~(" << query << ")"; }

```

eval 成员比较复杂, 我们将在类定义体之外实现它, eval 函数见 15.9.6 节。

3. BinaryQuery 类

BinaryQuery 类是一个抽象类, 保存 AndQuery 和 OrQuery 两个查询类型所需的数据, AndQuery 和 OrQuery 有两个操作数:

```

class BinaryQuery: public Query_base {
protected:
    BinaryQuery(Query left, Query right, std::string op):
        lhs(left), rhs(right), oper(op) { }
    // abstract class: BinaryQuery doesn't define eval
    std::ostream& display(std::ostream &os) const
    { return os << "(" << lhs << " " << oper << " "
        << rhs << ")"; }
};

```

```

    const Query lhs, rhs;    // right- and left-hand operands
    const std::string oper;  // name of the operator
};

```

BinaryQuery 中的数据是两个 Query 操作数，以及显示查询时使用的操作符符号。这些数据均声明为 const，因为一旦建立了查询的内容就不应该再改变。构造函数接受两个操作数以及操作符符号，将它们存储在适当的数据成员中。

要显示一个 Binaryoperator 对象，打印由圆括号括住的表达式、该表达式由左操作数后接操作符、再接右操作数构成。像显示 NotQuery 对象一样，用于打印 left 和 right 的重载 << 操作符最终对基础 Query_base 对象的 display 进行虚函数调用。

617



BinaryQuery 类没有定义 eval 函数，因此继承了一个纯虚函数。这样，BinaryQuery 也是一个抽象类，不能创建 BinaryQuery 类型的对象。

4. AndQuery 和 OrQuery 类

AndQuery 类和 OrQuery 类几乎完全相同：

```

class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery (Query left, Query right):
        BinaryQuery(left, right, "&") { }
    // concrete class: AndQuery inherits display and defines remaining pure virtual
    std::set<line_no> eval(const TextQuery&) const;
};

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery (Query left, Query right):
        BinaryQuery(left, right, "|") { }
    // concrete class: OrQuery inherits display and defines remaining pure virtual
    std::set<line_no> eval(const TextQuery&) const;
};

```

这两个类将各自的操作符设为友元，并定义了构造函数用适当的操作符创建它们的 BinaryQuery 基类部分。它们继承 BinaryQuery 类的 display 函数定义，但各自定义了自己的 eval 函数版本。

习题

习题 15.40 对图 15-4 中建立的表达式：

- 列出处理这个表达式所执行的构造函数。
- 列出执行 `cout << q` 所调用的 display 函数和重载的 << 操作符。
- 列出计算 `q.eval` 时所调用的 eval 函数。

15.9.6 eval 函数

查询类层次的中心是虚函数 eval。每个 eval 函数调用其操作数的 eval 函数，然后应用自

己的逻辑：AndQuery 的 eval 操作返回两个操作数的结果的并集，OrQuery 的 eval 操作返回交集，NotQuery 的 eval 操作比较复杂：它必须返回不在其操作数的集合中的行编号。

618

1. OrQuery::eval

OrQuery 对象合并由它的两个操作数返回的行编号集合——其结果是它的两个操作数的结果的并集：

```
// returns union of its operands' result sets
set<TextQuery::line_no>
OrQuery::eval(const TextQuery& file) const
{
    // virtual calls through the Query handle to get result sets for the operands
    set<line_no> right = rhs.eval(file),
                ret_lines = lhs.eval(file); // destination to hold results
    // inserts the lines from right that aren't already in ret_lines
    ret_lines.insert(right.begin(), right.end());
    return ret_lines;
}
```

eval 函数首先调用每个操作数的 eval 函数，操作数的 eval 函数调用 Query::eval，Query::eval 再调用基础 Query_base 对象的虚函数 eval，每个调用获得其操作数出现在其中的行的编号 set。然后 OrQuery::eval 函数调用 ret_lines 的 insert 函数，传递一对迭代器表示对右操作数求值所返回的 set。因为 ret_lines 是一个 set 对象，这个调用将 right 中未在 left 中出现的元素加到 ret_lines 中。调用 insert 函数之后，ret_lines 包含在 left 集或在 right 集的每个行编号。返回 ret_lines 而结束 OrQuery::eval 函数。

2. AndQuery::eval

AndQuery 的 eval 版本使用了完成集合式操作的一个标准库算法。标准库附录中说明了这些算法，见 A.2.8 节。

```
// returns intersection of its operands' result sets
set<TextQuery::line_no>
AndQuery::eval(const TextQuery& file) const
{
    // virtual calls through the Query handle to get result sets for the operands
    set<line_no> left = lhs.eval(file),
                right = rhs.eval(file);
    set<line_no> ret_lines; // destination to hold results
    // writes intersection of two ranges to a destination iterator
    // destination iterator in this call adds elements to ret
    set_intersection(left.begin(), left.end(),
                    right.begin(), right.end(),
                    inserter(ret_lines, ret_lines.begin()));
    return ret_lines;
}
```

eval 函数的这个版本使用 set_intersection 算法查找两个查询中的公共行：该算法接受 5 个迭代器，前 4 个表示两个输入范围，最后一个表示目的地。算法将同时在两个输入范围中存在的每个元素写到目的地。该调用的目的地是一个迭代器（11.3.1 节），它将新元素插入到 ret_lines 中。

619

3. NotQuery::eval

NotQuery 查找未出现操作数的每个文本行。要支持这个函数，需要 TextQuery 类增加一个成员返回文件的大小，以便了解存在什么样的行编号。

```
// returns lines not in its operand's result set
set<TextQuery::line_no>
NotQuery::eval(const TextQuery& file) const
{
    // virtual call through the Query handle to eval
    set<TextQuery::line_no> has_val = query.eval(file);
    set<line_no> ret_lines;
    // for each line in the input file, check whether that line is in has_val
    // if not, add that line number to ret_lines
    for (TextQuery::line_no n = 0; n != file.size(); ++n)
        if (has_val.find(n) == has_val.end())
            ret_lines.insert(n);
    return ret_lines;
}
```

像其他 eval 函数一样，首先调用该对象的操作数的 eval 函数。该调用返回操作数所出现的行编号的 set，而我们想要的是不出现操作数的行编号的 set，通过查找输入文件的每个行编号获得该 set。使用必须加到 TextQuery 的 size 成员控制 for 循环，该循环将没有在 has_val 中出现的每个行编号加到 ret_lines 中，一旦处理完所有的行编号，就返回 ret_lines。

习题

习题 15.41 实现 Query 类和 Query_base 类，并为第 10 章的 TextQuery 类增加需要的 size 操作。通过计算和打印如图 15-4 所示的查询，测试你的应用程序。

习题 15.42 设计并实现下述增强中的一个：

- (a) 引入基于同一句子而不是同一行计算单词的支持。
- (b) 引入历史系统，用户可以用编号查阅前面的查询，并可以在其中增加内容或与其他查询组合。
- (c) 除了显示匹配数目和所有匹配行之外，允许用户对中间查询计算和最终查询指出要显示的行的范围。

小结

继承和动态绑定的思想，简单但功能强大。继承使我们能够编写新类，新类与基类共享行为但重定义必要的行为。动态绑定使编译器能够在运行时根据对象的动态类型确定运行函数的哪个版本。继承和动态绑定的结合使我们能够编写具有特定类型行为而又独立于类型的程序。

在 C++ 中，动态绑定仅在通过引用或指针调用时才能应用于声明为虚的函数。C++ 程序定义继承层次接口的句柄类很常见，这些类分配并管理指向继承层次中对象的指针，因此能够使用户代码在无须处理指针的情况下获得动态行为。

继承对象由基类部分和派生类部分组成。继承对象是通过在处理派生部分之前对对象的基类部分进行构造、复制和赋值，而进行构造、复制和赋值的。因为派生类对象包含基类部分，所以可以将派生类型的引用或指针转换为基类类型的引用或指针。

即使另外不需要析构函数，基类通常也应定义一个虚析构函数。如果经常会在指向基类的指针实际指向派生类对象时删除它，析构函数就必须为虚函数。

术语

abstract base class (抽象基类) 具有或继承了一个或多个纯虚函数的类。不能创建抽象基类类型的对象。抽象基类的存在定义了一个接口，派生类将为基类中定义的纯虚函数定义特定类型的实现，以完成类型。

base class (基类) 从中派生其他类的类。基类的成员成为派生类的成员。

class derivation list (类派生列表) 类定义用它指出类是派生类。派生列表包含可选的访问级别和基类的名字，如果不指定访问标号，继承的类型取决于用来定义派生类的保留字，默认情况下，如果派生类用保留字 `struct` 定义，则公有继承基类，如果派生类用保留字 `class` 定义，则私有继承基类。

derived class (派生类) 继承了其他类的类。基类的成员也是派生类的成员，派生类可以重定义其基类的成员还可以定义新成员。派生类作用域嵌套在基类作用域中，因此派生类可以直接访问基类的成员。派生类中定义的与基类成员同名的成员屏蔽基类成员，具体而言，派生类中的成员函数不重载基类成员。基类中的被屏蔽成员可以用作用域操作符访问。

direct base class (直接基类) **immediate base class** 的同义词。

dynamic binding (动态绑定) 延迟到运行时才选择运行哪个函数。在 C++ 中，动态绑定指的是在运行时基于引用或指针绑定的对象的基础类型而选择运行哪个 `virtual` 函数。

dynamic type (动态类型) 运行时类型。基类类型的指针和引用可以绑定到派生类型的对象，在这种情况下，静态类型是基类引用（或指针），但动态类型是派生类引用（或指针）。

handle class (句柄类) 提供到其他类的接口的类。一般用于分配和管理继承层次对象的指针。

immediate base class (直接基类) 被派生类直接继承的基类。直接基类是在派生列表中指定的基类，直接基类本身可以是派生类。

indirect base class (间接基类) 非直接基类。直接基类直接或间接继承的类是派生类的间接基类。

inheritance hierarchy (继承层次) 描述因共享公共基类的继承而相关联的类之间关系的术语。

object-oriented programming (面向对象编程) 描述使用数据抽象、继承和动态绑定的程序的术语。

polymorphism (多态性) 从意为“多种形态”的希腊单词派生的术语。在面向对象编程中，多态性指的是基于引用或指针的动态类型获得类型明确的行为的能力。

private inheritance (私有继承) 实现继承的一种形式，在这种形式中，`private` 基类的 `public` 和 `protected` 成员在派生类中均为 `private`。

protected access label (受保护访问标号) 定义在 `protected` 标号之后的成员可以被类成员、友元和派生类成员（非友元）访问。类的普通用户不能访问 `protected` 成员。

protected inheritance (受保护继承) 在受保护继承中，基类的 `protected` 和 `public` 成员在派生类中均为 `protected`。

public inheritance (公有继承) 基类的 `public` 接口是派生类的 `public` 接口的一部分。

pure virtual (纯虚函数) 类的头文件中在函数形参表末尾用 `=0` 声明的虚函数。类不必（但可以）定义纯虚函数。带纯虚函数的类为抽象类。如果派生类没有定义所继承的纯虚函数的自身版本，则派生类也是抽象类。

refactoring (重构) 重新设计程序将相关部分集合到一个抽象中，用新抽象的使用代替原来的代码。在面向对象编程中，重新设计继承层

次中的类时经常发生重构。响应需求的改变通常需要进行重构。一般而言，对类进行重构时，需要将数据或函数成员移到继承层次的最高公共点以避免代码重复。

sliced (切掉的) 描述用派生类型对象对基类类型对象进行初始化或赋值时情况的术语。对象的派生部分被“切掉”，只留下基类部分，赋给基类对象。

static type (静态类型) 编译时类型。对象的静态类型与动态类型相同。引用或指针所引用的对象的动态类型可以不同于引用或指针的静态类型。

virtual function (虚函数) 定义类型明确的行为的成员函数。通过引用或指针进行的虚函数调用，在运行时基于引用或指针所绑定的对象而确定。

621
622

第 16 章

模板与泛型编程

目录

16.1	模板定义.....	526
16.2	实例化.....	535
16.3	模板编译模型.....	542
16.4	类模板成员.....	545
16.5	一个泛型句柄类.....	560
16.6	模板特化.....	564
16.7	重载与函数模板.....	570
	小结.....	573
	术语.....	574

所谓泛型编程就是以独立于任何特定类型的方式编写代码。使用泛型程序时，我们需要提供具体程序实例所操作的类型或值。第二部分中描述的标准库的容器、迭代器和算法都是泛型编程的例子。每种容器（如 `vector`）都有单一的定义，但可以定义许多不同种类的 `vector`，它们的区别在于所包含的元素类型。

模板是泛型编程的基础。使用模板时可以无须了解模板的定义。本章将介绍怎样定义自己的模板类和模板函数。

泛型编程与面向对象编程一样，都依赖于某种形式的多态性¹。面向对象编程中的多态性在运行时应用于存在继承关系的类。我们能够编写使用这些类的代码，忽略基类与派生类之间类型上的差异。只要使用基类的引用或指针，基类类型或派生类类型的对象就可以使用相同的代码。

在泛型编程中，我们所编写的类和函数能够多态地用于跨越编译时不相关的类型。一个类或一个函数可以用来操纵多种类型的对象。标准库中的容器、迭代器和算法是很好的泛型编程的例子。标准库用独立于类型的方式定义每个容器、迭代器和算法，因此几乎可以在任意类型上使用标准库的类和函数。例如，虽然 `vector` 的设计者不可能了解应用程序特定的类，但我们能够定义 `Sales_item` 对象组成的 `vector`。

在 C++ 中，模板是泛型编程的基础。模板是创建类或函数的蓝图或公式。例如，标准库定义了一个类模板，该模板定义了 `vector` 的含义，它可以用于产生任意数量的特定类型的 `vector` 类，例如，`vector<int>` 或 `vector<string>`。本书第二部分介绍了怎样使用泛型类型和泛型函数，本章将介绍怎样自定义模板。

16.1 模板定义

假设想要编写一个函数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。实践中，我们可能希望定义几个这样的函数，每一个可以比较一种给定类型的值，第一次尝试可能是定义几个重载函数：

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
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;
}
```

这些函数几乎相同，它们之间唯一的区别是形参的类型，每个函数的函数体是相同的。

每个要比较的类型都需重复函数的函数体，不仅麻烦而且容易出错。更重要的是，需要事先知道究竟可能会比较哪些类型。如果希望将函数用于未知类型，这种策略就不起作用了。

16.1.1 定义函数模板

我们可以不用为每个类型定义一个新函数，而是只定义一个函数模板（function template）。函数模板是一个独立于类型的函数，可作为一种方式，产生函数的特定类型版本。例如，可以编

1. 面向对象编程所依赖的多态性称为运行时多态性，泛型编程所依赖的多态性称为编译时多态性或参数式多态性。——译者注

写名为 `compare` 的函数模板，它告诉编译器如何为我们想要比较的类型产生特定的 `compare` 版本。

下面是 `compare` 的模板版本：

```
// implement strcmp-like generic compare function
// returns 0 if the values are equal, 1 if v1 is larger, -1 if v1 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

模板定义以关键字 `template` 开始，后接**模板形参表**（`template parameter list`），模板形参表是用尖括号括住的一个或多个**模板形参**（`template parameter`）的列表，形参之间以逗号分隔。



模板形参表不能为空。

1. 模板形参表

模板形参表很像函数形参表，函数形参表定义了特定类型的局部变量但并不初始化那些变量，在运行时再提供实参来初始化形参。

同样，模板形参表示可以在类或函数的定义中使用的类型或值。例如，`compare` 函数声明了一个名为 `T` 的类型形参。在 `compare` 内部，可以使用名字 `T` 引用一个类型，`T` 表示哪个实际类型由编译器根据所用的函数而确定。

模板形参可以是表示类型的**类型形参**（`type parameter`），也可以是表示常量表达式的**非类型形参**（`nontype parameter`）。非类型形参跟在类型说明符之后声明，16.1.5 节将进一步介绍非类型形参。类型形参跟在关键字 `class` 或 `typename` 之后定义，例如，`class T` 是名为 `T` 的类型形参，在这里 `class` 和 `typename` 没有区别。

2. 使用函数模板

使用函数模板时，编译器会推断哪个（或哪些）**模板实参**（`template argument`）绑定到模板形参。一旦编译器确定了实际的模板实参，就称它**实例化**（`instantiate`）了函数模板的一个实例。

625

实质上，编译器将确定用什么类型代替每个类型形参，以及用什么值代替每个非类型形参。推导出实际模板实参后，编译器使用实参代替相应的模板形参产生并编译该版本的函数。编译器承担了为我们使用的每种类型而编写函数的单调工作。

对于以下的调用

```
int main ()
{
    // T is int;
    // compiler instantiates int compare(const int&, const int&)
    cout << compare(1, 0) << endl;
    // T is string;
    // compiler instantiates int compare(const string&, const string&)
```

```

    string s1 = "hi", s2 = "world";
    cout << compare(s1, s2) << endl;
    return 0;
}

```

编译器将实例化 `compare` 的两个不同版本, 编译器将用 `int` 代替 `T` 创建第一个版本, 并用 `string` 代替 `T` 创建第二个版本。

3. inline 函数模板

函数模板可以用与非模板函数一样的方式声明为 `inline`。说明符放在模板形参表之后、返回类型之前, 不能放在关键字 `template` 之前。

```

// ok: inline specifier follows template parameter list
template <typename T> inline T min(const T&, const T&);
// error: incorrect placement of inline specifier
inline template <typename T> T min(const T&, const T&);

```

习题

习题 16.1 编写一个模板返回形参的绝对值。至少用三种不同类型的值调用模板。注意: 在 16.3 节讨论编译器怎样处理模板实例化之前, 你应该将每个模板定义和该模板的所有使用放在同一文件中。

习题 16.2 编写一个函数模板, 接受一个 `ostream` 引用和一个值, 将该值写入流。用至少四种不同类型调用函数。通过写至 `cout`、写至文件和写至 `stringstream` 来测试你的程序。

习题 16.3 当调用两个 `string` 对象的 `compare` 时, 传递用字符串字面值初始化的两个 `string` 对象。如果编写以下代码会发生什么?

```
compare ("hi", "world");
```

626

16.1.2 定义类模板

就像可以定义函数模板一样, 也可以定义类模板。



为了举例说明类模板, 我们将为标准库 `queue` 类 (9.7 节) 实现一个自己的版本。用户程序应使用标准的 `queue` 类, 而不是我们这里定义的这个 `Queue` 类。

我们自定义的 `Queue` 类必须能够支持不同类型的对象, 所以将它定义为类模板 (`class template`)。 `Queue` 类将支持的操作是标准 `queue` 类接口的子集:

- `push` 操作, 在队尾增加一项
- `pop` 操作, 从队头删除一项
- `front` 操作, 返回队头元素的引用
- `empty` 操作, 指出队列中是否有元素

16.4 节将介绍怎样实现 `Queue` 类, 这里先定义它的接口:

```

template <class Type> class Queue {
public:
    Queue ();                // default constructor

```

```

    Type &front ();           // return element from head of Queue
    const Type &front () const;
    void push (const Type &); // add element to back of Queue
    void pop();               // remove element from head of Queue
    bool empty() const;       // true if no elements in the Queue
private:
    // ...
};

```

类模板也是模板，因此必须以关键字 `template` 开头，后接模板形参表。Queue 模板接受一个名为 `Type` 的模板类型形参。

除了模板形参表外，类模板的定义看起来与任意其他类相似。类模板可以定义数据成员、函数成员和类型成员，也可以使用访问标号控制对成员的访问，还可以定义构造函数和析构函数等等。在类和类成员的定义中，可以使用模板形参作为类型或值的占位符，在使用类时再提供那些类型或值。

例如，Queue 模板有一个模板类型形参，可以在任何可以使用类型名字的地方使用该形参。在这个模板定义中，用 `Type` 指定重载 `front` 操作的返回类型以及作为 `push` 操作的形参类型。

627

使用类模板

与调用函数模板形成对比，使用类模板时，必须为模板形参显式指定实参：

```

Queue<int> qi;                // Queue that holds ints
Queue< vector<double> > qc;    // Queue that holds vectors of doubles
Queue<string> qs;             // Queue that holds strings

```

编译器使用实参来实例化这个类的特定类型版本。实质上，编译器用用户提供的实际特定类型代替 `Type`，重新编写 Queue 类。在这个例子中，编译器将实例化三个 Queue 类：第一个用 `int` 代替 `Type`，第二个用 `vector<double>` 代替 `Type`，第三个用 `string` 代替 `Type`。

习题

习题 16.4 什么是函数模板？什么是类模板？

习题 16.5 定义一个函数模板，返回两个值中较大的一个。

习题 16.6 类似于我们的 `queue` 简化版本，编写一个名为 `List` 的类模板，作为标准 `list` 类的简化版本。

16.1.3 模板形参

像函数形参一样，程序员为模板形参选择的字没有本质含义。在我们的例子中，将 `compare` 的模板类型形参命名为 `T`，但也可以将它命名为任意名字：

```

// equivalent template definition
template <class Glorp>
int compare(const Glorp &v1, const Glorp &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}

```

该代码定义的 `compare` 模板与前面一样。

可以给模板形参赋予的唯一含义是区别形参是类型形参还是非类型形参。如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。

如果希望使用模板形参所表示的类型或值，可以使用与对应模板形参相同的名字。例如，

628 `compare` 函数中所有的 `Glorp` 引用将在该函数被实例化时确定为同一类型。

1. 模板形参作用域

模板形参的名字可以在声明为模板形参之后直到模板声明或定义的末尾处使用。

模板形参遵循常规名字屏蔽规则。与全局作用域中声明的对象、函数或类型同名的模板形参会屏蔽全局名字：

```
typedef double T;
template <class T> T calc(const T &a, const T &b)
{
    // tmp has the type of the template parameter T
    // not that of the global typedef
    T tmp = a;
    // ...
    return tmp;
}
```

将 `T` 定义为 `double` 的全局类型别名将被名为 `T` 的类型形参所屏蔽，因此，`tmp` 不是 `double` 型，相反，`tmp` 的类型是绑定到模板形参的任意类型。

2. 使用模板形参名字的限制

用作模板形参的名字不能在模板内部重用：

```
template <class T> T calc(const T &a, const T &b)
{
    typedef double T; // error: redeclares template parameter T
    T tmp = a;
    // ...
    return tmp;
}
```

这一限制还意味着模板形参的名字只能在同一模板形参表中使用一次：

```
// error: illegal reuse of template parameter name V
template <class V, class V> V calc(const V&, const V&) ;
```

当然，正如可以重用函数形参名字一样，模板形参的名字也能在不同模板中重用：

```
// ok: reuses parameter type name across different templates
template <class T> T calc (const T&, const T&) ;
template <class T> int compare(const T&, const T&) ;
```

3. 模板声明

像其他任意函数或类一样，对于模板可以只声明而不定义。声明必须指出函数或类是一个模板：

```
// declares compare but does not define it
template <class T> int compare(const T&, const T&) ;
```

629

同一模板的声明和定义中，模板形参的名字不必相同。

```
// all three uses of calc refer to the same function template
// forward declarations of the template
template <class T> T calc(const T&, const T&) ;
template <class U> U calc(const U&, const U&) ;
// actual definition of the template
template <class Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

每个模板类型形参前面必须带上关键字 `class` 或 `typename`，每个非类型形参前面必须带上类型名字，省略关键字或类型说明符是错误的：

```
// error: must precede U by either typename or class
template <typename T, U> T calc (const T&, const U&) ;
```

习题

习题 16.7 解释下面每个函数模板的定义并指出是否有非法的。改正所发现的错误。

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

习题 16.8 如果有，解释下面哪些声明是错误的并说明为什么。

- (a) `template <class Type> Type bar(Type, Type) ;`
`template <class Type> Type bar(Type, Type) ;`
- (b) `template <class T1, class T2> void bar(T1, T2) ;`
`template <class C1, typename C2> void bar(C1, C2) ;`

习题 16.9 编写行为类似于标准库中 `find` 算法的模板。你的模板应接受一个类型形参，该形参指定函数形参（一对迭代器）的类型。使用你的函数在 `vector<int>` 和 `vector<string>` 中查找给定值。

16.1.4 模板类型形参

类型形参由关键字 `class` 或 `typename` 后接说明符构成。在模板形参表中，这两个关键字具有相同的含义，都指出后面所接的名字表示一个类型。

模板类型形参可作为类型说明符用在模板中的任何地方，与内置类型说明符或类类型说明符的使用方式完全相同。具体而言，它可以用于指定返回类型或函数形参类型，以及在函数体中用于变量声明或强制类型转换。

```
// ok: same type used for the return type and both parameters
template <class T> T calc (const T& a, const T& b)
{
    // ok: tmp will have same type as the parameters & return type
    T tmp = a;
    // ...
    return tmp;
}
```

1. typename 与 class 的区别

在函数模板形参表中，关键字 `typename` 和 `class` 具有相同含义，可以互换使用，两个关键字都可以在同一模板形参表中使用：

```
// ok: no distinction between typename and class in template parameter list
template <typename T, class U> calc (const T&, const U&);
```

使用关键字 `typename` 代替关键字 `class` 指定模板类型形参也许更为直观，毕竟，可以使用内置类型（非类类型）作为实际的类型形参，而且，`typename` 更清楚地指明后面的名字是一个类型名。但是，关键字 `typename` 是作为标准 C++ 的组成部分加入到 C++ 中的，因此旧的程序更有可能只用关键字 `class`。

2. 在模板定义内部指定类型

除了定义数据成员或函数成员之外，类还可以定义类型成员。例如，标准库的容器类定义了不同的类型，如 `size_type`，使我们能够以独立于机器的方式使用容器。如果要在函数模板内部使用这样的类型，必须告诉编译器我们正在使用的名字指的是一个类型。必须显式地这样做，因为编译器（以及程序的读者）不能通过检查得知，由类型形参定义的名字何时是一个类型何时是一个值。例如，考虑下面的函数：

```
template <class Parm, class U>
Parm fcn(Parm* array, U value)
{
    Parm::size_type * p; // If Parm::size_type is a type, then a declaration
                        // If Parm::size_type is an object, then multiplication
}
```

631

我们知道 `size_type` 必定是绑定到 `Parm` 的那个类型的成员，但我们不知道 `size_type` 是一个类型成员的名字还是一个数据成员的名字，默认情况下，编译器假定这样的名字指定数据成员，而不是类型。

如果希望编译器将 `size_type` 当作类型，则必须显式告诉编译器这样做：

```
template <class Parm, class U>
Parm fcn(Parm* array, U value)
{
    typename Parm::size_type * p; // ok: declares p to be a pointer
}
```

通过在成员名前加上关键字 `typename` 作为前缀，可以告诉编译器将成员当作类型。通过编写 `typename Parm::size_type`，指出绑定到 `Parm` 的类型的 `size_type` 成员是类型的名字。当然，这一声明给用来实例化 `fcn` 的类型增加了一个职责：那些类型必须具有名为 `size_type` 的成员，而且该成员是一个类型。



如果拿不准是否需要以 `typename` 指明一个名字是一个类型，那么指定它是个好主意。在类型之前指定 `typename` 没有害处，因此，即使 `typename` 是不必要的，也没有关系。

习题

- 习题 16.10 声明为 `typename` 的类型形参与声明为 `class` 的类型形参有区别吗？区别在哪里？
- 习题 16.11 何时必须使用 `typename`？
- 习题 16.12 编写一个函数模板，接受表示未知类型迭代器的一对值，找出在序列中出现得最频繁的值。
- 习题 16.13 编写一个函数，接受一个容器的引用并打印该容器的元素。使用容器的 `size_type` 和 `size` 成员控制打印元素的循环。
- 习题 16.14 重新编写上题的函数，使用从 `begin` 和 `end` 返回的迭代器来控制循环。

16.1.5 非类型模板形参

模板形参不必都是类型。本节将介绍函数模板使用的非类型形参。在介绍了类模板实现的更多内容之后，16.4.2 节将介绍类模板的非类型形参。

在调用函数时非类型形参将用值代替，值的类型在模板形参表中指定。例如，下面的函数模板声明了 `array_init` 是一个含有一个类型模板形参和一个非类型模板形参的函数模板。函数本身接受一个形参，该形参是数组的引用（7.2.4 节）：

```
// initialize elements of an array to zero
template <class T, size_t N> void array_init(T (&parm)[N])
{
    for (size_t i = 0; i != N; ++i) {
        parm[i] = 0;
    }
}
```

模板非类型形参是模板定义内部的常量值，在需要常量表达式的时候，可使用非类型形参（例如，像这里所做的一样）指定数组的长度。

当调用 `array_init` 时，编译器从数组实参计算非类型形参的值：

```
int x[42];
double y[10];
array_init(x); // instantiates array_init(int(&)[42])
array_init(y); // instantiates array_init(double(&)[10])
```

编译器将为 `array_init` 调用中用到的每种数组实例化一个 `array_init` 版本。对于上面的程序，编译器将实例化 `array_init` 的两个版本：第一个实例的形参绑定到 `int[42]`，另一个实例中的形参绑定到 `double[10]`。

类型等价性与非类型形参

对模板的非类型形参而言，求值结果相同的表达式将认为是等价的。下面的两个 `array_init` 调用引用的是相同的实例——`array_init<int, 42>`：

```
int x[42];
const int sz = 40;
int y[sz + 2];
array_init(x); // instantiates array_init(int(&)[42])
array_init(y); // equivalent instantiation
```

习题

习题 16.15 编写可以确定数组长度的函数模板。

习题 16.16 将 7.2.4 节的 printValues 函数重新编写为可用于打印不同长度数组内容的函数模板。

16.1.6 编写泛型程序

编写模板时，代码不可能针对特定类型，但模板代码总是要对将使用的类型做一些假设。例如，虽然 compare 函数从技术上说对任意类型都是有效的，但实际上，实例化的版本可能是非法的。

633

产生的程序是否合法，取决于函数中使用的操作以及所用类型支持的操作。compare 函数有三条语句：

```
if (v1 < v2) return -1; // < on two objects of type T
if (v2 < v1) return 1;  // < on two objects of type T
return 0;               // return int; not dependent on T
```

前两条语句包含隐式依赖于形参类型的代码，if 测试对形参使用 < 操作符，直到编译器看见 compare 调用并且 T 绑定到一个实际类型时，才知道形参的类型，使用哪个 < 操作符完全取决于实参类型。

如果用不支持 < 操作符的对象调用 compare，则该调用将是无效的：

```
Sales_item item1, item2;
// error: no < on Sales_item
cout << compare(item1, item2) << endl;
```

程序会出错。Sales_item 类型没有定义 < 操作符，所以该程序不能编译。



在函数模板内部完成的操作限制了可用于实例化该函数的类型。程序员的责任是，保证用作函数实参的类型实际上支持所用的任意操作，以及保证在模板使用那些操作的环境中那些操作运行正常。

编写独立于类型的代码

编写良好泛型代码的技巧超出了本书的范围，但是，有个一般原则值得注意。



编写模板代码时，对实参类型的要求尽可能少是很有益的。

虽然简单，但它说明了编写泛型代码的两个重要原则：

- 模板的形参是 const 引用。
- 函数体中的测试只用 < 比较。

通过将形参设为 const 引用，就可以允许使用不允许复制的类型。大多数类型（包括内置类型和我们已使用过的除 IO 类型之外的所有标准库的类型）都允许复制。但是，也有不允许复制的类类型。将形参设为 const 引用，保证这种类型可以用于 compare 函数，而且，如果有比较大的对象调用 compare，则这个设计还可以使函数运行得更快。

一些读者可能认为使用<和>操作符两者进行比较会更加自然:

```
// expected comparison
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

但是, 将代码编写为

```
// expected comparison
if (v1 < v2) return -1;
if (v2 < v1) return 1; // equivalent to v1 > v2
return 0;
```

可以减少对可用于 compare 函数的类型的要求, 这些类型必须支持<, 但不必支持>。

习题

习题 16.17 在 3.3.2 节的“关键概念”中, 我们注意到, C++程序员习惯于使用!=而不用<, 解释这一习惯的基本原理。

习题 16.18 本节中我们提到应该慎重地编写 compare 中的比较以避免要求类型同时具有<和>操作符, 另一方面, 往往假定类型既有==又有!=。解释为什么这一看似不一致的处理实际上反映了良好的编程风格。

警告: 链接时的编译时错误

一般而言, 编译模板时, 编译器可能会在三个阶段中标识错误: 第一阶段是编译模板定义本身时。在这个阶段中编译器一般不能发现许多错误, 可以检测到诸如漏掉分号或变量名拼写错误一类的语法错误。

第二个错误检测时间是在编译器见到模板的使用时。在这个阶段, 编译器仍没有很多检查可做。对于函数模板的调用, 许多编译器只检查实参的数目和类型是否恰当, 编译器可以检测到实参太多或太少, 也可以检测到假定类型相同的两个实参是否真地类型相同。对于类模板, 编译器可以检测提供的模板实参的正确数目。

产生错误的第三个时间是在实例化的时候, 只有在这个时候可以发现类型相关的错误。根据编译器管理实例化的方式(将在 16.3 节讨论), 有可能在链接时报告这些错误。

重要的是, 要认识到在编译模板定义的时候, 对程序是否有效所知不多。类似地, 甚至可能会在已经成功编译了使用模板的每个文件之后出现编译错误。只在实例化期间检测错误的情况很少, 错误检测可能发生在链接时。

16.2 实例化

模板是一个蓝图, 它本身不是类或函数。编译器用模板产生指定的类或函数的特定类型版本。产生模板的特定类型实例的过程称为实例化, 这个术语反映了创建模板类型或模板函数的新“实

例”的概念。

模板在使用时将进行实例化，类模板在引用实际模板类类型时实例化，函数模板在调用它或用它对函数指针进行初始化或赋值时实例化。

1. 类的实例化

当编写 `Queue<int> qi;` 时，编译器自动创建名为 `Queue<int>` 的类。实际上，编译器通过重新编写 `Queue` 模板，用类型 `int` 代替模板形参的每次出现而创建 `Queue<int>` 类。实例化的类就像已经编写的一样：

```
// simulated version of Queue instantiated for type int
template <class Type> class Queue<int> {
public:
    Queue();                // this bound to Queue<int>*
    int &front();            // return type bound to int
    const int &front() const; // return type bound to int
    void push(const int &);  // parameter type bound to int
    void pop();              // type invariant code
    bool empty() const;     // type invariant code
private:
    // ...
};
```

要为 `string` 类型的对象创建 `Queue` 类，可以编写

```
Queue<string> qs;
```

在这个例子中，用 `string` 代替 `Type` 的每次出现。



类模板的每次实例化都会产生一个独立的类类型。为 `int` 类型实例化的 `Queue` 与任意其他 `Queue` 类型没有关系，对其他 `Queue` 类型的成员也没有特殊的访问权。

2. 类模板形参是必需的

想要使用类模板，就必须显式指定模板实参：

```
Queue qs; // error: which template instantiation?
```

636

类模板不定义类型，只有特定的实例才定义了类型。特定的实例化是通过提供模板实参与每个模板形参匹配而定义的。模板实参在用逗号分隔并用尖括号括住的列表中指定：

```
Queue<int> qi;          // ok: defines Queue that holds ints
Queue<string> qs;       // ok: defines Queue that holds strings
```

用模板类定义的类型总是包含模板实参。例如，`Queue` 不是类型，而 `Queue<int>` 或 `Queue<string>` 是类型。

3. 函数模板实例化

使用函数模板时，编译器通常会为我们推断模板实参：

```
int main()
{
    compare(1, 0);          // ok: binds template parameter to int
    compare(3.14, 2.7);     // ok: binds template parameter to double
    return 0;
}
```

```
}
```

这个程序实例化了 `compare` 的两个版本：一个用 `int` 代替 `T`，另一个用 `double` 代替 `T`，实质上是编译器为我们编写了 `compare` 的这两个实例：

```
int compare(const int &v1, const int &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;
}
```

16.2.1 模板实参推断

要确定应该实例化哪个函数，编译器会查看每个实参。如果相应形参声明为类型形参的类型，则编译器从实参的类型推断形参的类型。在 `compare` 的例子中，两个实参有同样的模板类型，都是用类型形参 `T` 声明的。

第一个调用 `compare(1, 0)` 中，实参为 `int` 类型；第二个调用 `compare(3.14, 2.7)` 中，实参为 `double` 类型。从函数实参确定模板实参的类型和值的过程叫做**模板实参推断**（`template argument deduction`）。

637

1. 多个类型形参的实参必须完全匹配

模板类型形参可以用作一个以上函数形参的类型。在这种情况下，模板类型推断必须为每个对应的函数实参产生相同的模板实参类型。如果推断的类型不匹配，则调用将会出错：

```
template <typename T>
int compare(const T& v1, const T& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int main()
{
    short si;
    // error: cannot instantiate compare(short, int)
    // must be: compare(short, short) or
    //           compare(int, int)
    compare(si, 1024);
    return 0;
}
```

这个调用是错误的，因为调用 `compare` 时的实参类型不相同，从第一个实参推断出的模板类型是 `short`，从第二个实参推断出 `int` 类型，两个类型不匹配，所以模板实参推断失败。

如果 `compare` 的设计者想要允许实参的常规转换，则函数必须用两个类型形参来定义：

```
// argument types can differ, but must be compatible
template <typename A, typename B>
int compare(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：

```
short si;
compare(si, 1024); // ok: instantiates compare(short, int)
```

但是，比较那些类型的值的<操作符必须存在。

2. 类型形参的实参的受限转换

考虑下面的 compare 调用：

```
short s1, s2;
int i1, i2;
compare(i1, i2);           // ok: instantiate compare(int, int)
compare(s1, s2);          // ok: instantiate compare(short, short)
```

638

第一个调用产生将 T 绑定到 int 的实例，为第二个调用创建新的实例，将 T 绑定到 short。

如果 compare(int, int) 是普通的非模板函数，则第二个调用会匹配那个函数，short 实参将提升（5.12.2 节）为 int。因为 compare 是一个模板，所以将实例化一个新函数，将类型形参绑定到 short。

一般而言，不会转换实参以匹配已有的实例化，相反，会产生新的实例。除了产生新的实例化之外，编译器只会执行两种转换：

- **const 转换：**接受 const 引用或 const 指针的函数可以分别用非 const 对象的引用或指针来调用，无须产生新的实例化。如果函数接受非引用类型，形参类型和实参都忽略 const，即，无论传递 const 或非 const 对象给接受非引用类型的函数，都使用相同的实例化。
- **数组或函数到指针的转换：**如果模板形参不是引用类型，则对数组或函数类型的实参应用常规指针转换。数组实参将当作指向其第一个元素的指针，函数实参当作指向函数类型的指针。

例如，考虑对函数 fobj 和 fref 的调用。fobj 函数复制它的形参，而 fref 的形参是引用：

```
template <typename T> T fobj(T, T);           // arguments are copied
template <typename T>
T fref(const T&, const T&);                 // reference arguments
string s1("a value");
const string s2("another value");
fobj(s1, s2);           // ok: calls f(string, string), const is ignored
fref(s1, s2);           // ok: non const object s1 converted to const reference
int a[10], b[42];
fobj(a, b); // ok: calls f(int*, int*)
fref(a, b); // error: array types don't match; arguments aren't converted to pointers
```

第一种情况下，传递 string 对象和 const string 对象作为实参，即使这些类型不完全匹配，两个调用也都是合法的。在 fobj 的调用中，实参被复制，因此原来的对象是否为 const 无关紧要。在 fref 的调用中，形参类型是 const 引用，对引用形参而言，转换为 const 是可以接受的转换，所以这个调用也正确。

在第二种情况中，将传递不同长度的数组实参。fobj 的调用中，数组不同无关紧要，两个数组都转换为指针，fobj 的模板形参类型是 int*。但是，fref 的调用是非法的，当形参为引用时（7.2.4 节），数组不能转换为指针，a 和 b 的类型不匹配，所以调用将出错。

639

3. 应用于非模板实参的常规转换



类型转换的限制只适用于类型为模板形参的那些实参。

用普通类型定义的形参可以使用常规转换（7.1.2 节），下面的函数模板 sum 有两个形参：

```
template <class Type> Type sum(const Type &op1, int op2)
{
    return op1 + op2;
}
```

第一个形参 op1 具有模板形参类型，它的实际类型到函数使用时才知道。第二个形参 op2 的类型已知，为 int。

因为 op2 的类型是固定的，在调用 sum 的时候，可以对传递给 op2 的实参应用常规转换：

```
double d = 3.14;
string s1("hiya"), s2(" world");
sum(1024, d); // ok: instantiates sum(int, int), converts d to int
sum(1.4, d);  // ok: instantiates sum(double, int), converts d to int
sum(s1, s2); // error: s2 cannot be converted to int
```

在前两个调用中，第二个实参 dd 的类型与对应函数形参的类型不同，但是，这些调用是正确的：有从 double 到 int 的转换。因为第二个形参的类型独立于模板形参，编译器将隐式转换 dd。第一个调用导致实例化函数 sum(int, int)，第二个调用实例化 sum(double, int)。

第三个调用是错误的。没有从 string 到 int 的转换，使用 string 实参来匹配 int 形参与一般情况一样，是非法的。

4. 模板实参推断与函数指针

可以使用函数模板对函数指针进行初始化或赋值（7.9 节），这样做的时候，编译器使用指针的类型实例化具有适当模板实参的模板版本。

例如，假定有一个函数指针指向返回 int 值的函数，该函数接受两个形参，都是 const int 引用，可以用该指针指向 compare 的实例化：

```
template <typename T> int compare(const T&, const T&);
// pf1 points to the instantiation int compare (const int&, const int&)
int (*pf1) (const int&, const int&) = compare;
```

pf1 的类型是一个指针，指向“接受两个 const int& 类型形参并返回 int 值的函数”，形参的类型决定了 T 的模板实参的类型，T 的模板实参为 int 型，指针 pf1 引用的是将 T 绑定到 int 的实

640 例化。



获取函数模板实例化的地址的时候，上下文必须是这样的：它允许为每个模板形参确定唯一的类型或值。

如果不能从函数指针类型确定模板实参，就会出错。例如，假定有两个名为 func 的函数，每个函数接受一个指向函数实参的指针。func 的第一个版本接受有两个 const string 引用形参并返回 string 对象的函数的指针，func 的第二个版本接受带两个 const int 引用形参并返回 int 值的函数的指针，不能使用 compare 作为传给 func 的实参：

```
// overloaded versions of func; each take a different function pointer type
void func(int(*) (const string&, const string&));
void func(int(*) (const int&, const int&));
func(compare); // error: which instantiation of compare?
```

问题在于，通过查看 func 的形参类型不可能确定模板实参的唯一类型，对 func 的调用可以实例化下列函数中的任意一个：

```
compare(const string&, const string&)
compare(const int&, const int&)
```

因为不能为传给 func 的实参确定唯一的实例化，该调用会产生一个编译时（或链接时）错误。

习题

习题 16.19 什么是实例化？

习题 16.20 在模板实参推断期间发生什么？

习题 16.21 指出对模板实参推断中涉及的函数实参允许的类型转换。

习题 16.22 对于下面的模板

```
template <class Type>
Type calc (const Type* array, int size);
template <class Type>
Type fcn(Type p1, Type p2;
```

下面这些调用有错吗？如果有，哪些是错误的？为什么？

```
double dobj;    float fobj;    char cobj;
int ai[5] = { 511, 16, 8, 63, 34 };
```

- (a) calc(cobj, 'c');
- (b) calc(dobj, fobj);
- (c) fcn(ai, cobj);

641

16.2.2 函数模板的显式实参

在某些情况下，不可能推断模板实参的类型。当函数的返回类型必须与形参表中所用的所有类型都不同时，最常出现这一问题。在这种情况下，有必要覆盖模板实参推断机制，并显式指定为模板形参所用的类型或值。

1. 指定显式模板实参

考虑下面的问题。我们希望定义名为 `sum`、接受两个不同类型实参的函数模板，希望返回类型足够大，可以包含按任意次序传递的任意两个类型的两个值的和，怎样才能做到？应如何指定 `sum` 的返回类型？

```
// T or U as the return type?
template <class T, class U> ??? sum(T, U);
```

在这个例子中，答案是没有一个形参在任何时候都可行，使用任一形参都一定会在某些时候失败：

```
// neither T nor U works as return type
sum(3, 4L); // second type is larger; want U sum(T, U)
sum(3L, 4); // first type is larger; want T sum(T, U)
```

解决这一问题的一个办法，可能是强制 `sum` 的调用者将较小的类型强制转换（5.12.4 节）为希望作为结果使用的类型：

```
// ok: now either T or U works as return type
int i; short s;
sum(static_cast<int>(s), i); // ok: instantiates int sum(int, int)
```

2. 在返回类型中使用类型形参

指定返回类型的一种方式是引入第三个模板形参，它必须由调用者显式指定：

```
// T1 cannot be deduced: it doesn't appear in the function parameter list
template <class T1, class T2, class T3>
T1 sum(T2, T3);
```

这个版本增加了一个模板形参以指定返回类型。只有一个问题：没有实参的类型可用于推断 `T1` 的类型，相反，调用者必须在每次调用 `sum` 时为该形参显式提供实参。

为调用提供显式模板实参与定义类模板的实例很类似，在以逗号分隔、用尖括号括住的列表中指定显式模板实参。显式模板类型的列表出现在函数名之后、实参表之前：

```
// ok T1 explicitly specified; T2 and T3 inferred from argument types
long val3 = sum<long>(i, lng); // ok: calls long sum(int, long)
```

这一调用显式指定 `T1` 的类型，编译器从调用中传递的实参推断 `T2` 和 `T3` 的类型。

642

显式模板实参从左至右与对应模板形参相匹配，第一个模板实参与第一个模板形参匹配，第二个实参与第二个形参匹配，以此类推。假如可以从函数形参推断，则只有结尾（最右边）形参的显式模板实参可以省略。如果这样编写 `sum` 函数：

```
// poor design: Users must explicitly specify all three template parameters
template <class T1, class T2, class T3>
T3 alternative_sum(T2, T1);
```

则总是必须为所有三个形参指定实参：

```
// error: can't infer initial template parameters
long val3 = alternative_sum<long>(i, lng);
// ok: All three parameters explicitly specified
long val2 = alternative_sum<long, int, long>(i, lng);
```

3. 显式实参与函数模板的指针

可以使用显式模板实参的另一个例子是 16.2.1 节中有二义性的程序, 通过使用显式模板实参能够消除二义性:

```
template <typename T> int compare(const T&, const T&);
// overloaded versions of func; each take a different function pointer type
void func(int(*) (const string&, const string&));
void func(int(*) (const int&, const int&));
func(compare<int>); // ok: explicitly specify which version of compare
```

像前面一样, 需要在调用中传递 `compare` 实例给名为 `func` 的重载函数。只查看不同版本 `func` 的形参表来选择传递 `compare` 的哪个实例是不可能的, 两个不同的实例都可能满足该调用。显式模板形参需要指出应使用哪个 `compare` 实例以及调用哪个 `func` 函数。

习题

习题 16.23 标准库函数 `max` 接受单个类型形参, 可以传递 `int` 和 `double` 对象调用 `max` 吗? 如果可以, 怎样做? 如果不能, 为什么?

习题 16.24 在 16.2.1 节我们看到, 对于具有单个模板类型形参的 `compare` 版本, 传给它的实参必须完全匹配, 如果想要用兼容类型如 `int` 和 `short` 调用该函数, 可以使用显式模板实参指定 `int` 或 `short` 作为形参类型。编写程序使用具有一个模板形参的 `compare` 版本, 使用允许你传递 `int` 和 `short` 类型实参的显式模板实参调用 `compare`。

习题 16.25 使用显式模板实参, 传递两个字符串字面值调用 `compare` 是切合实际的。

习题 16.26 对于下面的 `sum` 模板定义:

```
template <class T1, class T2, class T3> T1 sum(T2, T3);
```

解释下面的每个调用, 是否有错? 如果有, 指出哪些是错误的, 对每个错误, 解释错在哪里。

```
double dobj1, dobj2; float fobj1, fobj2; char cobj1, cobj2;
```

- (a) `sum(dobj1, dobj2);`
- (b) `sum<double, double, double>(fobj1, fobj2);`
- (c) `sum<int>(cobj1, cobj2);`
- (d) `sum<double, ,double>(fobj2, dobj2);`

16.3 模板编译模型

当编译器看到模板定义的时候, 它不立即产生代码。只有在看到用到模板时, 如调用了函数模板或调用了类模板的对象的时候, 编译器才产生特定类型的模板实例。

一般而言, 当调用函数的时候, 编译器只需要看到函数的声明。类似地, 定义类类型的对象时, 类定义必须可用, 但成员函数的定义不是必须存在的。因此, 应该将类定义和函数声明放在头文件中, 而普通函数和类成员函数的定义放在源文件中。

643

模板则不同: 要进行实例化, 编译器必须能够访问定义模板的源代码。当调用函数模板或类

模板的成员函数的时候，编译器需要函数定义，需要那些通常放在源文件中的代码。

标准 C++ 为编译模板代码定义了两种模型。在两种模型中，构造程序的方式很大程度上是相同的：类定义和函数声明放在头文件中，而函数定义和成员定义放在源文件中。两种模型的不同在于，编译器怎样使用来自源文件的定义。如本书所述，所有编译器都支持第一种模型，称为“包含”模型，只有一些编译器支持第二种模型，“分别编译”模型。



要编译使用自己的类模板和函数模板的代码，必须查阅编译器的用户指南，看看编译器怎样处理实例化。

1. 包含编译模型

在包含编译模型 (inclusion compilation model) 中，编译器必须看到用到的所有模板的定义。一般而言，可以通过在声明函数模板或类模板的头文件中添加一条 `#include` 指示使定义可用，该 `#include` 引入了包含相关定义的源文件：

644

```
// header file utilities.h
#ifndef UTILITIES_H // header guard (Section 2.9.2, p. 69)
#define UTILITIES_H
template <class T> int compare(const T&, const T&);
// other declarations

#include "utilities.cc" // get the definitions for compare etc.
#endif

// implementation file utilities.cc
template <class T> int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
// other definitions
```

这一策略使我们能够保持头文件和实现文件的分离，但是需要保证编译器在编译使用模板的代码时能看到两种文件。

某些使用包含模型的编译器，特别是较老的编译器，可以产生多个实例。如果两个或多个单独编译的源文件使用同一模板，这些编译器将为每个文件中的模板产生一个实例。通常，这种方法意味着给定模板将实例化超过一次。在链接的时候，或者在预链接阶段，编译器会选择一个实例化而丢弃其他的。在这种情况下，如果有许多实例化同一模板的文件，编译时性能会显著降低。对许多应用程序而言，这种编译时性能降低不大可能在现代计算机上成为问题，但是，在大系统环境中，编译时选择问题可能变得非常重要。

这种编译器通常支持某些机制，避免同一模板的多个实例化中隐含的编译时开销。编译器优化编译时性能的方法各不相同。如果使用模板的程序的编译时间难于承担，请查阅编译器的用户指南，看看你的编译器能提供什么支持以避免多余的实例化。

2. 分别编译模型

在分别编译模型 (separate compilation model) 中，编译器会为我们跟踪相关的模板定义。但

是，我们必须让编译器知道要记住给定的模板定义，可以使用 **export** 关键字 (export keyword) 来做这件事。

645

export 关键字能够指明给定的定义可能会需要在其他文件中产生实例化。在一个程序中，一个模板只能定义为导出一次。编译器在需要产生这些实例化时计算出怎样定位模板定义。export 关键字不必在模板声明中出现。

一般我们在函数模板的定义中指明函数模板为导出的，这是通过在关键字 **template** 之前包含 **export** 关键字而实现的：

```
// the template definition goes in a separately-compiled source file
export template <typename Type>
Type sum(Type t1, Type t2) /* ... */
```

这个函数模板的声明像通常一样应放在头文件中，声明不必指定 **export**。

对类模板使用 **export** 更复杂一些。通常，类声明必须放在头文件中，头文件中的类定义体不应该使用关键字 **export**，如果在头文件中使用了 **export**，则该头文件只能被程序中的一个源文件使用。

相反，应该在类的实现文件中使用 **export**：

```
// class template header goes in shared header file
template <class Type> class Queue { ... };
// Queue.cc implementation file declares Queue as exported
export template <class Type> class Queue;
#include "Queue.h"
// Queue member definitions
```

导出类的成员将自动声明为导出的。也可以将类模板的个别成员声明为导出的，在这种情况下，关键字 **export** 不在类模板本身指定，而是只在被导出的特定成员定义上指定。导出成员函数的定义不必在使用成员时可见。任意非导出成员的定义必须像在包含模型中一样对待：定义应放在定义类模板的头文件中。

习题

习题 16.27 确定你的编译器使用的是哪种编译模型。编写并调用函数模板，在保存未知类型对象的 **vector** 中查找中间值。（注：中间值是这样一个值，一半元素比它大，一半元素比它小。）用常规方式构造你的程序：函数定义应放在一个文件中，它的声明放在一个头文件中，定义和使用函数模板的代码应包含该头文件。

习题 16.28 如果所用的编译器支持分别编译模型，将类模板的成员函数和 **static** 数据成员的定义放在哪里？为什么？

646

习题 16.29 如果你的编译器使用包含模型，将那些模板成员定义放在哪里？为什么？

警告：类模板中的名字查找

编译模板是异常困难的工作。幸好，它是由编译器作者处理的任务。不幸的是，某些复杂性被推到模板用户的身上：模板包含两种名字：

(1) 独立于模板形参的那些名字

(2) 依赖于模板形参的那些名字

设计者的责任是，保证所有不依赖于模板形参的名字在模板本身的作用域中定义。

模板用户的责任是，保证与用来实例化模板的类型相关的所有函数、类型和操作符的声明可见。这个责任意味着，在实例化类模板的成员或函数模板的时候，用户必须保证这些声明是可见的。

适当使用头文件的结构良好的程序都容易满足这两个要求。模板的作者应提供头文件，该头文件包含在类模板或在其成员定义中使用的所有名字的声明。在用特定类型定义模板或者使用该模板的成员之前，用户必须保证包含了模板类型的头文件，以及定义用作成员类型的类型的头文件。

16.4 类模板成员

到目前为止，我们只介绍了怎样声明 Queue 类模板的接口成员，本节将介绍怎样实现该类。



标准库将 queue 实现为其他容器之上的适配器 (9.7 节)。为了强调在使用低级数据结构中涉及的编程要点，我们将 Queue 实现为链表。实际上，在我们的实现中使用标准库容器可能是一个更好的决定。

1. Queue 的实现策略

如图 16-1 所示，我们的实现使用两个类：

(1) QueueItem 类表示 Queue 的链表中的节点，该类有两个数据成员 item 和 next：

- item 保存 Queue 中元素的值，它的类型随 Queue 的每个实例而变化。
- next 是队列中指向下一 QueueItem 对象的指针。

Queue 中的每个元素保存在一个 QueueItem 对象中。

(2) Queue 类将提供 16.1.2 节描述的接口函数，Queue 类也有两个数据成员：head 和 tail，这些成员是 QueueItem 指针。

像标准容器一样，Queue 类将复制指定给它的值。

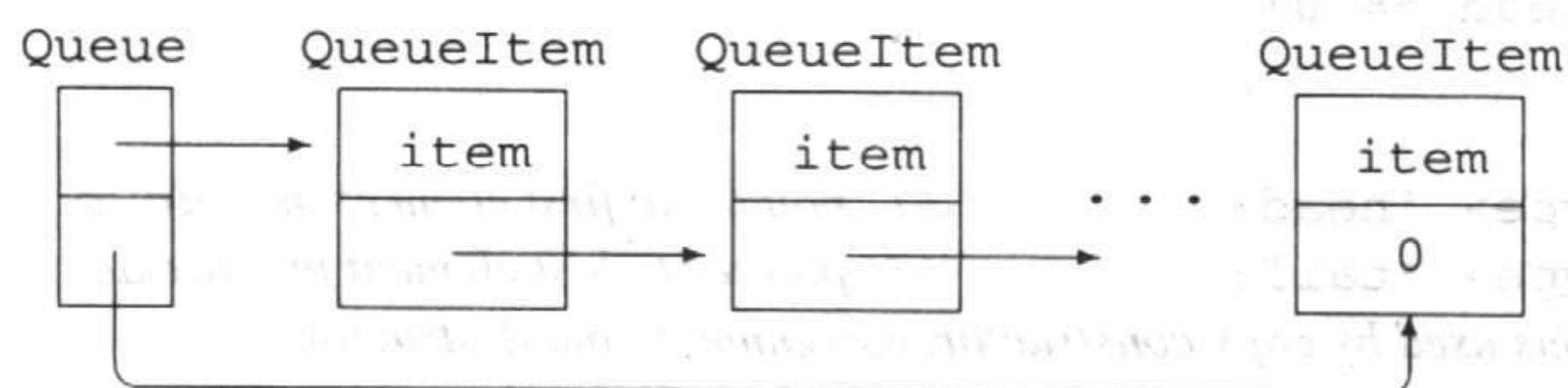


图 16-1 Queue 的实现

2. QueueItem 类

首先编写 QueueItem 类：

```
template <class Type> class QueueItem {
// private class: no public section
```

```

    QueueItem(const Type &t): item(t), next(0) { }
    Type item;                // value stored in this element
    QueueItem *next;          // pointer to next element in the Queue
};

```

这个类似乎已经差不多完整了：它保存由其构造函数初始化的两个数据成员。像 Queue 类一样，QueueItem 是一个类模板，该类使用模板形参指定 item 成员的类型，Queue 中每个元素的值将保存在 item 中。

每当实例化一个 Queue 类的时候，也将实例化 QueueItem 的相同版本。例如，如果创建 Queue<int>，则将实例化一个伙伴类 QueueItem<int>。

QueueItem 类为私有类——它没有公用接口。我们这个类只是为了实现 Queue，并不想用于一般目的，因此，它没有公用成员。需要将 Queue 类设为 QueueItem 类的友元，以便 Queue 类的成员能够访问 QueueItem 的成员。16.4.4 节将介绍怎样做。



在类模板的作用域内部，可以用它的非限定名字引用该类。

3. Queue 类

现在充实 Queue 类：

```

template <class Type> class Queue {
public:
    // empty Queue
    Queue(): head(0), tail(0) { }
    // copy control to manage pointers to QueueItems in the Queue
    Queue(const Queue &Q): head(0), tail(0)
                                { copy_elems(Q); }
    Queue& operator=(const Queue&);
    ~Queue() { destroy(); }
    // return element from head of Queue
    // unchecked operation: front on an empty Queue is undefined
    Type& front() { return head->item; }
    const Type &front() const { return head->item; }
    void push(const Type &); // add element to back of Queue
    void pop ();             // remove element from head of Queue
    bool empty () const {   // true if no elements in the Queue
        return head == 0;
    }
private:
    QueueItem<Type> *head; // pointer to first element in Queue
    QueueItem<Type> *tail; // pointer to last element in Queue
    // utility functions used by copy constructor, assignment, and destructor
    void destroy(); // delete all the elements
    void copy_elems(const Queue&); // copy elements from parameter
};

```

除了接口成员之外，还增加了三个复制控制成员（第 13 章）以及那些成员所用的相关实用函数。private 实用函数 destroy 和 copy_elems 将完成释放 Queue 中的元素以及从另一 Queue 复制元素到这个 Queue 的任务。复制控制成员用于管理数据成员 head 和 tail，head 和 tail

是指向 Queue 中首尾元素的指针，这些成员是 QueueItem<Type>类型的值。

Queue 类实现了几个成员函数：

- 默认构造函数，将 head 和 tail 指针置 0，指明当前 Queue 为空。
- 复制构造函数，初始化 head 和 tail，并调用 copy_elems 从它的初始器复制元素。
- 几个 front 函数，返回头元素的值。这些函数不进行检查：像标准 queue 中的类似操作一样，用户不能在空 Queue 上运行 front 函数。
- empty 函数，返回 head 与 0 的比较结果。如果 head 为 0，Queue 为空；否则，Queue 是非空的。

4. 模板作用域中模板类型的引用

这个类的主要部分应该是我们熟悉的。它只与我们已经定义过的类有少许区别。新的内容是 Queue 类型和 QueueItem 类型的引用中对模板类型形参的使用（或缺少）。

通常，当使用类模板的名字的时候，必须指定模板形参。这一规则有个例外：在类本身的作用域内部，可以使用类模板的非限定名。例如，在默认构造函数和复制构造函数的声明中，名字 Queue 是 Queue<Type>的缩写表示。实质上，编译器推断，当我们引用类的名字时，引用的是同一版本。因此，复制构造函数定义其实等价于：

```
Queue<Type>(const Queue<Type> &Q): head(0), tail(0)
{ copy_elems(Q); }
```

编译器不会为类中使用的其他模板的模板形参进行这样的推断，因此，在声明伙伴类 QueueItem 的指针时，必须指定类型形参：

```
QueueItem<Type> *head;    // pointer to first element in Queue
QueueItem<Type> *tail;    // pointer to last element in Queue
```

这些声明指出，对于 Queue 类的给定实例化，head 和 tail 指向为同一模板形参实例化的 QueueItem 类型的对象，即，在 Queue<int>实例化的内部，head 和 tail 的类型是 QueueItem<int>。在 head 和 tail 成员的定义中省略模板形参将是错误的：

```
QueueItem *head;          // error: which version of QueueItem?
QueueItem *tail;          // error: which version of QueueItem?
```

习题

习题 16.30 如果有，指出下面类模板声明（或声明对）中哪些是非法的。

- template <class Type> class C1;
template <class Type, int size> class C1;
- template <class T, U, class V> class C2;
- template <class C1, typename C2> class C3 { };
- template <typename myT, class myT> class C4 { };
- template <class Type, int *ptr> class C5;
template <class T, int *pi> class C5;

习题 16.31 下面 List 的定义不正确，怎样改正？

```
template <class elemType> class ListItem;
template <class elemType> class List {
```

```

public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
    ListItem *find(elemType value);
private:
    ListItem *front;
    ListItem *end;
};

```

650

16.4.1 类模板成员函数

类模板成员函数的定义具有如下形式：

- 必须以关键字 `template` 开头，后接类的模板形参表。
- 必须指出它是哪个类的成员。
- 类名必须包含其模板形参。

从这些规则可以看到，在类外定义的 `Queue` 类的成员函数的开头应该是：

```
template <class T> ret-type Queue<T>::member-name
```

1. **destroy** 函数

为了举例说明在类外定义的类型模板成员函数，我们来看 `destroy` 函数：

```

template <class Type> void Queue<Type>::destroy()
{
    while (!empty())
        pop();
}

```

这个定义可以从左至右读作：

- 用名为 `Type` 的类型形参定义一个函数模板；
- 它返回 `void`；
- 它是在类模板 `Queue<Type>` 的作用域中。

在作用域操作符 (`::`) 之前使用的 `Queue<Type>` 指定成员函数所属的类。

跟在成员函数名之后的是函数定义。在 `destroy` 的例子中，函数体看来很像普通的非模板函数定义，它的工作是遍历这个 `Queue` 的每个分支，调用 `pop` 除去每一项。

2. **pop** 函数

`pop` 成员的作用是除去 `Queue` 的队头值：

```

template <class Type> void Queue<Type>::pop()
{
    // pop is unchecked: Popping off an empty Queue is undefined
    QueueItem<Type>* p = head; // keep pointer to head so we can delete it
    head = head->next;          // head now points to next element
    delete p;                  // delete old head element
}

```

651

pop 函数假设用户不会在空 Queue 上调用 pop。pop 的工作是除去 Queue 的头元素。必须重置 head 指针以指向 Queue 中的下一元素，然后删除 head 位置的元素。唯一有技巧的部分是记得保持指向该元素的一个单独指针，以便在重置 head 指针之后可以删除元素。

3. push 函数

push 成员将新项放在队列末尾：

```
template <class Type> void Queue<Type>::push(const Type &val)
{
    // allocate a new QueueItem object
    QueueItem<Type> *pt = new QueueItem<Type>(val);
    // put item onto existing queue
    if (empty())
        head = tail = pt; // the queue now has only one element
    else {
        tail->next = pt; // add new element to end of the queue
        tail = pt;
    }
}
```

这个函数首先分配新的 QueueItem 对象，用传递的值初始化它。这里实际上有些令人惊讶的工作，陈述如下：

- (1) QueueItem 构造函数将实参复制到 QueueItem 对象的 item 成员。像标准容器所做的一样，Queue 类存储所给元素的副本。
- (2) 如果 item 为类类型，item 的初始化使用 item 所具有任意类型的复制构造函数。
- (3) QueueItem 构造函数还将 next 指针初始化为 0，以指出该元素没有指向其他 QueueItem 对象。

因为将在 Queue 的末尾增加元素，将 next 置 0 正是我们所希望的。

创建和初始化新元素之后，必须将它链入 Queue。如果 Queue 为空，则 head 和 tail 都应该指向这个新元素。如果 Queue 中已经有元素了，则使当前 tail 元素指向这个新元素。旧的 tail 不再是最后一个元素了，这也是通过使 tail 指向新构造的元素指明的。

4. copy_elems¹ 函数

我们将赋值操作符的实现留作习题，剩下要编写的函数只有 copy_elems 了。设计该函数的目的是供赋值操作符和复制构造函数使用，它的工作是从形参中复制元素到这个 Queue：

```
template <class Type>
void Queue<Type>::copy_elems(const Queue &orig)
{
    // copy elements from orig into this Queue
    // loop stops when pt == 0, which happens when we reach orig.tail
    for (QueueItem<Type> *pt = orig.head; pt; pt = pt->next)
        push(pt->item); // copy the element
}
```

在 for 循环中复制元素，for 循环始于将 pt 设为等于形参的 head 指针。循环进行直至获得 orig 中最后一个元素之后，pt 为 0。对于 orig 中的每个元素，将该元素值的副本 push

1. 此处原文误为 copy。——译者注

到这个 Queue，并推进 pt 以指向 orig 中的下一元素。

5. 类模板成员函数的实例化

类模板的成员函数本身也是函数模板。像任何其他函数模板一样，需要使用类模板的成员函数产生该成员的实例化。与其他函数模板不同的是，在实例化类模板成员函数的时候，编译器不执行模板实参推断，相反，类模板成员函数的模板形参由调用该函数的对象的类型确定。例如，当调用 Queue<int> 类型对象的 push 成员时，实例化的 push 函数为

```
void Queue<int>::push(const int &val)
```

对象的模板实参能够确定成员函数模板形参，这一事实意味着，调用类模板成员函数比调用类似函数模板更灵活。用模板形参定义的函数形参的实参允许进行常规转换：

```
Queue<int> qi; // instantiates class Queue<int>
short s = 42;
int i = 42;
// ok: s converted to int and passed to push
qi.push(s); // instantiates Queue<int>::push(const int&)
qi.push(i); // uses Queue<int>::push(const int&)
f(s);      // instantiates f(const short&)
f(i);      // instantiates f(const int&)
```

6. 何时实例化类和成员

类模板的成员函数只有为程序所用才进行实例化。如果某函数从未使用，则不会实例化该成员函数。这一行为意味着，用于实例化模板的类型只需满足实际使用的操作的要求。9.1.1 节中只接受一个容量形参的顺序容器构造函数就是这样的例子，该构造函数使用元素类型的默认构造函数。如果有一个没有定义默认构造函数的类型，仍然可以定义容器来保存该类型，但是，不能使用只接受一个容量的构造函数。

定义模板类型的对象时，该定义导致实例化类模板。定义对象也会实例化用于初始化该对象的任一构造函数，以及该构造函数调用的任意成员：

```
// instantiates Queue<int> class and Queue<int>::Queue()
Queue<string> qs;
qs.push("hello"); // instantiates Queue<int>::push
```

第一个语句实例化 Queue<string> 类及其默认构造函数，第二个语句实例化 push 成员函数。

push 成员的实例化：

```
template <class Type> void Queue<Type>::push(const Type &val)
{
    // allocate a new QueueItem object
    QueueItem<Type> *pt = new QueueItem<Type>(val);
    // put item onto existing queue
    if (empty())
        head = tail = pt; // the queue now has only one element
    else {
        tail->next = pt; // add new element to end of the queue
        tail = pt;
    }
}
```

将依次实例化伙伴类 QueueItem<string> 及其构造函数。

Queue 类中的 QueueItem 成员是指针。类模板的指针定义不会对类进行实例化，只有用到这样的指针时才会对类进行实例化。因此，在创建 Queue 对象时不会实例化 QueueItem 类，相反，在使用诸如 front、push 或 pop 这样的 Queue 成员时才会实例化 QueueItem 类。

习题

习题 16.32 为 Queue 类实现赋值操作符。

习题 16.33 解释在 copy_elems 函数中新创建的 Queue 对象中的 next 指针怎样设置。

习题 16.34 编写 16.1.2 节习题中定义的 List 类的成员函数定义。

习题 16.35 编写 14.7 节中描述的 CheckedPtr 类的泛型版本。

654

16.4.2 非类型形参的模板实参

我们已经了解了如何实现类模板，现在来看看类模板的非类型形参。我们将为第 12 章引入的 Screen 类定义一个新版本，借此介绍类模板的非类型形参。在这个例子中，将 Screen 类重新定义为模板，以高度和宽度为形参。

```
template <int hi, int wid>
class Screen {
public:
    // template nontype parameters used to initialize data members
    Screen(): screen(hi * wid, '#'), cursor (0),
               height(hi), width(wid) { }
    // ...
private:
    std::string      screen;
    std::string::size_type cursor;
    std::string::size_type height, width;
};
```

这个模板有两个形参，均为非类型形参。当用户定义 Screen 对象时，必须为每个形参提供常量表达式以供使用。类在默认构造函数中使用这些形参设置默认 Screen 的尺寸。

像任意类模板一样，使用 Screen 类型时必须显式声明形参值：

```
Screen<24,80> hp2621; // screen 24 lines by 80 characters
```

对象 hp2621 使用模板实例化 Screen<24, 80>。hi 的模板实参是 24，而 wid 的模板实参是 80，两种情况下，模板实参都是常量表达式。



非类型模板实参必须是编译时常量表达式。

习题

习题 16.36 每个带标号的语句，会导致实例化吗？如果会，解释为什么。

```
template <class T> class Stack { };
void f1(Stack<char>); // (a)
class Exercise {
```

```

        Stack<double> &rsd;           // (b)
        Stack<int>     si;             // (c)
    };
    int main() {
        Stack<char> *sc;               // (d)
        fl(*sc);                      // (e)
        int iObj = sizeof(Stack< string >); // (f)
    }

```

习题 16.37 下面哪些模板实例化是有效的？解释为什么实例化无效。

```

template <class T, int size> class Array { /* . . . */ };
template <int hi, int wid> class Screen { /* . . . */ };
(a) const int hi = 40, wi = 80; Screen<hi, wi+32> sObj;
(b) const int arr_size = 1024; Array<string, arr_size> a1;
(c) unsigned int asize = 255; Array<int, asize> a2;
(e) const double db = 3.1415; Array<double, db> a3;

```

16.4.3 类模板中的友元声明

在类模板中可以出现三种友元声明，每一种都声明了与一个或多个实体的友元关系：

- (1) 普通非模板类或函数的友元声明，将友元关系授予明确指定的类或函数。
- (2) 类模板或函数模板的友元声明，授予对友元所有实例的访问权。
- (3) 只授予对类模板或函数模板的特定实例的访问权的友元声明。

1. 普通友元

非模板类或非模板函数可以是类模板的友元：

```

template <class Type> class Bar {
    // grants access to ordinary, nontemplate class and function
    friend class FooBar;
    friend void fcn();
    // ...
};

```

这个声明是说，FooBar 的成员和 fcn 函数可以访问 Bar 类的任意实例的 private 成员和 protected 成员。

2. 一般模板友元关系

友元可以是类模板或函数模板：

```

template <class Type> class Bar {
    // grants access to Foo1 or templ_fcn1 parameterized by any type
    template <class T> friend class Foo1;
    template <class T> friend void templ_fcn1(const T&);
    // ...
};

```

这些友元声明使用与类本身不同的类型形参，该类型形参指的是 Foo1 和 templ_fcn1 的类型形参。在这两种情况下，都将没有数目限制的和函数设为 Bar 的友元。Foo1 的友元声明是说，Foo1 的任意实例都可以访问 Bar 的任意实例的私有元素，类似地，templ_fcn1 的任意实例可以访问 Bar 的任意实例。

这个友元声明在 Bar 与其友元 Foo1 和 templ_fcn1 的每个实例之间建立了一对多的映射。对 Bar 的每个实例而言, Foo1 或 templ_fcn1 的所有实例都是友元。

3. 特定的模板友元关系

除了将一个模板的所有实例设为友元, 类也可以只授予对特定实例的访问权:

```
template <class T> class Foo2;
template <class T> void templ_fcn2(const T&);
template <class Type> class Bar {
    // grants access to a single specific instance parameterized by char*
    friend class Foo2<char*>;
    friend void templ_fcn2<char*>(char* const &);
    // ...
};
```

即使 Foo2 本身是类模板, 友元关系也只扩展到 Foo2 的形参类型为 char* 的特定实例。类似地, templ_fcn2 的友元声明是说, 只有形参类型为 char* 的函数实例是 Bar 类的友元。形参类型为 char* 的 Foo2 和 templ_fcn2 的特定实例可以访问 Bar 的每个实例。

下面形式的友元声明更为常见:

```
template <class T> class Foo3;
template <class T> void templ_fcn3(const T&);
template <class Type> class Bar {
    // each instantiation of Bar grants access to the
    // version of Foo3 or templ_fcn3 instantiated with the same type
    friend class Foo3<Type>;
    friend void templ_fcn3<Type>(const Type&);
    // ...
};
```

这些友元定义了 Bar 的特定实例与使用同一模板实参的 Foo3 或 templ_fcn3 的实例之间的友元关系。每个 Bar 实例有一个相关的 Foo3 和 templ_fcn3 友元:

```
Bar<int> bi;    // Foo3<int> and templ_fcn3<int> are friends
Bar<string> bs; // Foo3<string>, templ_fcn3<string> are friends
```

只有与给定 Bar 实例有相同模板实参的那些 Foo3 或 templ_fcn3 版本是友元。因此, Foo3<int> 可以访问 Bar<int> 的私有部分, 但不能访问 Bar<string> 或者任意其他 Bar 实例的私有部分。

4. 声明依赖性

当授予对给定模板的所有实例的访问权的时候, 在作用域中不需要存在该类模板或函数模板的声明。实质上, 编译器将友元声明也当作类或函数的声明对待。

想要限制对特定实例化的友元关系时, 必须在可以用于友元声明之前声明类或函数:

```
template <class T> class A;
template <class T> class B {
public:
    friend class A<T>;           // ok: A is known to be a template
    friend class C;              // ok: C must be an ordinary, nontemplate class
    template <class S> friend class D; // ok: D is a template
    friend class E<T>;           // error: E wasn't declared as a template
    friend class F<int>;          // error: F wasn't declared as a template
};
```

如果没有事先告诉编译器该友元是一个模板，则编译器将认为该友元是一个普通非模板类或非模板函数。

16.4.4 Queue 和 QueueItem 的友元声明

QueueItem 类不打算为一般程序所用：它的所有成员都是私有的。为了让 Queue 类使用 QueueItem 类，QueueItem 类必须将 Queue 类设为友元。

1. 将类模板设为友元

像我们已经看到的，将类模板设为友元的时候，类设计者必须决定友元关系应设置多广。在 QueueItem 类的例子中，需要决定 QueueItem 类应该将友元关系授予所有的 Queue 类实例，还是只授予特定实例。

将每个 Queue 类设为每个 QueueItem 类的友元太宽泛了，允许用 string 类型实例化的 Queue 类去访问用 double 类型实例化的 QueueItem 类的成员是没有意义的。Queue<string> 实例只应该用 string 实例化的 QueueItem 类的友元，即，对于实例化的 Queue 类的每种类型，我们想要 Queue 类和 QueueItem 类之间的一对一映射：

```
// declaration that Queue is a template needed for friend declaration in QueueItem
template <class Type> class Queue;
template <class Type> class QueueItem {
    friend class Queue<Type>;
    // ...
};
```

这个声明建立了想要的一对一映射，只将与 QueueItem 类用同样类型实例化的 Queue 类设为友元。

658

2. Queue 输出操作符

Queue 类接口中可能增加的一个有用操作，是输出 Queue 对象的内容的能力。提供输出操作符的重载实例，可以做到这一点。这个操作符将遍历 Queue 中的元素链表并输出每个元素的值，将在一对尖括号内输出元素。

因为希望能够输出任意类型 Queue 的内容，所以需要将输出操作符也设为模板：

```
template <class Type>
ostream& operator<<(ostream &os, const Queue<Type> &q)
{
    os << "< ";
    QueueItem<Type> *p;
    for (p = q.head; p; p = p->next)
        os << p->item << " ";
    os << ">";
    return os;
}
```

如果 int 类型的 Queue 包含值 3、5、8 和 13，这个 Queue 的输出显示如下：

```
< 3 5 8 13 >
```

如果 Queue 为空，for 循环不执行。结果是输出一对空的尖括号。

3. 将函数模板设为友元

输出操作符需要成为 Queue 类和 QueueItem 类的友元。它使用 Queue 类的 head 成员和 QueueItem 类的 next 和 item 成员。我们的类将友元关系授予用同样类型实例化的输出操作符的特定实例：

```
// function template declaration must precede friend declaration in QueueItem
template <class T>
std::ostream& operator<<(std::ostream&, const Queue<T>&);
template <class Type> class QueueItem {
    friend class Queue<Type>;
    // needs access to item and next
    friend std::ostream&
    operator<< <Type> (std::ostream&, const Queue<Type>&);
    // ...
};
template <class Type> class Queue {
    // needs access to head
    friend std::ostream&
    operator<< <Type> (std::ostream&, const Queue<Type>&);
};
```

659

每个友元声明授予对对应 operator<<实例的访问权，即输出 Queue<int>的输出操作符是 Queue<int>类（以及 QueueItem<int>类）的友元，它不是任意其他 Queue 类型的友元。

4. 类型依赖性与输出操作符

Queue 类的输出 operator<<依赖于 item 对象的 operator<<实际输出每个元素：

```
os << p->item << " ";
```

当使用 p->item 作为<<操作符的操作数的时候，使用的是为 item 所属的任意类型而定义的<<。

此代码是 Queue 和 Queue 保存的元素之间的类型依赖性的例子。实际上，绑定到 Queue 且使用 Queue 输出操作符的每种类型本身必须有输出操作符。没有语言机制指定或强制 Queue 自身定义中的依赖性。为没有定义输出操作符的类创建 Queue 对象是合法的，但输出保存这种类型的 Queue 对象会发生编译时（或链接时）错误。

习题

习题 16.38 编写 Screen 类模板，使用非类型形参定义 Screen 的高度和宽度。

习题 16.39 为 Screen 模板类实现输入和输出操作符。

习题 16.40 要使输入和输出操作符能够工作，Screen 类需要友元吗？如果需要，要哪些友元解释为什么需要每个友元声明。

习题 16.41 Queue 类中的 operator<<的友元声明是：

```
friend std::ostream&
operator<< <Type> (std::ostream&, const Queue<Type>&);
```

将 Queue 形参写为 const Queue&而不是 const Queue<Type>&，会有什么结果？

习题 16.42 编写一个输入操作符，读一个 istream 对象并将读到的值放入一个 Queue 对象中。

16.4.5 成员模板

任意类（模板或非模板）可以拥有本身为类模板或函数模板的成员，这种成员称为成员模板（member template），成员模板不能为虚。

660 成员模板的一个例子是标准容器的 assign 成员（9.3.8 节），接受两个迭代器的 assign 版本使用模板形参表示其迭代器形参的类型。另一个成员模板例子是接受两个迭代器的容器构造函数（9.1.1 节）。该构造函数和 assign 成员使我们能够从不同但兼容的元素类型序列和/或不同容器类型建立容器。实现了自己的 Queue 类之后，我们现在能够更好地理解这些标准容器成员的设计了。

考虑 Queue 类的复制构造函数：它接受一个形参，是 Queue<Type>的引用。想要通过从 vector 对象中复制元素而创建 Queue 对象，是办不到的，因为没有从 vector 到 Queue 的转换。类似地，想要从 Queue<short>复制元素到 Queue<int>，也办不到。同样的逻辑应用于赋值操作符，它也接受一个 Queue<Type>&类型的形参。

问题在于，复制构造函数和赋值操作符固定了容器和元素的类型。我们希望定义一个构造函数和一个 assign 成员，使容器类型和元素类型都能变化。需要形参类型变化的时候，就需要定义函数模板。在这个例子中，我们将定义构造函数和 assign 成员接受一对在其他序列指明范围的迭代器，这些函数将有一个表示迭代器类型的模板类型形参。



标准 queue 类没有定义这些成员：不支持从其他容器建立 queue 对象或给 queue 对象赋值。我们在这里定义这些成员只是为了举例说明。

1. 定义成员模板

模板成员声明看起来像任意模板的声明一样：

```
template <class Type> class Queue {
public:
    // construct a Queue from a pair of iterators on some sequence
    template <class It>
    Queue(It beg, It end):
        head(0), tail(0) { copy_elems(beg, end); }
    // replace current Queue by contents delimited by a pair of iterators
    template <class Iter> void assign(Iter, Iter);
    // rest of Queue class as before
private:
    // version of copy to be used by assign to copy elements from iterator range
    template <class Iter> void copy_elems(Iter, Iter);
};
```

成员声明的开头是自己的模板形参表。构造函数和 assign 成员各有一个模板类型形参，这些函数使用该类型形参作为其函数形参的类型，它们的函数形参是指明要复制元素范围的迭代器。

2. 在类外部定义成员模板

像非模板成员一样，成员模板可以定义在包含它的类或类模板定义的内部或外部。我们已经在类定义体内部定义了构造函数，它的工作是从迭代器实参形成的迭代器范围复制元素，实际复

制工作是通过调用 `copy_elems` 的迭代器版本完成的。

当在类模板作用域外部定义成员模板的时候，必须包含两个模板形参表：

```
template <class T> template <class Iter>
void Queue<T>::assign(Iter beg, Iter end)
{
    destroy();           // remove existing elements in this Queue
    copy_elems(beg, end); // copy elements from the input range
}
```

当成员模板是类模板的成员时，它的定义必须包含类模板形参以及自己的模板形参。首先是类模板形参表，后面接着成员自己的模板形参表。`assign` 函数定义的开头为

```
template <class T> template <class Iter>
```

第一个模板形参表 `template<class T>` 是类模板的，第二个模板形参表 `template<class Iter>` 是成员模板的。

`assign` 函数的行为非常简单：它首先调用 `destroy` 函数，`destroy` 函数释放这个 `Queue` 的现存成员，然后 `assign` 成员调用名为 `copy_elems` 的新实用函数，完成从输入范围复制元素的工作。`copy_elems` 函数也是一个成员模板：

```
template <class Type> template <class It>
void Queue<Type>::copy_elems(It beg, It end)
{
    while (beg != end) {
        push(*beg);
        ++beg;
    }
}
```

`copy_elems` 的迭代器版本遍历由一对迭代器指定的输入范围，它对范围内的每个元素调用 `push` 函数，实际上由 `push` 函数将元素加入 `Queue`。



因为 `assign` 函数删除现存容器中的成员，所以传给 `assign` 函数的迭代器有必要引用不同容器中的元素。标准容器的 `assign` 成员和迭代器构造函数有相同的限制。

662

3. 成员模板遵循常规访问控制

成员模板遵循与任意其他类成员一样的访问规则。如果成员模板为私有的，则只有该类的成员函数和友元可以使用该成员模板。因为函数成员模板 `assign` 是公有的，所以整个程序都可以使用它；`copy_elems` 是私有的，所以只有 `Queue` 的友元和成员可以访问它。

4. 成员模板和实例化

与其他成员一样，成员模板只有在程序中使用时才实例化。类模板的成员模板的实例化比类模板的普通成员函数的实例化要复杂一点。成员模板有两种模板形参：由类定义的和由成员模板本身定义的。类模板形参由调用函数的对象的类型确定，成员定义的模板形参的行为与普通函数模板一样。这些形参都通过常规模板实参推断（16.2.1 节）而确定。

要理解实例化的原理，我们来看看使用这些成员从 `short` 数组或 `vector<int>` 复制和赋值

元素:

```
short a[4] = { 0, 3, 6, 9 };
// instantiates Queue<int>::Queue(short *, short *)
Queue<int> qi(a, a + 4); // copies elements from a into qi
vector<int> vi(a, a + 4);
// instantiates Queue<int>::assign(vector<int>::iterator,
//                                vector<int>::iterator)
qi.assign(vi.begin(), vi.end());
```

因为所构造的是 `Queue<int>` 类型的对象, 我们知道编译器将为 `Queue<int>` 实例化基于迭代器的构造函数。该构造函数本身模板形参的类型由编译器根据 `a` 和 `a+4` 的类型推断, 而该类型为 `short` 指针。因此, `qi` 的定义将实例化

```
void Queue<int>::Queue(short *, short *);
```

这个构造函数的效果是, 从名为 `a` 的数组中复制 `short` 类型的元素到 `qi`。

对 `assign` 的调用将实例化 `qi` 的成员。`qi` 具有 `Queue<int>` 类型, 因此, 这个调用将实例化名为 `assign` 的 `Queue<int>` 成员。该函数本身是函数模板, 像对任意其他函数模板一样, 编译器从传给调用的实参推断 `assign` 的模板实参, 推断得到的类型是 `vector<int>::iterator`, 即, 这个调用将实例化

```
void Queue<int>::assign(vector<int>::iterator,
                        vector<int>::iterator);
```

663

16.4.6 完整的 Queue 类

为了完整起见, 在这里给出 `Queue` 类的最终定义:

```
// declaration that Queue is a template needed for friend declaration in QueueItem
template <class Type> class Queue;
// function template declaration must precede friend declaration in QueueItem
template <class T>
std::ostream& operator<<(std::ostream&, const Queue<T>&);
template <class Type> class QueueItem {
    friend class Queue<Type>;
    // needs access to item and next
    friend std::ostream& // defined on page 659
    operator<< <Type> (std::ostream&, const Queue<Type>&);
    // private class: no public section
    QueueItem(const Type &t): item(t), next(0) { }
    Type item; // value stored in this element
    QueueItem *next; // pointer to next element in the Queue
};
template <class Type> class Queue {
    // needs access to head
    friend std::ostream& // defined on page 659
    operator<< <Type> (std::ostream&, const Queue<Type>&);
public:
    // empty Queue
    Queue(): head(0), tail(0) { }
    // construct a Queue from a pair of iterators on some sequence
    template <class It>
    Queue(It beg, It end):
```

```

        head(0), tail(0) { copy_elems(beg, end); }
// copy control to manage pointers to QueueItems in the Queue
Queue(const Queue &Q): head(0), tail(0)
                        { copy_elems(Q); }
Queue& operator=(const Queue&); // left as exercise for the reader
~Queue() { destroy(); }
// replace current Queue by contents delimited by a pair of iterators
template <class Iter> void assign(Iter, Iter);
// return element from head of Queue
// unchecked operation: front on an empty Queue is undefined
Type& front()           { return head->item; }
const Type &front() const { return head->item; }
void push(const Type &); // defined on page 652
void pop();              // defined on page 651
bool empty() const {    // true if no elements in the Queue
    return head == 0;
}
private:
    QueueItem<Type> *head;           // pointer to first element in Queue
    QueueItem<Type> *tail;          // pointer to last element in Queue
// utility functions used by copy constructor, assignment, and destructor
void destroy();                    // defined on page 651
void copy_elems(const Queue&);     // defined on page 652
// version of copy to be used by assign to copy elements from iterator range
// defined on page 662
template <class Iter> void copy_elems(Iter, Iter);
};
// Inclusion Compilation Model: include member function definitions as well
#include "Queue.cc"

```

664

未在类本身中定义的成员可在本章前面几节中找到，跟在这些成员后面的注释指出了可以在哪一节找到它们。

习题

习题 16.43 为你的 List 类增加 assign 成员和一个参数为一对迭代器的构造函数。

习题 16.44 为了举例说明怎样实现类模板，我们实现了自己的 Queue 类。可以简化实现的一种方式可能是将 Queue 定义在一个现存的标准库容器类型之上，用这种方法，可以避免必须管理 Queue 元素的分配和回收。用 std::List 保存实际 Queue 元素，重新实现 Queue 类。

16.4.7 类模板的 static 成员

类模板可以像任意其他类一样声明 static 成员（12.6 节）。以下代码：

```

template <class T> class Foo {
public:
    static std::size_t count() { return ctr; }
    // other interface members
private:
    static std::size_t ctr;
    // other implementation members
};

```

定义了名为 `Foo` 的类模板，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。

`Foo` 类的每个实例化有自己的 `static` 成员：

```
// Each object shares the same Foo<int>::ctr and Foo<int>::count members
Foo<int> fi, fi2, fi3;
// has static members Foo<string>::ctr and Foo<string>::count
Foo<string> fs;
```

665

每个实例化表示截然不同的类型，所以给定实例化的所有对象都共享一个 `static` 成员。因此，`Foo<int>` 类型的任意对象共享同一 `static` 成员 `ctr`，`Foo<string>` 类型的对象共享另一个不同的 `ctr` 成员。

1. 使用类模板的 `static` 成员

通常，可以通过类类型的对象访问类模板的 `static` 成员，或者通过使用作用域操作符直接访问成员。当然，当试图通过类使用 `static` 成员的时候，必须引用实际的实例化：

```
Foo<int> fi, fi2; // instantiates Foo<int> class
size_t ct = Foo<int>::count(); // instantiates Foo<int>::count
ct = fi.count(); // ok: uses Foo<int>::count
ct = fi2.count(); // ok: uses Foo<int>::count
ct = Foo::count(); // error: which template instantiation?
```

与任意其他成员函数一样，`static` 成员函数只有在程序中使用时才进行实例化。

2. 定义 `static` 成员

像使用任意其他 `static` 数据成员一样，必须在类外部出现数据成员的定义。在类模板含有 `static` 成员的情况下，成员定义必须指出它是类模板的成员：

```
template <class T>
size_t Foo<T>::ctr = 0; // define and initialize ctr
```

`static` 数据成员像定义在类外部的任意其他类成员一样定义，它用关键字 `template` 开头，后面接着类模板形参表和类名。在这个例子中，`static` 数据成员的名字以 `Foo<T>` 为前缀，表示成员属于类模板 `Foo`。

16.5 一个泛型句柄类



这个例子体现了 C++ 相当复杂的语言应用，理解它需要很好地理解继承和模板。在熟悉了这些特性之后再研究这个例子也许会有帮助。另一方面，这个例子还能很好地测试你对这些特性的理解程度。

在第 15 章定义了两个句柄类：`Sales_item` 类（15.8 节）和 `Query` 类（15.9 节）。这两个类管理继承层次中对象的指针，句柄的用户不必管理指向这些对象的指针，用户代码可以使用句柄类来编写。句柄能够动态分配和释放相关继承类的对象，并且将所有“实际”工作转发给继承层次中的底层类。

666

这两个句柄类似但并不相同：类似之处在于都定义了使用计数式的复制控制，管理指向继承

层次中某类型对象的指针；不同之处在于它们提供给继承层次用户的接口。

两个类的使用计数的实现是相同的。这类问题非常适合于泛型编程：可以定义类模板管理指针和进行使用计数。原本不相关的 `Sales_item` 类型和 `Query` 类型，可通过使用该模板进行公共的使用计数工作而得以简化。至于是公开还是隐藏下层的继承层次，句柄可以保持不同。

本节将实现一个泛型句柄类（generic handle class），提供管理使用计数和基础对象的操作。然后，我们重新编写 `Sales_item` 类，展示它怎样使用泛型句柄而不是定义自己的使用计数操作。

16.5.1 定义句柄类

`Handle` 类的行为类似于指针：复制 `Handle` 对象将不会复制基础对象，复制之后，两个 `Handle` 对象将引用同一基础对象。要创建 `Handle` 对象，用户需要传递属于由 `Handle` 管理的类型（或从该类型派生的类型）的动态分配对象的地址，从此刻起，`Handle` 将“拥有”这个对象。而且，一旦不再有任意 `Handle` 对象与该对象关联，`Handle` 类将负责删除该对象。

对于这一设计，我们的泛型 `Handle` 类的实现如下：

```
/* generic handle class: Provides pointerlike behavior. Although access through
 * an unbound Handle is checked and throws a runtime_error exception.
 * The object to which the Handle points is deleted when the last Handle goes away.
 * Users should allocate new objects of type T and bind them to a Handle.
 * Once an object is bound to a Handle, the user must not delete that object.
 */
template <class T> class Handle {
public:
    // unbound handle
    Handle(T *p = 0): ptr(p), use(new size_t(1)) { }
    // overloaded operators to support pointer behavior
    T& operator*();
    T* operator->();
    const T& operator*() const;
    const T* operator->() const;
    // copy control: normal pointer behavior, but last Handle deletes the object
    Handle(const Handle& h): ptr(h.ptr), use(h.use)
                                { ++*use; }

    Handle& operator=(const Handle&);
    ~Handle() { rem_ref(); }
private:
    T* ptr;           // shared object
    size_t *use;      // count of how many Handles spoint to *ptr
    void rem_ref()
        { if (--*use == 0) { delete ptr; delete use; } }
};
```

这个类看来与其他句柄类似，赋值操作符也类似。

```
template <class T>
inline Handle<T>& Handle<T>::operator=(const Handle &rhs)
{
    ++*rhs.use;        // protect against self-assignment
    rem_ref();         // decrement use count and delete pointers if needed
    ptr = rhs.ptr;
    use = rhs.use;
    return *this;
}
```

Handle 类将定义的其他成员是解引用操作符和成员访问操作符，这些操作符将用于访问基础对象。让这些操作检查 Handle 是否确实绑定到对象，可以提供一种安全措施。如果 Handle 没有绑定到对象，则试图访问对象将抛出一个异常。

这些操作的非 const 版本看来如下所示：

```
template <class T> inline T& Handle<T>::operator*()
{
    if (ptr) return *ptr;
    throw std::runtime_error
        ("dereference of unbound Handle");
}
template <class T> inline T* Handle<T>::operator->()
{
    if (ptr) return ptr;
    throw std::runtime_error
        ("access through unbound Handle");
}
```

const 版本与此类似，留作习题。

习题

习题 16.45 实现一个 Handle 类的自己的版本。

习题 16.46 解释复制 Handle 类型的对象时会发生什么。

习题 16.47 Handle 类对用来实例化实际 Handle 类的类型有限制吗？如果有，限制有哪些？

习题 16.48 解释如果用户将 Handle 对象与局部对象关联会发生什么。解释如果用户删除 Handle 对象所关联的对象会发生什么。

16.5.2 使用句柄

我们希望 Handle 类能够用于其他类的内部实现中。但是，为了帮助理解 Handle 类怎样工作，将首先介绍一个较简单的例子。这个例子通过分配一个 int 对象，并将一个 Handle 对象绑定到新分配的 int 对象而说明 Handle 的行为：

```
{ // new scope
    // user allocates but must not delete the object to which the Handle is attached
    Handle<int> hp(new int(42));
    { // new scope
        Handle<int> hp2 = hp; // copies pointer; use count incremented
        cout << *hp << " " << *hp2 << endl; // prints 42 42
        *hp2 = 10; // changes value of shared underlying int
    } // hp2 goes out of scope; use count is decremented
    cout << *hp << endl; // prints 10
} // hp goes out of scope; its destructor deletes the int
```

即使是 Handle 的用户分配了 int 对象，Handle 析构函数也将删除它。在外层代码块末尾最后一个 Handle 对象超出作用域时，删除该 int 对象。为了访问基础对象，应用了 Handle 的 * 操作符，该操作符返回对基础 int 对象的引用。

使用 **Handle** 对象对指针进行使用计数

作为在类实现中使用 **Handle** 的例子，可以重新实现 `Sales_item` 类（15.8.1 节），该类的这个版本定义相同的接口，但可以通过用 `Handle<Item_base>` 对象代替 `Item_base` 指针而删去复制控制成员：

```
class Sales_item {
public:
    // default constructor: unbound handle
    Sales_item(): h() { }
    // copy item and attach handle to the copy
    Sales_item(const Item_base &item): h(item.clone()) { }
    // no copy control members: synthesized versions work
    // member access operators: forward their work to the Handle class
    const Item_base& operator*() const { return *h; }
    const Item_base* operator->() const
        { return h.operator->(); }
private:
    Handle<Item_base> h; // use-counted handle
};
```

虽然 `Sales_item` 类的接口没变，它的实现与原来的相当不同：

- 两个类都定义了默认构造函数和以 `Item_base` 对象为参数的 `const` 引用的构造函数。
- 两个类都将重载的 `*` 和 `->` 操作符定义为 `const` 成员。

基于 **Handle** 的 `Sales_item` 版本有一个数据成员，该数据成员是关联传给构造函数的 `Item_base` 对象的副本上的 **Handle** 对象。因为 `Sales_item` 的这个版本没有指针成员，所以不需要复制控制成员，`Sales_item` 的这个版本可以安全地使用合成的复制控制成员。管理使用计数和相关 `Item_base` 对象的工作在 **Handle** 内部完成。

因为接口没变，所以不需要改变使用 `Sales_item` 类的代码。例如，15.8.3 节中编写的程序可以无须改变而使用：

```
double Basket::total() const
{
    double sum = 0.0; // holds the running total
    /* find each set of items with the same isbn and calculate
     * the net price for that quantity of items
     * iter refers to first copy of each book in the set
     * upper_bound refers to next element with a different isbn
     */
    for (const_iter iter = items.begin();
         iter != items.end();
         iter = items.upper_bound(*iter))
    {
        // we know there's at least one element with this key in the Basket
        // virtual call to net_price applies appropriate discounts, if any
        sum += (*iter)->net_price(items.count(*iter));
    }
    return sum;
}
```

调用 `net_price` 函数的语句值得仔细分析一下：

```
sum += (*iter)->net_price(items.count(*iter));
```

这个语句使用->操作符获取并运行 net_price 函数，重要的是理解这个操作符怎样工作：

- (*iter) 返回 h，h 是使用计数式句柄的成员。
- 因此，(*iter)-> 使用句柄类的重载箭头操作符。
- 编译器计算 h.operator->()，获得该 Handle 对象保存的 Item_base 指针。
- 编译器对该 Item_base 指针解引用，并调用指针所指对象的 net_price 成员。

670

习题

习题 16.49 实现本节提出的 Sales_item 句柄的版本，该版本使用泛型 Handle 类管理 Item_base 指针。

习题 16.50 重新运行函数计算销售总额。列出让你的代码工作必须进行的所有修改。

习题 16.51 重新编写 15.9.4 节的 Query 类以使用泛型 Handle 类。注意你需要将 Handle 类设为 Query_base 类的友元，以使它能够访问 Query_base 构造函数。列出并解释让程序工作要做的其他所有修改。

16.6 模板特化



本章其余部分将介绍一个比较高级的主题，在第一次阅读时可以跳过它。

我们并不总是能够写出对所有可能被实例化的类型都最合适的模板。某些情况下，通用模板定义对于某个类型可能是完全错误的，通用模板定义也许不能编译或者做错误的事情；另外一些情况下，可以利用关于类型的一些特殊知识，编写比从模板实例化来的函数更有效率的函数。

compare 函数和 Queue 类都是这一问题的好例子：与 C 风格字符串一起使用时，它们都不能正确工作。让我们再来看看 compare 函数模板：

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

如果用两个 const char* 实参调用这个模板定义，函数将比较指针值。它将告诉我们这两个指针在内存中的相对位置，但没有说明与指针所指数组的内容有关的任何事情。

为了能够将 compare 函数用于字符串，必须提供一个知道怎样比较 C 风格字符串的特殊定义。这些版本是特化的，这一事实对模板的用户透明。对用户而言，调用特化函数或使用特化类，与使用从通用模板实例化的版本无法区别。

671

16.6.1 函数模板的特化

模板特化 (template specialization) 是这样的一个定义, 该定义中一个或多个模板形参的实际类型或实际值是指定的。特化的形式如下:

- 关键字 `template` 后面接一对空的尖括号 (`<>`);
- 再接模板名和一对尖括号, 尖括号中指定这个特化定义的模板形参;
- 函数形参表;
- 函数体。

下面的程序定义了当模板形参类型绑定到 `const char*` 时, `compare` 函数的特化:

```
// special version of compare to handle C-style character strings
template <>
int compare<const char*>(const char* const &v1,
                        const char* const &v2)
{
    return strcmp(v1, v2);
}
```

特化的声明必须与对应的模板相匹配。在这个例子中, 模板有一个类型形参和两个函数形参, 函数形参是类型形参的 `const` 引用, 在这里, 将类型形参固定为 `const char*`, 因此, 函数形参是 `const char*` 的 `const` 引用。

现在, 当调用 `compare` 函数的时候, 传给它两个字符指针, 编译器将调用特化版本。编译器将为任意其他实参类型 (包括普通 `char*`) 调用泛型版本:

```
const char *cp1 = "world", *cp2 = "hi";
int i1, i2;
compare(cp1, cp2); // calls the specialization
compare(i1, i2);  // calls the generic version instantiated with int
```

1. 声明模板特化

与任意函数一样, 函数模板特化可以声明而无须定义。模板特化声明看起来与定义很像, 但省略了函数体:

```
// declaration of function template explicit specialization
template<>
int compare<const char*>(const char* const&,
                        const char* const&);
```

672

这个声明由一个后接返回类型的空模板形参表 (`template<>`), 后接一对尖括号中指定的显式模板实参的函数名 (可选), 以及函数形参表构成。模板特化必须总是包含空模板形参说明符, 即 `template<>`, 而且, 还必须包含函数形参表。如果可以从函数形参表推断模板实参, 则不必显式指定模板实参:

```
// error: invalid specialization declarations
// missing template<>
int compare<const char*>(const char* const&,
                        const char* const&);

// error: function parameter list missing
template<> int compare<const char*>;
```

```
// ok: explicit template argument const char* deduced from parameter types
template<> int compare(const char* const&,
                     const char* const&);
```

2. 函数重载与模板特化

在特化中省略空的模板形参表 `template<>` 会有令人惊讶的结果。如果缺少该特化语法，则结果是声明该函数的重载非模板版本：

```
// generic template definition
template <class T>
int compare(const T& t1, const T& t2) { /* ... */ }

// OK: ordinary function declaration
int compare(const char* const&, const char* const&);
```

`compare` 的定义没有定义模板特化，相反，它声明了一个普通函数，该函数含有返回类型和可与模板实例化相匹配的形参表。

下一节将更详细地介绍重载和模板的交互作用。现在，重要的是知道，当定义非模板函数的时候，对实参应用常规转换；当特化模板的时候，对实参类型不应用转换。在模板特化版本的调用中，实参类型必须与特化版本函数的形参类型完全匹配，如果不完全匹配，编译器将为实参从模板定义实例化一个实例。

3. 不是总能检测到重复定义

如果程序由多个文件构成，模板特化的声明必须在使用该特化的每个文件中出现。不能在一些文件中从泛型模板定义实例化一个函数模板，而在其他文件中为同一模板实参集合特化该函数模板。

673



最佳
实践

与其他函数声明一样，应在一个头文件中包含模板特化的声明，然后使用该特化的每个源文件包含该头文件。

4. 普通作用域规则适用于特化

在能够声明或定义特化之前，它所特化的模板的声明必须在作用域中。类似地，在调用模板的这个版本之前，特化的声明必须在作用域中：

```
// define the general compare template
template <class T>
int compare(const T& t1, const T& t2) { /* ... */ }

int main() {
    // uses the generic template definition
    int i = compare("hello", "world");
    // ...
}

// invalid program: explicit specialization after call
template<>
int compare<const char*>(const char* const& s1,
                        const char* const& s2)
{ /* ... */ }
```

这个程序有错误，因为在声明特化之前，进行了可以与特化相匹配的一个调用。当编译器看到一个函数调用时，它必须知道这个版本需要特化，否则，编译器将可能从模板定义实例化该函数。



对具有同一模板实参集的同一模板，程序不能既有显式特化又有实例化。

特化出现在对该模板实例的调用之后是错误的。

习题

习题 16.52 定义函数模板 count 计算一个 vector 中某些值的出现次数。

习题 16.53 编写一个程序调用上题中定义的 count 函数，首先传给该函数一个 double 型 vector，然后传递一个 int 型 vector，最后传递一个 char 型 vector。

习题 16.54 引入 count 函数的一个特化模板实例以处理 string 对象。重新运行你所编写的调用函数模板实例化的程序。

674

16.6.2 类模板的特化

当用于 C 风格字符串时，Queue 类具有与 compare 函数相似的问题。在这种情况下，问题出在 push 函数中，该函数复制给定值以创建 Queue 中的新元素。默认情况下，复制 C 风格字符串只会复制指针，不会复制字符。这种情况下复制指针将出现共享指针在其他环境中会出现的所有问题，最严重的是，如果指针指向动态内存，用户就有可能删除指针所指的数组。

1. 定义类特化

为 C 风格字符串的 Queue 提供正确行为的一种途径，是为 const char* 定义整个类的特化版本：

```
/* definition of specialization for const char*
 * this class forwards its work to Queue<string>;
 * the push function translates the const char* parameter to a string
 * the front functions return a string rather than a const char*
 */
template<> class Queue<const char*> {
public:
    // no copy control: Synthesized versions work for this class
    // similarly, no need for explicit default constructor either
    void push(const char*);
    void pop() {real_queue.pop();}
    bool empty() const {return real_queue.empty();}
    // Note: return type does not match template parameter type
    std::string front() {return real_queue.front();}
    const std::string &front() const
        {return real_queue.front();}
private:
    Queue<std::string> real_queue; // forward calls to real_queue
};
```

这个实现给了 Queue 一个数据元素：string 对象的 Queue。各个成员将它们的工作委派给

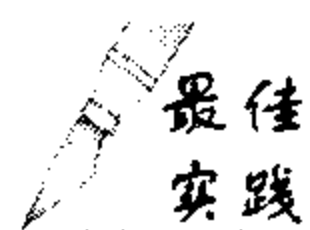
这个成员。例如，通过调用 `real_queue` 的 `pop` 实现 `pop` 成员。

`Queue` 类的这个版本没有定义复制控制成员，它唯一的数据成员为类类型，该类类型在被复制、被赋值或被撤销时完成正确的工作。可以使用合成的复制控制成员。

这个 `Queue` 类实现了与 `Queue` 的模板版本大部分相同但不完全相同的接口，区别在于 `front` 成员返回的是 `string` 而不是 `char*`，这样做是为了避免必须管理字符数组——如果想要返回指针，就需要字符数组。

值得注意的是，特化可以定义与模板本身完全不同的成员。如果一个特化无法从模板定义某个成员，该特化类型的对象就不能使用该成员。类模板成员的定义不会用于创建显式特化成员的定义。

675



最佳
实践

类模板特化应该与它所特化的模板定义相同的接口，否则当用户试图使用未定义的成员时会感到奇怪。

2. 类特化定义



注解

在类特化外部定义成员时，成员之前不能加 `template<>` 标记。

我们的类只在类的外部定义了一个成员：

```
void Queue<const char*>::push(const char* val)
{
    return real_queue.push(val);
}
```

虽然这个函数几乎没有做什么工作，但它隐式复制了 `val` 指向的字符数组。复制是在对 `real_queue.push` 的调用中进行的，该调用从 `const char*` 实参创建了一个新的 `string` 对象。`const char*` 实参使用了以 `const char*` 为参数的 `string` 构造函数，`string` 构造函数将 `val` 所指的数组中的字符复制到未命名的 `string` 对象，该对象将被存储在 `push` 到 `real_queue` 的元素中。

习题

习题 16.55 `Queue` 针对 `const char*` 的特化版本中的注释指出，不必定义默认构造函数或复制控制成员，解释为什么对于 `Queue` 的这个版本合成成员就足够了。

习题 16.56 我们已经解释过未针对 `const char*` 特化的 `Queue` 的泛型行为，使用泛型 `Queue` 模板解释下面代码中会发生什么：

```
Queue<const char*> q1;
q1.push("hi"); q1.push("bye"); q1.push("world");
Queue<const char*> q2(q1); // q2 is a copy of q1
```

```
Queue<const char*> q3; // empty Queue
q1 = q3;
```

具体而言，就是说明在 `q2` 的初始化和 `q3` 的赋值之后，`q1` 和 `q2` 是什么值。

习题 16.57 我们的 `Queue` 特化版本从 `front` 函数返回 `string` 对象而不是 `const char*`，你认为为什么这样做？你能够怎样实现 `Queue` 以返回 `const char*`？讨论每种方法的优缺点。

676

16.6.3 特化成员而不特化类

如果更深入一点分析我们的类，就能够看到代码可以简化：除了特化整个模板之外，还可以只特化 push 和 pop 成员。我们将特化 push 成员以复制字符数组，并且特化 pop 成员以释放该副本使用的内存：

```
template <>
void Queue<const char*>::push(const char *const &val)
{
    // allocate a new character array and copy characters from val
    char* new_item = new char[strlen(val) + 1];
    strncpy(new_item, val, strlen(val) + 1);
    // store pointer to newly allocated and initialized element
    QueueItem<const char*> *pt =
        new QueueItem<const char*>(new_item);
    // put item onto existing queue
    if (empty())
        head = tail = pt; // queue has only one element
    else {
        tail->next = pt; // add new element to end of queue
        tail = pt;
    }
}
template <>
void Queue<const char*>::pop()
{
    // remember head so we can delete it
    QueueItem<const char*> *p = head;
    delete head->item; // delete the array allocated in push
    head = head->next; // head now points to next element
    delete p; // delete old head element
}
```

现在，类类型 Queue<const char*> 将从通用类模板定义实例化而来，而 push 和 pop 函数例外。调用 Queue<const char*> 对象的 push 或 pop 函数时，将调用特化版本；调用任意其他成员时，将从类模板为 const char* 实例化一个通用版本。

特化声明

成员特化的声明与任何其他函数模板特化一样，必须以空的模板形参表开头：

```
// push and pop specialized for const char*
template <>
void Queue<const char*>::push(const char* const &);
template <> void Queue<const char*>::pop();
```

这些声明应放在 Queue 类的头文件中。

677

习题

习题 16.58 前一小节中给出的 Queue 类的特化，以及本小节中 push 和 pop 函数的特化，只适用于 const char* 类型的 Queue。为普通 char* 实现特化 Queue 的这两种不同方式。

习题 16.59 如果走只特化 push 函数的路线，对于 C 风格字符串的 Queue，front 返回什么值？

习题 16.60 讨论这两个设计的优缺点：为 `const char*` 定义该类的特化版本和只特化 `push` 和 `pop` 函数。具体而言，比较 `front` 的行为的异同以及用户代码中的错误破坏 `Queue` 元素的可能性。

16.6.4 类模板的部分特化

如果类模板有一个以上的模板形参，我们也许想要特化某些模板形参而非全部。使用类模板的部分特化可以做到这一点：

```
template <class T1, class T2>
class some_template {
    // ...
};
// partial specialization: fixes T2 as int and allows T1 to vary
template <class T1>
class some_template<T1, int> {
    // ...
};
```

类模板的部分特化（*partial specialization*）本身也是模板。部分特化的定义看来像模板定义，这种定义以关键字 `template` 开头，接着是由尖括号（`<>`）括住的模板形参表。部分特化的模板形参表是对应的类模板定义形参表的子集。`some_template` 的部分特化只有一个名为 `T1` 的模板类型形参，第二个模板形参 `T2` 的实参已知为 `int`。部分特化的模板形参表只列出未知模板实参的那些形参。

使用类模板的部分特化

部分特化与对应类模板有相同名字，即这里的 `some_template`。类模板的名字后面必须接着模板实参列表，前面例子中，模板实参列表是 `<T1, int>`。因为第一个模板形参的实参值未知，实参列表使用模板形参名 `T1` 作为占位符，另一个实参是类型 `int`，为 `int` 而部分特化模板。

像任何其他类模板一样，部分特化是在程序中使用隐式实例化：

```
some_template<int, string> foo;    // uses template
some_template<string, int> bar;    // uses partial specialization
```

注意第二个变量的类型，形参为 `string` 和 `int` 的 `some_template`，既可以从普通类模板定义实例化，也可以从部分特化实例化。为什么选择部分特化来实例化该模板呢？当声明了部分特化的时候，编译器将为实例化选择最特化的模板定义，当没有部分特化可以使用的时候，就使用通用模板定义。`foo` 的实例化类型与提供的部分特化不匹配，因此，`foo` 的类型必然从通用类模板实例化，将 `int` 绑定到 `T1` 并将 `string` 绑定到 `T2`。部分特化只用于实例化第二个类型为 `int` 的 `some_template` 类型。

部分特化的定义与通用模板的定义完全不会冲突。部分特化可以具有与通用类模板完全不同的成员集合。类模板成员的通用定义永远不会用来实例化类模板部分特化的成员。

16.7 重载与函数模板

函数模板可以重载：可以定义有相同名字但形参数目或类型不同的多个函数模板，也可以定

义与函数模板有相同名字的普通非模板函数。

当然, 声明一组重载函数模板不保证可以成功调用它们, 重载的函数模板可能会导致二义性。

1. 函数匹配与函数模板

如果重载函数中既有普通函数又有函数模板, 确定函数调用的步骤如下:

(1) 为这个函数名建立候选函数集合, 包括:

(a) 与被调用函数名字相同的任意普通函数。

(b) 任意函数模板实例化, 在其中, 模板实参推断发现了与调用中所用函数实参相匹配的模板实参。

(2) 确定哪些普通函数是可行的 (7.8.2 节) (如果有可行函数的话)。候选集合中的每个模板实例都是可行的, 因为模板实参推断保证函数可以被调用。

679

(3) 如果需要转换来进行调用, 根据转换的种类排列可行函数, 记住, 调用模板函数的实例所允许的转换是有限的。

(a) 如果只有一个函数可选, 就调用这个函数。

(b) 如果调用有二义性, 从可行函数集合中去掉所有函数模板实例。

(4) 重新排列去掉函数模板实例的可行函数。

- 如果只有一个函数可选, 就调用这个函数。

- 否则, 调用有二义性。

2. 函数模板匹配的例子

考虑下面普通函数和函数模板的重载集合:

```
// compares two objects
template <typename T> int compare(const T&, const T&);
// compares elements in two sequences
template <class U, class V> int compare(U, U, V);
// plain functions to handle C-style character strings
int compare(const char*, const char*);
```

重载集合包含三个函数: 第一个模板处理简单值, 第二个模板比较两个序列的元素, 第三个是处理 C 风格字符串的普通函数。

3. 确定重载函数模板的调用

可以在不同类型上调用这些函数:

```
// calls compare(const T&, const T&) with T bound to int
compare(1, 0);
// calls compare(U, U, V), with U and V bound to vector<int>::iterator
vector<int> ivec1(10), ivec2(20);
compare(ivec1.begin(), ivec1.end(), ivec2.begin());
int ia1[] = {0,1,2,3,4,5,6,7,8,9};
// calls compare(U, U, V) with U bound to int*
// and V bound to vector<int>::iterator
compare(ia1, ia1 + 10, ivec1.begin());
// calls the ordinary function taking const char* parameters
const char const_arr1[] = "world", const_arr2[] = "hi";
compare(const_arr1, const_arr2);
// calls the ordinary function taking const char* parameters
```

```
char ch_arr1[] = "world", ch_arr2[] = "hi";
compare(ch_arr1, ch_arr2);
```

680 下面依次介绍每个调用。

compare(1, 0): 两个形参都是 `int` 类型。候选函数是第一个模板将 `T` 绑定到 `int` 的实例化，以及名为 `compare` 的普通函数。但该普通函数不可行——不能将 `int` 对象传给期待 `char*` 对象的形参。用 `int` 实例化的函数与该调用完全匹配，所以选择它。

```
compare(ivec1.begin(), ivec1.end(), ivec2.begin() )
```

compare(ia1, ia1+10, ivec1.begin()): 这两个调用中，唯一可行的函数是有三个形参的模板的实例化。带两个参数的模板和普通非模板¹函数都不能匹配这两个调用。

compare(const_arr1, const_arr2): 这个调用正如我们所期待的，调用普通函数。该函数和将 `T` 绑定到 `const char*` 的第一个模板都是可行的，也都完全匹配。根据规则(3)(b)，会选择普通函数。从候选集合中去掉模板实例，只剩下普通函数可行。

compare(ch_arr1, ch_arr2): 这个调用也绑定到普通函数。候选者是将 `T` 绑定到 `char*` 的函数模板的版本，以及接受 `const char*` 实参的普通函数，两个函数都需要稍加转换将数组 `ch_arr1` 和 `ch_arr2` 转换为指针。因为两个函数一样匹配，所以普通函数优先于模板版本。

4. 转换与重载的函数模板

设计一组重载函数，其中一些是模板而另一些是普通函数，这可能是困难的。这样做需要深入理解类型之间的关系，具体而言，就是当涉及模板时可以发生和不能发生的隐式转换。

来看两个例子，看看为什么当重载集合中既有模板又有非模板版本的时候，设计正确工作的重载函数是困难的。首先，考虑用指针代替数组本身的 `compare` 调用：

```
char *p1 = ch_arr1, *p2 = ch_arr2;
compare(p1, p2);
```

这个调用与模板版本匹配！通常，我们希望无论是传递数组，还是传递指向该数组元素的指针，都获得同一函数。但是，在这个例子中，将 `char*` 绑定到 `T` 的函数模板与该调用完全匹配。普通版本仍然需要从 `char*` 到 `const char*` 的转换，所以优先选择函数模板。

另一个具有惊人结果的改变是，如果 `compare` 的模板版本有一个 `T` 类型的形参代替 `T` 的 `const` 引用，会发生的情况：

```
template <typename T> int compare2(T, T);
```

681 这个例子中，如果有一个普通类型的数组，则无论传递数组本身，还是传递指针，都将调用模板版本。调用非模板版本的唯一途径是在实参是 `const char` 或者 `const char*` 指针数组的时候：

```
// calls compare(T, T) with T bound to char*
compare(ch_arr1, ch_arr2);
// calls compare(T, T) with T bound to char*
compare(p1, p2);
// calls the ordinary function taking const char* parameters
compare(const_arr1, const_arr2);
const char *cp1 = const_arr1, *cp2 = const_arr2;
// calls the ordinary function taking const char* parameters
```

1. 此处原文误作“非重载”。——译者注

```
compare(cp1, cp2);
```

在这些情况下，普通函数和函数模板都完全匹配。像通常一样，当匹配同样好时，非模板版本优先。



最佳
实践

设计既包含函数模板又包含非模板函数的重载函数集合是困难的，因为可能会使函数的用户感到奇怪，定义函数模板特化（16.6 节）几乎总是比使用非模板版本更好。

习题

习题 16.61 实现 `compare` 函数的三个版本。在每个函数中包含一个输出语句，指出正在调用哪个函数。使用这些函数检查对其余问题的回答。

习题 16.62 对于本节定义的 `compare` 函数和变量，解释下面每个函数调用中，哪个函数被调用以及为什么。

```
compare(ch_arr1, const_arr1);
compare(ch_arr2, const_arr2);
compare(0, 0);
```

习题 16.63 对于下面的每个调用，列出候选函数和可行函数，指出调用是否有效，以及如果有效，调用哪个函数。

```
template <class T> T calc(T, T);
double calc(double, double);
template <> char calc<char>(char, char);
int ival; double dval; float fd;
calc(0, ival);          calc(0.25, dval);
calc(0, fd);            calc(0, 'J');
```

682

小结

模板是 C++ 语言与众不同的特性，是标准库的基础。模板是独立于类型的蓝图，编译器可以用它产生多种特定类型的实例。我们只需编写一次模板，编译器将为使用模板的不同类型实例化模板。既可以编写函数模板又可以编写类模板。

函数模板是建立算法库的基础，类模板是建立标准库容器和迭代器类型的基础。

编译模板需要编程环境的支持。语言为实例化模板定义了两个主要策略：包含模型和分别编译模型。这些模型规定了模板定义应该放在头文件还是源文件中，就此而言，它们影响着构建系统的方式。现在，所有编译器实现了包含模型，只有一些编译器实现了分别编译模型。编译器的用户指南应该会说明系统怎样管理模板。

显式模板实参使我们能够固定一个或多个模板形参的类型或值。显式实参使我们能够设计无需从对应实参推断模板类型的函数，也使我们能够对实参进行转换。

模板特化是一种特化的定义，它定义了模板的不同版本，将一个或多个形参绑定到特定类型或特定值。对于默认模板定义不适用的类型，特化非常有用。

术语

class template (类模板) 可以用来定义一组特定类型的类的类定义。类模板用 `template` 关键字后接用尖括号 (`<>`) 括住、以逗号分隔的一个或多个形参的列表来定义。

export keyword (导出关键字) 用来指出编译器必须记住相关模板定义位置的关键字，支持模板实例化的分别编译模型的编译器使用它。`export` 关键字一般与函数定义一起出现，类通常在相关类实现文件中声明为 `export`。在一个程序中，一个模板只能用 `export` 关键字定义一次。

function template (函数模板) 可用于不同类型的函数的定义。函数模板用 `template` 关键字后接用尖括号 (`<>`) 括住、以逗号分隔的一个或多个形参的列表来定义。

generic handle class (泛型句柄类) 保存和管理指向其他类的指针的类。泛型句柄接受单个类型形参，并且分配和管理指向该类型对象的指针。句柄类定义了必要的复制控制成员，它还定义了解引用操作符 (`*`) 和箭头成员访问操作符 (`->`)。泛型句柄不需要了解它管理的类型。

inclusion compilation model (包含编译模型) 编译器用来寻找模板定义的机制，它依赖于模板定义被包含在每个使用模板的文件中。一般而言，模板定义存储在一个头文件中，使用模板的任意文件必须包含该头文件。

instantiation (实例化) 用实际模板实参产生模板特定实例的编译器过程，在该实例中，用对应实参代替形参。函数基于调用中使用的实参自动实例化，使用类模板时必须显式提供模板实参。

member template (成员模板) 类或类模板

的是函数模板的成员。成员模板不能为虚。

nontype parameter (非类型形参) 表示值的模板形参。在实例化函数模板的时候，将每个非类型形参绑定到一个常量表达式，该常量表达式作为调用中使用的实参给出。在实例化类模板的时候，将每个非类型形参绑定到一个常量表达式，该常量表达式作为类实例化中使用的实参给出。

partial specialization (部分特化) 类模板的一个版本，其中指定了某些但非全部的模板形参。

separate compilation model (分别编译模型) 编译器用来查找模板定义的机制，它允许将模板定义和声明存储在独立的文件中。模板声明放在一个头文件中，而定义只在程序中出现一次，一般在源文件中，无论什么编程环境所支持的编译器都必须找到该源文件，并实例化程序所使用的模板版本。

template argument (模板实参) 在使用模板类型（如定义对象或在强制类型转换中指定类型）的时候，指定的类型或值。

template argument deduction (模板实参推断) 编译器用来确定实例化哪个函数模板的过程。编译器检查用模板形参指定的实参的类型，它用绑定到模板形参的类型或值自动实例化函数的一个版本。

template parameter (模板形参) 在模板形参表中指定的、可以在模板定义内部使用的名字。模板形参可以是类型形参或非类型形参。要使用模板，必须为每个模板形参指定实参，编译器使用这些类型或值来实例化类的一个版本，其中形参的使用以实参代替。当使用函数模板的时候，编译器从函数调用中的实参推

断模板实参，并使用推断得到的模板实参实例化特定函数。

template parameter list (模板形参表) 在模板定义或声明中使用的类型形参或非类型形参（以逗号分隔）的列表。

template specialization (模板特化) 类模板或类模板的成员的重定义，其中指定了模板形参。在定义了被特化的模板之前，不能出现该

模板的特化。在使用任意实参特化的模板之前，必须先出现模板特化。

type parameter (类型形参) 模板形参表中使用的表示类型的名字。当实例化模板的时候，将每个类型形参绑定到实际类型。在函数模板中，从实参类型中推断类型或者在调用中显式指定类型。类模板的类型实参必须在使用类的时候指定。

683
1
684

高级主题

目录

第 17 章 用于大型程序的工具.....	579
第 18 章 特殊工具与技术.....	631

这一部分涵盖了一些高级特征，虽然它们在适当情况下是有用的，但不是每个 C++ 程序员都需要。这些特征分为两类：用于大规模问题的，以及应用于特殊问题的。

第 17 章涵盖了异常处理、命名空间和多重继承。这些特征在大规模问题的环境中最有价值。

即使是可由一个开发人员编写的简单程序也可以从异常处理中获益，这是我们之所以要在第 6 章介绍异常处理基础知识的原因。但是，在需要大的编程团队的问题中，处理不可预期的运行时错误变得更加重要，更难管理。第 17 章将讨论另外一些有用的异常处理设施，还将更详细地介绍怎样处理异常、资源分配和回收异常的含义，以及怎样定义和使用自己的异常类。

大规模应用程序经常使用来自多个独立供应商的代码，如果将供应商定义的名字都放在一个命名空间，组合独立开发的库即使不是完全不可能，将是很困难的。独立开发的库几乎不可避免地会使用彼此相同的名字，一个库中定义的名字可能会与其他库中的同名冲突。为了避免名字冲突，可以将名字定义在 namespace 内。

其实从本书一开始我们已经使用了命名空间。只要用到标准库中的名字，其实都是

在使用名为 `std` 的命名空间中定义的名字。第 17 章将介绍怎样自定义命名空间。

第 17 章的最后介绍了一个重要但不常用的语言特征——多重继承。多重继承对于相当复杂的继承层次最为有用。

第 18 章讨论了几个特殊的工具和技术，这些工具和技术可应用于某些特定类型的问题。

第 18 章的第一节介绍了类怎样自定义优化内存管理，接着介绍 C++ 对运行时类型识别 (RTTI) 的支持，这种设施使我们能够在运行时确定对象的实际类型。

接下来，介绍怎样定义和使用类成员的指针。类成员的指针不同于指向普通数据或函数的指针，普通指针只根据对象或函数的类型而变化，而成员的指针还必须反映成员所属的类。

然后，介绍另外三种聚合类型：联合、嵌套类和局部类。

第 18 章最后简要介绍了一些固有的不可移植的特征：`volatile` 限定符、位域和链

686 接指示。

第 17 章

用于大型程序的工具

目录

17.1 异常处理.....	580
17.2 命名空间.....	599
17.3 多重继承与虚继承.....	614
小结.....	628
术语.....	628

用 C++ 解决的问题，其复杂性千变万化，有的问题一个程序员几小时便可解决，有的则需要用许多年开发和修改的数千万行代码构成的庞大系统。本书前面所介绍的内容对这些编程问题都同样适用。

C++ 语言包含的一些特征在问题比较复杂，非个人所能管理时最为有用。本章的主题就是这些特征，即异常处理、命名空间和多重继承。

相对于小的程序员团队所能开发的系统需求而言，大规模编程对程序设计语言的要求更高。大规模应用程序往往具有下列特殊要求：

(1) 更严格的正常运转时间以及更健壮的错误检测和错误处理。错误处理经常必须跨越独立开发的多个子系统进行。

(2) 能够用各种库（可能包含独立开发的库）构造程序。

(3) 能够处理更复杂的应用概念。

C++中有下列三个特征分别针对这些要求：异常处理、命名空间和多重继承。本章将介绍这三个特征。

17.1 异常处理

使用异常处理，程序中独立开发的各部分能够就程序执行期间出现的问题相互通信，并处理这些问题。程序的一个部分能够检测出本部分无法解决的问题，这个问题检测部分可以将问题传递给准备处理问题的其他部分。



通过异常我们能够将问题的检测和问题解决分离，这样程序的问题检测部分可以不必了解如何处理问题。

C++的异常处理中，需要由问题检测部分抛出一个对象给处理代码，通过这个对象的类型和内容，两个部分能够就出现了什么错误进行通信。

6.13 节介绍了在 C++ 中使用异常的基本概念和基本机制，在该节中，假定更复杂的书店应用程序可以通过异常就出现的问题进行通信。例如，如果操作数的 isbn 成员不匹配，Sales_item 的加操作符可以抛出一个异常：

```
// throws exception if both objects do not refer to the same isbn
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    if (!lhs.same_isbn(rhs))
        throw runtime_error("Data must refer to same ISBN");
    // ok, if we're still here the ISBNs are the same so it's okay to do the addition
    Sales_item ret(lhs);                // copy lhs into a local object that we'll return
    ret += rhs;                          // add in the contents of rhs
    return ret;                          // return a copy of ret
}
```

688

程序中将 Sales_item 对象相加的那些部分可以使用一个 try 块，以便在异常发生时捕获异常：

```
// part of the application that interacts with the user
Sales_item item1, item2, sum;
while (cin >> item1 >> item2) {        // read two transactions
    try {
        sum = item1 + item2;            // calculate their sum
        // use sum
    } catch (const runtime_error &e) {
```

```

        cerr << e.what() << " Try again.\n"
            << endl;
    }
}

```

本节将扩展对这些基础知识的讨论并涵盖另外一些异常处理设施。有效使用异常处理需要理解：在抛出异常时会发生什么，在捕获异常时又会发生什么，还有用来传递错误的对象的含义。

17.1.1 抛出类类型的异常

异常是通过抛出（throw）对象而引发（raise）的。该对象的类型决定应该激活哪个处理代码。被选中的处理代码是调用链中与该对象类型匹配且离抛出异常位置最近的那个。

异常以类似于将实参传递给函数的方式抛出和捕获。异常可以是可传给非引用形参的任意类型的对象，这意味着必须能够复制该类型的对象。

回忆一下，传递数组或函数类型实参的时候，该实参自动转换为一个指针。被抛出的对象将发生同样的自动转换，因此，不存在数组或函数类型的异常。相反，如果抛出一个数组，被抛出的对象转换为指向数组首元素的指针，类似地，如果抛出一个函数，函数被转换为指向该函数的指针（7.9 节）。

执行 throw 的时候，不会执行跟在 throw 后面的语句，而是将控制从 throw 转移到匹配的 catch，该 catch 可以是同一函数中局部的 catch，也可以在直接或间接调用发生异常的函数的另一个函数中。控制从一个地方传到另一地方，这有两个重要含义：

- (1) 沿着调用链的函数提早退出。17.1.2 节将讨论函数因异常而退出时会发生什么。
- (2) 一般而言，在处理异常的时候，抛出异常的块中的局部存储不存在了。

因为在处理异常的时候会释放局部存储，所以被抛出的对象就不能再局部存储，而是用 throw 表达式初始化一个称为**异常对象**（exception object）的特殊对象。异常对象由编译器管理，而且保证驻留在可能被激活的任意 catch 都可以访问的空间。这个对象由 throw 创建，并被初始化为被抛出的表达式的副本。异常对象将传给对应的 catch，并且在完全处理了异常之后撤销。

689



异常对象通过复制被抛出表达式的结果创建，该结果必须是可以复制的类型。

1. 异常对象与继承

在实践中，许多应用程序所抛出的表达式，其类型都来自某个继承层次。正如 17.1.7 节将介绍的，标准异常（6.13 节）定义在一个继承层次中。目前重要的是，知道 throw 表达式的形式如何与因继承而相关的类型相互影响。



当抛出一个表达式的时候，被抛出对象的静态编译时类型将决定异常对象的类型。

通常，使用静态类型抛出对象不成问题。当抛出一个异常的时候，通常在抛出点构造将抛出的对象，该对象表示出了什么问题，所以我们知道确切的异常类型。

2. 异常与指针

用抛出表达式抛出静态类型时，比较麻烦的一种情况是，在抛出中对指针解引用。对指针解引用的结果是一个对象，其类型与指针的类型匹配。如果指针指向继承层次中的一种类型，指针所指对象的类型就有可能与指针的类型不同。无论对象的实际类型是什么，异常对象的类型都与指针的静态类型相匹配。如果该指针是一个指向派生类对象的基类类型指针，则那个对象将被分割（15.3.1 节），只抛出基类部分。

如果抛出指针本身，可能会引发比分割对象更严重的问题。具体而言，抛出指向局部对象的指针总是错误的，其理由与从函数返回指向局部对象的指针是错误的是一样的（7.3.2 节）。抛出指针的时候，必须确定进入处理代码时指针所指向的对象存在。

如果抛出指向局部对象的指针，而且处理代码在另一函数中，则执行处理代码时指针所指向的对象将不再存在。即使处理代码在同一函数中，也必须确信指针所指向的对象在 catch 处存在。如果指针指向某个在 catch 之前退出的块中的对象，那么，将在 catch 之前撤销该局部对象。

690



抛出指针通常是个坏主意：抛出指针要求在对处理代码存在的任意地方存在指针所指向的对象。

习题

习题 17.1 下面的 throw 语句中，异常对象的类型是什么？

```
(a) range_error r("error"); (b) exception *p = &r;
    throw r;                  throw *p;
```

习题 17.2 如果第二个 throw 语句写成 throw p，会发生什么情况？

17.1.2 栈展开

抛出异常的时候，将暂停当前函数的执行，开始查找匹配的 catch 子句。首先检查 throw 本身是否在 try 块内部，如果是，检查与该 try 相关的 catch 子句，看是否其中之一与被抛出对象相匹配。如果找到匹配的 catch，就处理异常；如果找不到，就退出当前函数（释放当前函数的内存并撤销局部对象），并且继续在调用函数中查找。

如果对抛出异常的函数的调用是在 try 块中，则检查与该 try 相关的 catch 子句。如果找到匹配的 catch，就处理异常；如果找不到匹配的 catch，调用函数也退出，并且继续在调用这个函数的函数中查找。

这个过程，称之为栈展开（stack unwinding），沿嵌套函数调用链继续向上，直至为异常找到一个 catch 子句。只要找到能够处理异常的 catch 子句，就进入该 catch 子句，并在该处理代码中继续执行。当 catch 结束的时候，在紧接在与该 try 块相关的最后一个 catch 子句之后的点继续执行。

1. 为局部对象调用析构函数

栈展开期间，提早退出包含 throw 的函数和调用链中可能的其他函数。一般而言，这些函

数已经创建了可以在退出函数时撤销的局部对象。因异常而退出函数时，编译器保证适当地撤销局部对象。每个函数退出的时候，它的局部存储都被释放，在释放内存之前，撤销在异常发生之前创建的所有对象。如果局部对象是类类型的，就自动调用该对象的析构函数。通常，编译器不撤销内置类型的对象。

691



栈展开期间，释放局部对象所用的内存并运行类类型局部对象的析构函数。

如果一个块直接分配资源，而且在释放资源之前发生异常，在栈展开期间将不会释放该资源。例如，一个块可以通过调用 `new` 动态分配内存，如果该块因异常而退出，编译器不会删除该指针，已分配的内存将不会释放。

由类类型对象分配的资源一般会被适当地释放。运行局部对象的析构函数，由类类型对象分配的资源通常由它们的析构函数释放。17.1.8 节说明面对异常使用类管理资源分配的编程技术。

2. 析构函数应该从不抛出异常

栈展开期间会经常执行析构函数。在执行析构函数的时候，已经引发了异常但还没有处理它。如果在这个过程中析构函数本身抛出新的异常，又会发生什么呢？新的异常应该取代仍未处理的早先的异常吗？应该忽略析构函数中的异常吗？

答案是：在为某个异常进行栈展开的时候，析构函数如果又抛出自己的未经处理的另一个异常，将会导致调用标准库 `terminate` 函数。一般而言，`terminate` 函数将调用 `abort` 函数，强制从整个程序非正常退出。

因为 `terminate` 函数结束程序，所以析构函数做任何可能导致异常的事情通常都是非常糟糕的主意。在实践中，因为析构函数释放资源，所以它不太可能抛出异常。标准库类型都保证它们的析构函数不会引发异常。

3. 异常与构造函数

与析构函数不同，构造函数内部所做的事情经常会抛出异常。如果在构造函数对象的时候发生异常，则该对象可能只是部分被构造，它的一些成员可能已经初始化，而另一些成员在异常发生之前还没有初始化。即使对象只是部分被构造了，也要保证将会适当地撤销已构造的成员。

类似地，在初始化数组或其他容器类型的元素的时候，也可能发生异常，同样，也要保证将会适当地撤销已构造的元素。

4. 未捕获的异常终止程序

不能不处理异常。异常是足够重要的、使程序不能继续正常执行的事件。如果找不到匹配的 `catch`，程序就调用库函数 `terminate`。

692

17.1.3 捕获异常

`catch` 子句 (`catch clause`) 中的异常说明符 (`exception specifier`) 看起来像只包含一个形参的形参表，异常说明符是在其后跟一个 (可选) 形参名的类型名。

说明符的类型决定了处理代码能够捕获的异常种类。类型必须是完全类型，即必须是内置类

型或者是已经定义的程序员自定义类型。类型的前向声明不行。

当 `catch` 为了处理异常只需要了解异常的类型的时候，异常说明符可以省略形参名；如果处理代码需要已发生异常的类型之外的信息，则异常说明符就包含形参名，`catch` 使用这个名字访问异常对象。

1. 查找匹配的处理代码

在查找匹配的 `catch` 期间，找到的 `catch` 不必是与异常最匹配的那个 `catch`，相反，将选中第一个找到的可以处理该异常的 `catch`。因此，在 `catch` 子句列表中，最特殊的 `catch` 必须最先出现。

异常与 `catch` 异常说明符匹配的规则比匹配实参和形参类型的规则更严格，大多数转换都不允许——除下面几种可能的区别之外，异常的类型与 `catch` 说明符的类型必须完全匹配：

- 允许从非 `const` 到 `const` 的转换。也就是说，非 `const` 对象的 `throw` 可以与指定接受 `const` 引用的 `catch` 匹配。
- 允许从派生类型到基类类型的转换。
- 将数组转换为指向数组类型的指针，将函数转换为指向函数类型的适当指针。

在查找匹配 `catch` 的时候，不允许其他转换。具体而言，既不允许标准算术转换，也不允许为类类型定义的转换。

2. 异常说明符

进入 `catch` 的时候，用异常对象初始化 `catch` 的形参。像函数形参一样，异常说明符类型可以是引用。异常对象本身是被抛出对象的副本。是否再次将异常对象复制到 `catch` 位置取决于异常说明符类型。

693

如果说明符不是引用，就将异常对象复制到 `catch` 形参中，`catch` 操作异常对象的副本，对形参所做的任何改变都只作用于副本，不会作用于异常对象本身。如果说明符是引用，则像引用形参一样，不存在单独的 `catch` 对象，`catch` 形参只是异常对象的另一名字。对 `catch` 形参所做的改变作用于异常对象。

3. 异常说明符与继承

像形参声明一样，基类的异常说明符可以用于捕获派生类型的异常对象，而且，异常说明符的静态类型决定 `catch` 子句可以执行的动作。如果被抛出的异常对象是派生类类型的，但由接受基类类型的 `catch` 处理，那么，`catch` 不能使用派生类特有的任何成员。



最佳
实践

通常，如果 `catch` 子句处理因继承而相关的类型的异常，它就应该将自己的形参定义为引用。

如果 `catch` 形参是引用类型，`catch` 对象就直接访问异常对象，`catch` 对象的静态类型可以与 `catch` 对象所引用的异常对象的动态类型不同。如果异常说明符不是引用，则 `catch` 对象是异常对象的副本，如果 `catch` 对象是基类类型对象而异常对象是派生类型的，就将异常对象分割（15.3.1 节）为它的基类子对象。

而且，正如 15.2.4 节所介绍的，对象（相对于引用）不是多态的。当通过对象而不是引用使用虚函数的时候，对象的静态类型和动态类型相同，函数是虚函数也一样。只有通过引用或指针