小结

649

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

在C++语言中，动态绑定只作用于虚函数，并且需要通过指针或引用调用。

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

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

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

术语表

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

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

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

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

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

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

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

650

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

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

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

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

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

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

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

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

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

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

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

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

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

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

运行时绑定（run-time binding） 参见“动态绑定”。

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

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

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

第 16 章
模板与泛型编程

内容

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

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

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

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

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

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

16.1 定义模板

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

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

这两个函数几乎是相同的，唯一的差异是参数的类型，函数体则完全一样。

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

16.1.1 函数模板

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

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

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



在模板定义中，模板参数列表不能为空。

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

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

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

实例化函数模板

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

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

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

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

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

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

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

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

模板类型参数

654

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

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

类型参数前必须使用关键字 class 或 typename：

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

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

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

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

非类型模板参数

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

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

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

655

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

当我们调用这个版本的 compare 时：

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

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

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

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

在模板定义内，模板非类型参数是一个常量值。在需要常量表达式的地方，可以使用

非类型参数，例如，指定数组大小。



非类型模板参数的模板实参必须是常量表达式。

inline 和 constexpr 的函数模板

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

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



编写类型无关的代码

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

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

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

656

你可能认为既使用<运算符又使用>运算符来进行比较操作会更为自然：

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

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

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

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

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



模板程序应该尽量减少对实参类型的要求。

模板编译

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

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

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

657



函数模板和类模板成员函数的定义通常放在头文件中。

关键概念：模板和头文件

模板包含两种名字：

- 那些不依赖于模板参数的名字
- 那些依赖于模板参数的名字

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

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

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

大多数编译错误在实例化期间报告

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

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

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

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

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

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

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

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

658

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

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



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

16.1.1 节练习

练习 16.1：给出实例化的定义。

练习 16.2：编写并测试你自己版本的 compare 函数。

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

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

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

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

练习 16.7：编写一个 constexpr 模板，返回给定数组的大小。

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

16.1.2 类模板



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

659

定义类模板

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

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

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

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

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

660 ## 实例化类模板

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

例如，为了用我们的 Blob 模板定义一个类型，必须提供元素类型：

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

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

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

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

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

对我们指定的每一种元素类型，编译器都生成一个不同的类：

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

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



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

在模板作用域中引用模板类型



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

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

661

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

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

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

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

类模板的成员函数

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

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

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

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

对应的 Blob 的成员应该是这样的：

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

check 和元素访问成员

我们首先定义 check 成员，它检查一个给定的索引：

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

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

下标运算符和 back 函数用模板参数指出返回类型，其他未变：

662

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

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

pop_back 函数与原 StrBlob 的成员几乎相同：

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

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

Blob 构造函数

与其他任何定义在类模板外的成员一样，构造函数的定义要以模板参数开始：

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

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

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

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

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

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

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

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

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

类模板成员函数的实例化

663

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

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

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

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



默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被实例化。

在类代码内简化模板类名的使用

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

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

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

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

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

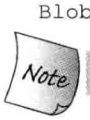
在类模板外使用类模板名

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

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

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

```
BlobPtr<T> ret = *this;
```



在一个类模板的作用域内，我们可以直接使用模板名而不必指定模板实参。

类模板和友元

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

一对一友好关系

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

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

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

665

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

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

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

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

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

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

通用和特定的模板友好关系

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

```
// 前置声明，在将模板的一个特定实例声明为友元时要用到
template <typename T> class Pal;
class C { // C 是一个普通的非模板类
    friend class Pal<C>; // 用类 C 实例化的 Pal 是 C 的一个友元
    // Pal2 的所有实例都是 C 的友元；这种情况无须前置声明
    template <typename T> friend class Pal2;
};
template <typename T> class C2 { // C2 本身是一个类模板
    // C2 的每个实例将相同实例化的 Pal 声明为友元
    friend class Pal<T>; // Pal 的模板声明必须在作用域之内
    // Pal2 的所有实例都是 C2 的每个实例的友元，不需要前置声明
    template <typename X> friend class Pal2;
    // Pal3 是一个非模板类，它是 C2 所有实例的友元
    friend class Pal3; // 不需要 Pal3 的前置声明
};
```

为了让所有实例成为友元，友元声明中必须使用与类模板本身不同的模板参数。

666 令模板自己的类型参数成为友元

C++ 11

在新标准中，我们可以将模板类型参数声明为友元：

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

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

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

模板类型别名

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

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

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

C++ 11

但是，新标准允许我们为类模板定义一个类型别名：

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

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

一个模板类型别名是一族类的别名：

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

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

当我们定义一个模板类型别名时，可以固定一个或多个模板参数：

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

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

667 类模板的 static 成员

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

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

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

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

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

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

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

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

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

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

类似任何其他成员函数，一个 static 成员函数只有在使用时才会实例化。

16.1.2 节练习

668

练习 16.9：什么是函数模板？什么是类模板？

练习 16.10：当一个类模板被实例化时，会发生什么？

练习 16.11：下面 List 的定义是错误的。应如何修正它？

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

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

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

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

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

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

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



16.1.3 模板参数

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

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

模板参数与作用域

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

669

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

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

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

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

模板声明

模板声明必须包含模板参数：

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

与函数参数相同，声明中的模板参数的名字不必与定义中相同：

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

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



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

使用类的类型成员

回忆一下，我们用作用域运算符（::）来访问 static 成员和类型成员（参见 7.4 节，第 253 页和 7.6 节，第 269 页）。在普通（非模板）代码中，编译器掌握类的定义。因此，它知道通过作用域运算符访问的名字是类型还是 static 成员。例如，如果我们写下 string::size_type，编译器有 string 的定义，从而知道 size_type 是一个类型。 670

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

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

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

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

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

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



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

默认模板实参

C++ 11 就像我们能为函数参数提供默认实参一样（参见 6.5.1 节，第 211 页），我们也可以提供**默认模板实参**（default template argument）。在新标准中，我们可以为函数和类模板提供默认实参。而更早的 C++标准只允许为类模板提供默认实参。

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

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

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

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

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

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

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

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

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

模板默认实参与类模板

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

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

```
    // 对数值的各种操作
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; // 空<>表示我们希望使用默认类型
```

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

16.1.3 节练习

672

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

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

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

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

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

16.1.4 成员模板

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

普通（非模板）类的成员模板

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

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

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

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

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

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

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

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

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

类模板的成员模板

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

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

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

此构造函数有自己的模板类型参数 It，作为它的两个函数参数的类型。

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

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

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

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

实例化与成员模板

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

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

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

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

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

16.1.4 节练习

675

练习 16.21：编写你自己的 DebugDelete 版本。

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

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

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

16.1.5 控制实例化





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

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

形式：

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

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

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

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

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

```
// Application.cc
// 这些模板类型必须在程序其他位置进行实例化
extern template class Blob<string>;
extern template int compare(const int&, const int&);
Blob<string> sa1, sa2; // 实例化会出现在其他位置
// Blob<int>及其接受 initializer_list 的构造函数在本文件中实例化
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); // 拷贝构造函数在本文件中实例化
int i = compare(a1[0], a2[0]); // 实例化出现在其他位置
```

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

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

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



对每个实例化声明，在程序中某个位置必须有其显式的实例化定义。

实例化定义会实例化所有成员

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



在一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数。

16.1.5 节练习

练习 16.25：解释下面这些声明的含义：

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

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

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

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



16.1.6 效率与灵活性

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

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

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

如何处理删除器的差异实际上就是这两个类功能的差异。但是，如我们将要看到的，这一实现策略上的差异可能对性能有重要影响。 677

在运行时绑定删除器

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

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

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

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

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

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

在编译时绑定删除器

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

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

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

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

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

16.1.6 节练习

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

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

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

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

16.2 模板实参推断

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

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

16.2.1 类型转换与模板类型参数



与非模板函数一样，我们在一次调用中传递给函数模板的实参被用来初始化函数的形参。如果一个函数形参的类型使用了模板类型参数，那么它采用特殊的初始化规则。只有很有限的几种类型转换会自动地应用于这些实参。编译器通常不是对实参进行类型转换，而是生成一个新的模板实例。 679

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

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

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

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

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

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

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



将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有 const 转换及数组或函数到指针的转换。 680

使用相同模板参数类型的函数形参

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

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

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

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

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

现在用户可以提供不同类型的实参了：

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

当然，必须定义了能比较这些类型的值的<运算符。

正常类型转换应用于普通函数实参

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

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

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

681

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

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



如果函数参数类型不是模板参数，则对实参进行正常的类型转换。

16.2.1 节练习

练习 16.32：在模板实参推断过程中发生了什么？

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

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

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

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

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

练习 16.36：进行下面的调用会发生什么：

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

16.2.2 函数模板显式实参



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

682

指定显式模板实参

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

我们可以定义表示返回类型的第三个模板参数，从而允许用户控制返回类型：

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

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

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

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

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

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

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

则我们总是必须为所有三个形参指定实参：

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

正常类型转换应用于显式指定的实参

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

683

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

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

16.2.2 节练习

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

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

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

16.2.3 尾置返回类型与类型转换

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

元素的引用：

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

我们并不知道返回结果的准确类型，但知道所需类型是所处理的序列的元素类型：

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

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

684

C++ 11

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

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

进行类型转换的标准库模板类

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

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

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

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

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

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

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

685

```
// 为了使用模板参数的成员，必须用 typename，参见 16.1.3 节（第 593 页）
template <typename It>
auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type
{
    // 处理序列
    return *beg; // 返回序列中一个元素的拷贝
}
```

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

表 16.1：标准类型转换模板

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

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

转换，从而 type 具有与 T 相同的类型。

16.2.3 节练习

686

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

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

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

16.2.4 函数指针和实参推断

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

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

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

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

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

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

我们可以通过使用显式模板实参来消除 func 调用的歧义：

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

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



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

687

16.2.5 模板实参推断和引用

为了理解如何从函数调用进行类型推断，考虑下面的例子：

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

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

从左值引用函数参数推断类型

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

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

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

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

从右值引用函数参数推断类型

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

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

688 **引用折叠和右值引用参数**

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

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

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

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

在这种情况下，我们可以使用第二个例外绑定规则：如果我们间接创建一个引用的引用，则这些引用形成了“折叠”。在所有情况下（除了一个例外），引用会折叠成一个普通的左值引用类型。在新标准中，折叠规则扩展到右值引用。只在一种特殊情况下引用会折叠成右值引用：右值引用的右值引用。即，对于一个给定类型 X：

C++ 11

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



引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数。

如果将引用折叠规则和右值引用的特殊类型推断规则组合在一起，则意味着我们可以对一个左值调用 f3。当我们将一个左值传递给 f3 的（右值引用）函数参数时，编译器推断 T 为一个左值引用类型：

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

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

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

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

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

这两个规则导致了两个重要结果：

689

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

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



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

编写接受右值引用参数的模板函数

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

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

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

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

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

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

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

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

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

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

690

16.2.5 节练习

练习 16.42：对下面每个调用，确定 T 和 val 的类型：

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

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

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

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

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

16.2.6 理解 std::move

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

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

何类型的实参，因此我们不会惊讶于它是一个函数模板。

std::move 是如何定义的

标准库是这样定义 move 的：

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

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

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

std::move 是如何工作的



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

691

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

因此，这个调用实例化 move<string>，即函数

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

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

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

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

因此，这个调用实例化 move<string&>，即

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

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

从一个左值 static_cast 到一个右值引用是允许的

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

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

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

16.2.6 节练习

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

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

16.2.7 转发

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

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

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

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

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

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

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

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

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

693

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

定义能保持类型信息的函数参数



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

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

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

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



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

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

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

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

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

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

在调用中使用 std::forward 保持类型信息

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

C++ 11

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

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

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



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

使用 forward，我们可以再次重写翻转函数：

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

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



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

16.2.7 节练习

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

16.3 重载与模板

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

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

影响：

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



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

编写重载模板

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

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

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

接下来，我们将定义打印指针的 debug_rep 版本：

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

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

节（第 617 页）介绍如何处理字符指针。

我们可以这样使用这些函数：

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

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

如果我们用一个指针调用 debug_rep：

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

两个函数都生成可行的实例：

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

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

697

多个可行模板

作为另外一个例子，考虑下面的调用：

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

此例中的两个模板都是可行的，而且两个都是精确匹配：

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

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

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



当有多个重载模板对一个调用提供同样好的匹配时，应选择最特例化的版本。

非模板和模板重载

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

的 string：

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

现在，当我们对一个 string 调用 debug_rep 时：

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

有两个同样好的可行函数：

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

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



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

重载模板和类型转换

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

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

本例中所有三个 debug_rep 版本都是可行的：

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

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

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

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

缺少声明可能导致程序行为异常

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

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

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



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

16.3 节练习

练习 16.48：编写你自己版本的 debug_rep 函数。

练习 16.49：解释下面每个调用会发生什么：

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

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



16.4 可变参数模板

C++11

一个**可变参数模板**（variadic template）就是一个接受可变数目参数的模板函数或模板类。可变数目的参数被称为**参数包**（parameter packet）。存在两种参数包：**模板参数包**

700
（template parameter packet），表示零个或多个模板参数；**函数参数包**（function parameter packet），表示零个或多个函数参数。

我们用一个省略号来指出一个模板参数或函数参数表示一个包。在一个模板参数列表

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

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

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

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

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

编译器会为 foo 实例化出四个不同的版本：

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

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

sizeof...运算符

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

C++ 11

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

16.4 节练习

701

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

练习 16.52：编写一个程序验证上一题的答案。

16.4.1 编写可变参数函数模板

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

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

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

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

这段程序的关键部分是可变参数函数中对 print 的调用：

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

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

702

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

递归会执行如下：

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

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

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

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



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

16.4.1 节练习

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

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

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



16.4.2 包扩展

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

703

例如，我们的 print 函数包含两个扩展：

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

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

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

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

最后两个实参的类型和模式一起确定了尾置参数的类型。此调用被实例化为：

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

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

用等价于：

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

理解包扩展

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

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

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

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

就好像我们这样编写代码一样

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

与之相对，下面的模式会编译失败

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

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

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

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



扩展中的模式会独立地应用于包中的每个元素。

16.4.2 节练习

练习 16.56：编写并测试可变参数版本的 errorMsg。

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



16.4.3 转发参数包

C++ 11 在新标准下，我们可以组合使用可变参数模板与 forward 机制来编写函数，实现将

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

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

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

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

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

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

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

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

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

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

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

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

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

construct 调用中的模式会扩展出

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

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

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

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

```
std::forward<string>(string("the end"))
```

forward<string>的结果类型是 string&&，因此 construct 将得到一个右值引用实参。construct 会继续将此实参传递给 string 的移动构造函数来创建新元素。

706

建议：转发和可变参数模板

可变参数函数通常将它们的参数转发给其他函数。这种函数通常具有与我们的 emplace_back 函数一样的形式：

```
// fun 有零个或多个参数，每个参数都是一个模板参数类型的右值引用
template<typename... Args>
void fun(Args&&... args) // 将 Args 扩展为一个右值引用的列表
{
    // work 的实参既扩展 Args 又扩展 args
    work(std::forward<Args>(args)...);
}
```

这里我们希望将 fun 的所有实参转发给另一个名为 work 的函数，假定由它完成函数的实际工作。类似 emplace_back 中对 construct 的调用，work 调用中的扩展既扩展了模板参数包也扩展了函数参数包。

由于 fun 的参数是右值引用，因此我们可以传递给它任意类型的实参；由于我们使用 std::forward 传递这些实参，因此它们的所有类型信息在调用 work 时都会得到保持。

16.4.3 节练习

练习 16.58：为你的 StrVec 类及你为 16.1.2 节（第 591 页）练习中编写的 Vec 类添加 emplace_back 函数。

练习 16.59：假定 s 是一个 string，解释调用 svec.emplace_back(s) 会发生什么。

练习 16.60：解释 make_shared（参见 12.1.1 节，第 401 页）是如何工作的。

练习 16.61：定义你自己版本的 make_shared。

16.5 模板特例化

编写单一模板，使之对任何可能的模板实参都是最适合的，都能实例化，这并不总是能办到。在某些情况下，通用模板的定义对特定类型是不适合的：通用定义可能编译失败或做得不正确。其他时候，我们也可以利用某些特定知识来编写更高效的代码，而不是从通用模板实例化。当我们不能（或不希望）使用模板版本时，可以定义类或函数模板的一个特例化版本。

我们的 compare 函数是一个很好的例子，它展示了函数模板的通用定义不适合一个特定类型（即字符指针）的情况。我们希望 compare 通过调用 strcmp 比较两个字符指针而非比较指针值。实际上，我们已经重载了 compare 函数来处理字符串字面常量（参见 16.1.1 节，第 579 页）：

707

```
// 第一个版本；可以比较任意两个类型
template <typename T> int compare(const T&, const T&);
// 第二个版本处理字符串字面常量
template<size_t N, size_t M>
int compare(const char (&)[N], const char (&)[M]);
```

但是，只有当我们传递给 compare 一个字符串字面常量或者一个数组时，编译器才会调用接受两个非类型模板参数的版本。如果我们传递给它字符指针，就会调用第一个版本：

```
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2);          // 调用第一个模板
compare("hi", "mom");     // 调用有两个非类型参数的版本
```

我们无法将一个指针转换为一个数组的引用，因此当参数是 p1 和 p2 时，第二个版本的 compare 是不可行的。

为了处理字符指针（而不是数组），可以为第一个版本的 compare 定义一个**模板特例化**（template specialization）版本。一个特例化版本就是模板的一个独立的定义，在其中一个或多个模板参数被指定为特定的类型。

定义函数模板特例化

当我们特例化一个函数模板时，必须为原模板中的每个模板参数都提供实参。为了指出我们正在实例化一个模板，应使用关键字 template 后跟一个空尖括号对（<>）。空尖括号指出我们将为原模板的所有模板参数提供实参：

```
// compare 的特殊版本，处理字符数组的指针
template <>
int compare(const char* const &p1, const char* const &p2)
{
    return strcmp(p1, p2);
}
```

理解此特例化版本的困难之处是函数参数类型。当我们定义一个特例化版本时，函数参数类型必须与一个先前声明的模板中对应的类型匹配。本例中我们特例化：

```
template <typename T> int compare(const T&, const T&);
```

其中函数参数为一个 const 类型的引用。类似类型别名，模板参数类型、指针及 const 之间的相互作用会令人惊讶（参见 2.5.1 节，第 60 页）。

我们希望定义此函数的一个特例化版本，其中 T 为 const char*。我们的函数要求一个指向此类型 const 版本的引用。一个指针类型的 const 版本是一个常量指针而不是指向 const 类型的指针（参见 2.4.2 节，第 56 页）。我们需要在特例化版本中使用的类型是 const char * const &，即一个指向 const char 的 const 指针的引用。

函数重载与模板特例化

708

当定义函数模板的特例化版本时，我们本质上接管了编译器的工作。即，我们为原模板的一个特殊实例提供了定义。重要的是要弄清：一个特例化版本本质上是一个实例，而非函数名的一个重载版本。



特例化的本质是实例化一个模板，而非重载它。因此，特例化不影响函数匹配。

我们将一个特殊的函数定义为一个特例化版本还是一个独立的非模板函数，会影响到函数匹配。例如，我们已经定义了两个版本的 compare 函数模板，一个接受数组引用参数，另一个接受 const T&。我们还定义了一个特例化版本来处理字符指针，这对函数匹配没有影响。当我们对字符串字面常量调用 compare 时

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

对此调用，两个函数模板都是可行的，且提供同样好的（即精确的）匹配。但是，接受字符数组参数的版本更特例化（参见 16.3 节，第 615 页），因此编译器会选择它。

如果我们将接受字符指针的 compare 版本定义为一个普通的非模板函数（而不是模板的一个特例化版本），此调用的解析就会不同。在此情况下，将会有三个可行的函数：两个模板和非模板的字符指针版本。所有三个函数都提供同样好的匹配。如前所述，当一个非模板函数提供与函数模板同样好的匹配时，编译器会选择非模板版本（参见 16.3 节，第 615 页）。

关键概念：普通作用域规则应用于特例化

为了特例化一个模板，原模板的声明必须在作用域中。而且，在任何使用模板实例的代码之前，特例化版本的声明也必须在作用域中。

对于普通类和函数，丢失声明的情况（通常）很容易发现——编译器将不能继续处理我们的代码。但是，如果丢失了一个特例化版本的声明，编译器通常可以用原模板生成代码。由于在丢失特例化版本时编译器通常会实例化原模板，很容易产生模板及其特例化版本声明顺序导致的错误，而这种错误又很难查找。

如果一个程序使用一个特例化版本，而同时原模板的一个实例具有相同的模板实参集合，就会产生错误。但是，这种错误编译器又无法发现。

Best Practices 模板及其特例化版本应该声明在同一个头文件中。所有同名模板的声明应该放在前面，然后是这些模板的特例化版本。

709 ## 类模板特例化

除了特例化函数模板，我们还可以特例化类模板。作为一个例子，我们将为标准库 hash 模板定义一个特例化版本，可以用它来将 Sales_data 对象保存在无序容器中。默认情况下，无序容器使用 hash<key_type>（参见 11.4 节，第 394 页）来组织其元素。为了让我们自己的数据类型也能使用这种默认组织方式，必须定义 hash 模板的一个特例化版本。一个特例化 hash 类必须定义：

- 一个重载的调用运算符（参见 14.8 节，第 506 页），它接受一个容器关键字类型的对象，返回一个 size_t。
- 两个类型成员，result_type 和 argument_type，分别调用运算符的返回类型和参数类型。
- 默认构造函数和拷贝赋值运算符（可以隐式定义，参见 13.1.2 节，第 443 页）。

在定义此特例化版本的 hash 时，唯一复杂的地方是：必须在原模板定义所在的命名空间中特例化它。我们将在 18.2 节（第 695 页）中介绍更多命名空间的相关内容。现在，我们只需知道——我们可以向命名空间添加成员。为了达到这一目的，首先必须打开命名空间：

```
// 打开 std 命名空间，以便特例化 std::hash
namespace std {
```

```
} // 关闭 std 命名空间；注意：右花括号之后没有分号
```

花括号对之间的任何定义都将成为命名空间 std 的一部分。

下面的代码定义了一个能处理 Sales_data 的特例化 hash 版本：

```
// 打开 std 命名空间，以便特例化 std::hash
namespace std {
template <> // 我们正在定义一个特例化版本，模板参数为 Sales_data
struct hash<Sales_data>
{
    // 用来散列一个无序容器的类型必须要定义下列类型
    typedef size_t result_type;
    typedef Sales_data argument_type; // 默认情况下，此类型需要==
    size_t operator()(const Sales_data& s) const;
    // 我们的类使用合成的拷贝控制成员和默认构造函数
};
size_t
hash<Sales_data>::operator()(const Sales_data& s) const
{
    return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue);
}
} // 关闭 std 命名空间；注意：右花括号之后没有分号
```

我们的 hash<Sales_data>定义以 template<>开始，指出我们正在定义一个全特例化的模板。我们正在特例化的模板名为 hash，而特例化版本为 hash<Sales_data>。接下来的类成员是按照特例化 hash 的要求而定义的。 710

类似其他任何类，我们可以在类内或类外定义特例化版本的成员，本例中就是在类外定义的。重载的调用运算符必须为给定类型的值定义一个哈希函数。对于一个给定值，任何时候调用此函数都应该返回相同的结果。一个好的哈希函数对不相等的对象（几乎总是）应该产生不同的结果。

在本例中，我们将定义一个好的哈希函数的复杂任务交给了标准库。标准库为内置类型和很多标准库类型定义了 hash 类的特例化版本。我们使用一个（未命名的）hash<string>对象来生成 bookNo 的哈希值，用一个 hash<unsigned>对象来生成 units_sold 的哈希值，用一个 hash<double>对象来生成 revenue 的哈希值。我们将这些结果进行异或运算（参见 4.8 节，第 137 页），形成给定 Sales_data 对象的完整的哈希值。

值得注意的是，我们的 hash 函数计算所有三个数据成员的哈希值，从而与我们为 Sales_data 定义的 operator==（参见 14.3.1 节，第 497 页）是兼容的。默认情况下，为了处理特定关键字类型，无序容器会组合使用 key_type 对应的特例化 hash 版本和 key_type 上的相等运算符。

假定我们的特例化版本在作用域中，当将 Sales_data 作为容器的关键字类型时，编译器就会自动使用此特例化版本：

```
// 使用 hash<Sales_data>和 14.3.1 节（第 497 页）中 Sales_data 的 operator==
unordered_multiset<Sales_data> SDset;
```

由于 hash<Sales_data>使用 Sales_data 的私有成员，我们必须将它声明为

Sales_data 的友元：

```
template <class T> class std::hash; // 友元声明所需要的
class Sales_data {
friend class std::hash<Sales_data>;
    // 其他成员定义，如前
};
```

这段代码指出特殊实例 hash<Sales_data>是 Sales_data 的友元。由于此实例定义在 std 命名空间中，我们必须记得在 friend 声明中应使用 std::hash。



为了让 Sales_data 的用户能使用 hash 的特例化版本，我们应该在 Sales_data 的头文件中定义该特例化版本。

类模板部分特例化

与函数模板不同，类模板的特例化不必为所有模板参数提供实参。我们可以只指定一
711 部分而非所有模板参数，或是参数的一部分而非全部特性。一个类模板的**部分特例化**（partial specialization）本身是一个模板，使用它时用户还必须为那些在特例化版本中未指定的模板参数提供实参。



我们只能部分特例化类模板，而不能部分特例化函数模板。

在 16.2.3 节（第 605 页）中我们介绍了标准库 remove_reference 类型。该模板是通过一系列的特例化版本来完成其功能的：

```
// 原始的、最通用的版本
template <class T> struct remove_reference {
    typedef T type;
};
// 部分特例化版本，将用于左值引用和右值引用
template <class T> struct remove_reference<T&> // 左值引用
    { typedef T type; };
template <class T> struct remove_reference<T&&> // 右值引用
    { typedef T type; };
```

第一个模板定义了最通用的模板。它可以用任意类型实例化；它将模板实参作为 type 成员的类型。接下来的两个类是原始模板的部分特例化版本。

由于一个部分特例化版本本质是一个模板，与往常一样，我们首先定义模板参数。类似任何其他特例化版本，部分特例化版本的名字与原模板的名字相同。对每个未完全确定类型的模板参数，在特例化版本的模板参数列表中都有一项与之对应。在类名之后，我们为要特例化的模板参数指定实参，这些实参列于模板名之后的尖括号中。这些实参与原始模板中的参数按位置对应。

部分特例化版本的模板参数列表是原始模板的参数列表的一个子集或者是一个特例化版本。在本例中，特例化版本的模板参数的数目与原始模板相同，但是类型不同。两个特例化版本分别用于左值引用和右值引用类型：

```
int i;
// decltype(42)为 int，使用原始模板
remove_reference<decltype(42)>::type a;
```

```
// decltype(i)为int&，使用第一个（T&）部分特例化版本
remove_reference<decltype(i)>::type b;
// decltype(std::move(i))为int&&，使用第二个（即T&&）部分特例化版本
remove_reference<decltype(std::move(i))>::type c;
```

三个变量a、b和c均为int类型。

特例化成员而不是类

我们可以只特例化特定成员函数而不是特例化整个模板。例如，如果Foo是一个模板类，包含一个成员Bar，我们可以只特例化该成员：

712

```
template <typename T> struct Foo {
    Foo(const T &t = T()): mem(t) { }
    void Bar() { /* ... */ }
    T mem;
    // Foo的其他成员
};
template<>              // 我们正在特例化一个模板
void Foo<int>::Bar() // 我们正在特例化Foo<int>的成员Bar
{
    // 进行应用于int的特例化处理
}
```

本例中我们只特例化Foo<int>类的一个成员，其他成员将由Foo模板提供：

```
Foo<string> fs;         // 实例化Foo<string>::Foo()
fs.Bar();               // 实例化Foo<string>::Bar()
Foo<int> fi;            // 实例化Foo<int>::Foo()
fi.Bar();               // 使用我们特例化版本的Foo<int>::Bar()
```

当我们用int之外的任何类型使用Foo时，其成员像往常一样进行实例化。当我们用int使用Foo时，Bar之外的成员像往常一样进行实例化。如果我们使用Foo<int>的成员Bar，则会使用我们定义的特例化版本。

16.5 节练习

练习 16.62：定义你自己版本的hash<Sales_data>，并定义一个Sales_data对象的unordered_multiset。将多条交易记录保存到容器中，并打印其内容。

练习 16.63：定义一个函数模板，统计一个给定值在一个vector中出现的次数。测试你的函数，分别传递给它一个double的vector，一个int的vector以及一个string的vector。

练习 16.64：为上一题中的模板编写特例化版本来处理vector<const char*>。编写程序使用这个特例化版本。

练习 16.65：在16.3节（第617页）中我们定义了两个重载的debug_rep版本，一个接受const char*参数，另一个接受char*参数。将这两个函数重写为特例化版本。

练习 16.66：重载debug_rep函数与特例化它相比，有何优点和缺点？

练习 16.67：定义特例化版本会影响debug_rep的函数匹配吗？如果不影响，为什么？

713 小结

模板是C++语言与众不同的特性，也是标准库的基础。一个模板就是一个编译器用来生成特定类类型或函数的蓝图。生成特定类或函数的过程称为实例化。我们只编写一次模板，就可以将其用于多种类型和值，编译器会为每种类型和值进行模板实例化。

我们既可以定义函数模板，也可以定义类模板。标准库算法都是函数模板，标准库容器都是类模板。

显式模板实参允许我们固定一个或多个模板参数的类型或值。对于指定了显式模板实参的模板参数，可以应用正常的类型转换。

一个模板特例化就是一个用户提供的模板实例，它将一个或多个模板参数绑定到特定类型或值上。当我们不能（或不希望）将模板定义用于某些特定类型时，特例化非常有用。

最新C++标准的一个主要部分是可变参数模板。一个可变参数模板可以接受数目和类型可变的参数。可变参数模板允许我们编写像容器的 `emplace` 成员和标准库 `make_shared` 函数这样的函数，实现将实参传递给对象的构造函数。

术语表

类模板（class template） 模板定义，可从它实例化出特定的类。类模板的定义以关键字 `template` 开始，后跟尖括号对<和>，其内为一个用逗号分隔的一个或多个模板参数的列表，随后是类的定义。

默认模板实参（default template argument） 一个类型或一个值，当用户未提供对应模板实参时，模板会使用它。

显式实例化（explicit instantiation） 一个声明，为所有模板参数提供了显式实参。用来指导实例化过程。如果声明是 `extern` 的，模板将不会被实例化；否则，模板将利用指定的实参进行实例化。对每个 `extern` 模板声明，在程序中某处必须有一个非 `extern` 的显式实例化。

显式模板实参（explicit template argument） 在一个函数调用中或定义模板类类型时，由用户提供的模板实参。显式模板实参在紧跟在模板名的尖括号对中给出。

函数参数包（function parameter pack） 表示零个或多个函数参数的参数包。

函数模板（function template） 模板定义，可从它实例化出特定函数。函数模板的定义以关键字 `template` 开始，后跟尖括号对<和>，其内为一个用逗号分隔的一个或多个模板参数的列表，随后是函数的定义。

实例化（instantiate） 编译器处理过程，用实际的模板实参来生成模板的一个特殊实例，其中参数被替换为对应的实参。当函数模板被调用时，会自动根据传递给它的实参来实例化。而使用类模板时，则需要我们提供显式模板实参。

714 **实例**（instantiation） 编译器从模板生成的类或函数。

成员模板（member template） 本身是模板的成员函数。成员模板不能是虚函数。

非类型参数（nontype parameter） 表示值的模板参数。非类型模板参数的实参必须是常量表达式。

包扩展（pack expansion） 处理过程，将一个参数包替换为其中元素的列表。

参数包（parameter pack） 表示零个或多个参数的模板或函数参数。

部分特例化（partial specialization） 类模板的一个版本，其中指定了某些但不是所

有模板参数，或是一个或多个参数的属性未被完全指定。

模式（pattern） 定义了扩展后参数包中每个元素的形式。

模板实参（template argument） 用来实例化模板参数的类型或值。

模板实参推断（template argument deduction） 编译器确定实例化哪个函数模板的过程。编译器检查那些使用模板参数的实参的类型，将这些类型或值绑定到模板参数，来自动实例化一个函数版本。

模板参数（template parameter） 在模板参数列表中指定的名字，可在模板定义内部使用。模板参数可以是类型参数，也可以是非类型参数。为了使用一个类模板，我们必须为每个模板参数提供显式实参。编译器使用这些类型或值实例化出一个类版本，其中所有用到模板参数的地方都被替换为实际的实参。当使用一个函数模板时，编译器使用调用中的函数实参推断模板实参，并使用推断出的模板实参实例化出一个特定的函数。

模板参数列表（template parameter list） 用逗号分隔的参数列表，用于模板的定义或声明中。每个参数可以是一个类型参数，也可以是一个非类型参数。

模板参数包（template parameter pack） 表示零个或多个模板参数的参数包。

模板特例化（template specialization） 类模板、类模板的成员或函数模板的重定义，其中指定了某些（或全部）模板参数。模板特例化版本必须出现在原模板的声明之后，必须出现在任何利用特殊实参来使用模板的代码之前。一个函数模板中的每个模板参数都必须完全特例化。

类型参数（type parameter） 模板参数列表中的名字，用来表示类型。类型参数在关键字 `typename` 或 `class` 之后指定。

类型转换（type transformation） 由标准库定义的类模板，可将给定的模板类型参数转换为一个相关类型。

可变参数模板（variadic template） 接受可变数目模板实参的模板。模板参数包用省略号指定（如 `class...`、`typename...`或 *`type-name...`*）

第Ⅳ部分
高级主题

内容

第 17 章　标准库特殊设施 635
第 18 章　用于大型程序的工具 683
第 19 章　特殊工具与技术 725

第Ⅳ部分将介绍 C++和标准库的一些附加特性，虽然这些特性在特定的情况下很有用，但并非每个 C++程序员都需要它们。这些特性分为两类：一类对于求解大规模的问题很有用；另一类适用于特殊问题而非通用问题。针对特殊问题的特性既有属于 C++语言的（将在第 19 章介绍），也有属于标准库的（将在第 17 章进行介绍）。

在第 17 章中我们介绍四个具有特殊目的的标准库设施：`bitset` 类和三个新标准库设施（`tuple`、正则表达式和随机数）。我们还将介绍 IO 库中某些不常用的部分。

第 18 章介绍异常处理、命名空间和多重继承。这些特性在设计大型程序时是最有用的。

即使是一个程序员就能编写的足够简单的程序，也能从异常处理机制受益，这也是为什么我们在第 5 章介绍了异常处理的基本知识的原因。但是，对于需要大型团队才能完成的程序设计问题，运行时错误处理才显得更为重要也更难于管理。在第 18 章中，我们会额外介绍一些有用的异常处理设施。我们还将详细讨论异常是如何处理的，并展示如何定义和使用自己的异常类。这一章还会介绍新标准中异常处理方面的改进——如何指出一个特定函数不会抛出异常。

大型应用程序通常会使用来自多个提供商的代码。如果提供商不得不将他们定义的名字放置在单一的命名空间中，那么将多个独立开发的库组合起来是很困难的（如果能组合的话）。独立开发的库几乎必然会使用与其他库相同的名字；对于某个库中定义的名字，如果另一个库中使用了相同的名字，就会引起冲突。为了避免名字冲突，我们可以在一个 `namespace` 中定义名字。

无论何时我们使用一个来自标准库的名字，实际上都是在使用名为 std 的命名空间中的名字。第 18 章将会展示如何定义我们自己的命名空间。

第 18 章最后介绍一个很重要但不太常用的语言特性：多重继承。多重继承对非常复杂的继承层次很有用。

第 19 章介绍几种用于特定类别问题的特殊工具和技术，包括如何重定义内存分配机制；C++对运行时类型识别（run-time type identification，RTTI）的支持——允许我们在运行时才确定一个表达式的实际类型；以及如何定义和使用指向类成员的指针。类成员指针不同于普通数据或函数指针。普通指针仅根据对象或函数的类型而变化，而类成员指针还必须反映成员所属的类。我们还将介绍三种附加的聚合类型：联合、嵌套类和局部类。这一章最后将简要介绍一组本质上不可移植的语言特性：volatile 修饰符、位域以及链接指令。

第 17 章
标准库特殊设施

内容

17.1 tuple 类型 636
17.2 bitset 类型 640
17.3 正则表达式 645
17.4 随机数 659
17.5 IO 库再探 666
小结 680
术语表 680

最新的 C++标准极大地扩充了标准库的规模和范围。实际上，从 1998 年的第一版标准到 2011 年的最新标准，标准库部分的篇幅增加了两倍以上。因此，介绍所有 C++标准库类的知识大大超出了本书范围。但是，有 4 个标准库设施，虽然它们比我们已经介绍的其他标准库设施更特殊，但也足够通用，应该放在一本入门书籍中进行介绍。这 4 个标准库设施是：`tuple`、`bitset`、随机数生成及正则表达式。此外，我们还将介绍 IO 库中一些具有特殊目的的部分。

718 标准库占据了新标准文本将近三分之二的篇幅。虽然我们不能详细介绍所有标准库设施，但仍有一些标准库设施在很多应用中都是有用的：tuple、bitset、正则表达式以及随机数。我们还将介绍一些附加的 IO 库功能：格式控制、未格式化 IO 和随机访问。

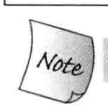
17.1 tuple 类型

C++ 11 **tuple** 是类似 pair（参见 11.2.3 节，第 379 页）的模板。每个 pair 的成员类型都不相同，但每个 pair 都恰好有两个成员。不同 tuple 类型的成员类型也不相同，但一个 tuple 可以有任意数量的成员。每个确定的 tuple 类型的成员数目是固定的，但一个 tuple 类型的成员数目可以与另一个 tuple 类型不同。

当我们希望将一些数据组合成单一对象，但又不想麻烦地定义一个新数据结构来表示这些数据时，tuple 是非常有用的。表 17.1 列出了 tuple 支持的操作。tuple 类型及其伴随类型和函数都定义在 tuple 头文件中。

表 17.1：tuple 支持的操作

`tuple<T1, T2, ..., Tn> t;`	t 是一个 tuple，成员数为 n，第 i 个成员的类型为 Ti。所有成员都进行值初始化（参见 3.3.1 节，第 88 页）
`tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);`	t 是一个 tuple，成员类型为 T1...Tn，每个成员用对应的初始值 v_i 进行初始化。此构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
`make_tuple(v1, v2, ..., vn)`	返回一个用给定初始值初始化的 tuple。tuple 的类型从初始值的类型推断
`t1 == t2` `t1 != t2`	当两个 tuple 具有相同数量的成员且成员对应相等时，两个 tuple 相等。这两个操作使用成员的==运算符来完成。一旦发现某对成员不等，接下来的成员就不用比较了
`t1` *relop* `t2`	tuple 的关系运算使用字典序（参见 9.2.7 节，第 304 页）。两个 tuple 必须具有相同数量的成员。使用<运算符比较 t1 的成员和 t2 中的对应成员
`get<i>(t)`	返回 t 的第 i 个数据成员的引用；如果 t 是一个左值，结果是一个左值引用；否则，结果是一个右值引用。tuple 的所有成员都是 public 的
`tuple_size<`*tupleType*`>::value`	一个类模板，可以通过一个 tuple 类型来初始化。它有一个名为 value 的 public constexpr static 数据成员，类型为 size_t，表示给定 tuple 类型中成员的数量
`tuple_element<i, tupleType>::type`	一个类模板，可以通过一个整型常量和一个 tuple 类型来初始化。它有一个名为 type 的 public 成员，表示给定 tuple 类型中指定成员的类型



我们可以将 tuple 看作一个“快速而随意”的数据结构。

17.1.1 定义和初始化 tuple

当我们定义一个 tuple 时，需要指出每个成员的类型：

```
tuple<size_t, size_t, size_t> threeD; // 三个成员都设置为 0
tuple<string, vector<double>, int, list<int>>
    someVal("constants", {3.14, 2.718}, 42, {0,1,2,3,4,5})
```

当我们创建一个 tuple 对象时，可以使用 tuple 的默认构造函数，它会对每个成员进行值初始化（参见 3.3.1 节，第 88 页）；也可以像本例中初始化 someVal 一样，为每个成员提供一个初始值。tuple 的这个构造函数是 explicit 的（参见 7.5.4 节，第 265 页），因此我们必须使用直接初始化语法：

```
tuple<size_t, size_t, size_t> threeD = {1,2,3};   // 错误
tuple<size_t, size_t, size_t> threeD{1,2,3};      // 正确
```

类似 make_pair 函数（参见 11.2.3 节，第 381 页），标准库定义了 make_tuple 函数，我们还可以用它来生成 tuple 对象：

```
// 表示书店交易记录的 tuple，包含：ISBN、数量和每册书的价格
auto item = make_tuple("0-999-78345-X", 3, 20.00);
```

类似 make_pair，make_tuple 函数使用初始值的类型来推断 tuple 的类型。在本例中，item 是一个 tuple，类型为 tuple<const char*, int, double>。

719

访问 tuple 的成员

一个 pair 总是有两个成员，这样，标准库就可以为它们命名（如，first 和 second）。但这种命名方式对 tuple 是不可能的，因为一个 tuple 类型的成员数目是没有限制的。因此，tuple 的成员都是未命名的。要访问一个 tuple 的成员，就要使用一个名为 **get** 的标准库函数模板。为了使用 get，我们必须指定一个显式模板实参（参见 16.2.2 节，第 603 页），它指出我们想要访问第几个成员。我们传递给 get 一个 tuple 对象，它返回指定成员的引用：

```
auto book = get<0>(item);          // 返回 item 的第一个成员
auto cnt = get<1>(item);           // 返回 item 的第二个成员
auto price = get<2>(item)/cnt;     // 返回 item 的最后一个成员
get<2>(item) *= 0.8;               // 打折 20%
```

尖括号中的值必须是一个整型常量表达式（参见 2.4.4 节，第 58 页）。与往常一样，我们从 0 开始计数，意味着 get<0>是第一个成员。

如果不知道一个 tuple 准确的类型细节信息，可以用两个辅助类模板来查询 tuple 成员的数量和类型：

720

```
typedef decltype(item) trans; // trans 是 item 的类型
// 返回 trans 类型对象中成员的数量
size_t sz = tuple_size<trans>::value; // 返回 3
// cnt 的类型与 item 中第二个成员相同
tuple_element<1, trans>::type cnt = get<1>(item); // cnt 是一个 int
```

为了使用 tuple_size 或 tuple_element，我们需要知道一个 tuple 对象的类型。与往常一样，确定一个对象的类型的最简单方法就是使用 decltype（参见 2.5.3 节，第 62 页）。在本例中，我们使用 decltype 来为 item 类型定义一个类型别名，用它来实例化

两个模板。

tuple_size 有一个名为 value 的 public static 数据成员，它表示给定 tuple 中成员的数量。tuple_element 模板除了一个 tuple 类型外，还接受一个索引值。它有一个名为 type 的 public 类型成员，表示给定 tuple 类型中指定成员的类型。类似 get，tuple_element 所使用的索引也是从 0 开始计数的。

关系和相等运算符

tuple 的关系和相等运算符的行为类似容器的对应操作（参见 9.2.7 节，第 304 页）。这些运算符逐对比较左侧 tuple 和右侧 tuple 的成员。只有两个 tuple 具有相同数量的成员时，我们才可以比较它们。而且，为了使用 tuple 的相等或不等运算符，对每对成员使用==运算符必须都是合法的；为了使用关系运算符，对每对成员使用<必须都是合法的。例如：

```
tuple<string, string> duo("1", "2");
tuple<size_t, size_t> twoD(1, 2);
bool b = (duo == twoD);   // 错误：不能比较 size_t 和 string
tuple<size_t, size_t, size_t> threeD(1, 2, 3);
b = (twoD < threeD);      // 错误：成员数量不同
tuple<size_t, size_t> origin(0, 0);
b = (origin < twoD);      // 正确：b 为 true
```



由于 tuple 定义了<和==运算符，我们可以将 tuple 序列传递给算法，并且可以在无序容器中将 tuple 作为关键字类型。

17.1.1 节练习

练习 17.1：定义一个保存三个 int 值的 tuple，并将其成员分别初始化为 10、20 和 30。

练习 17.2：定义一个 tuple，保存一个 string、一个 vector<string>和一个 pair<string, int>。

练习 17.3：重写 12.3 节（第 430 页）中的 TextQuery 程序，使用 tuple 代替 QueryResult 类。你认为哪种设计更好？为什么？

721 17.1.2 使用 tuple 返回多个值

tuple 的一个常见用途是从一个函数返回多个值。例如，我们的书店可能是多家连锁书店中的一家。每家书店都有一个销售记录文件，保存每本书近期的销售数据。我们可能希望在所有书店中查询某本书的销售情况。

假定每家书店都有一个销售记录文件。每个文件都将每本书的所有销售记录存放在一起。进一步假定已有一个函数可以读取这些销售记录文件，为每个书店创建一个 vector<Sales_data>，并将这些 vector 保存在 vector 的 vector 中：

```
// files 中的每个元素保存一家书店的销售记录
vector<vector<Sales_data>> files;
```

我们将编写一个函数，对于一本给定的书，在 files 中搜索出售过这本书的书店。对每家有匹配销售记录的书店，我们将创建一个 tuple 来保存这家书店的索引和两个迭代器。

索引指出了书店在 files 中的位置，而两个迭代器则标记了给定书籍在此书店的 vector<Sales_data>中第一条销售记录和最后一条销售记录之后的位置。

返回 tuple 的函数

我们首先编写查找给定书籍的函数。此函数的参数是刚刚提到的 vector 的 vector 以及一个表示书籍 ISBN 的 string。我们的函数将返回一个 tuple 的 vector，凡是销售了给定书籍的书店，都在 vector 中有对应的一项：

```
// matches 有三个成员：一家书店的索引和两个指向书店 vector 中元素的迭代器
typedef tuple<vector<Sales_data>::size_type,
              vector<Sales_data>::const_iterator,
              vector<Sales_data>::const_iterator> matches;
// files 保存每家书店的销售记录
// findBook 返回一个 vector，每家销售了给定书籍的书店在其中都有一项
vector<matches>
findBook(const vector<vector<Sales_data>> &files,
         const string &book)
{
    vector<matches> ret; // 初始化为空 vector
    // 对每家书店，查找与给定书籍匹配的记录范围（如果存在的话）
    for (auto it = files.cbegin(); it != files.cend(); ++it) {
        // 查找具有相同 ISBN 的 Sales_data 范围
        auto found = equal_range(it->cbegin(), it->cend(),
                                 book, compareIsbn);
        if (found.first != found.second) // 此书店销售了给定书籍
            // 记住此书店的索引及匹配的范围
            ret.push_back(make_tuple(it - files.cbegin(),
                                     found.first, found.second));
    }
    return ret; // 如果未找到匹配记录的话，ret 为空
}
```

for 循环遍历 files 中的元素，每个元素都是一个 vector。在 for 循环内，我们调用了一个名为 equal_range 的标准库算法，它的功能与关联容器的同名成员类似（参见 11.3.5 节，第 390 页）。equal_range 的前两个实参是表示输入序列的迭代器（参见 10.1 节，第 336 页），第三个参数是一个值。默认情况下，equal_range 使用<运算符来比较元素。由于 Sales_data 没有<运算符，因此我们传递给它一个指向 compareIsbn 函数的指针（参见 11.2.2 节，第 379 页）。 <722

equal_range 算法返回一个迭代器 pair，表示元素的范围。如果未找到 book，则两个迭代器相等，表示空范围。否则，返回的 pair 的 first 成员将表示第一条匹配的记录，second 则表示匹配的尾后位置。

使用函数返回的 tuple

一旦我们创建了 vector 保存包含匹配的销售记录的书店，就需要处理这些记录了。在此程序中，对每家包含匹配销售记录的书店，我们将打印其汇总销售信息：

```
void reportResults(istream &in, ostream &os,
                   const vector<vector<Sales_data>> &files)
{
    string s; // 要查找的书
```

```
    while (in >> s) {
        auto trans = findBook(files, s); // 销售了这本书的书店
        if (trans.empty()) {
            cout << s << " not found in any stores" << endl;
            continue; // 获得下一本要查找的书
        }
        for (const auto &store : trans) // 对每家销售了给定书籍的书店
            // get<n>返回 store 中 tuple 的指定的成员
            os << "store " << get<0>(store) << " sales: "
               << accumulate(get<1>(store), get<2>(store),
                             Sales_data(s))
               << endl;
    }
}
```

while 循环反复读取名为 in 的 istream 来获得下一本要处理的书。我们调用 findBook 来检查 s 是否存在，并将结果赋予 trans。我们使用 auto 来简化 trans 类型的代码编写，它是一个 tuple 的 vector。

如果 trans 为空，表示没有关于 s 的销售记录。在此情况下，我们打印一条信息并返回，执行下一步 while 循环来获取下一本要查找的书。

for 循环将 store 绑定到 trans 中的每个元素。由于不希望改变 trans 中的元素，我们将 store 声明为 const 的引用。我们使用 get 来打印相关数据：get<0>表示对应书店的索引、get<1>表示第一条交易记录的迭代器、get<2>表示尾后位置的迭代器。

由于 Sales_data 定义了加法运算符（参见 14.3 节，第 497 页），因此我们可以用标准库的 accumulate 算法（参见 10.2.1 节，第 338 页）来累加销售记录。我们用 Sales_data 的接受一个 string 参数的构造函数（参见 7.1.4 节，第 236 页）来初始化一个 Sales_data 对象，将此对象传递给 accumulate 作为求和的起点。此构造函数用给定的 string 初始化 bookNo，并将 units_sold 和 revenue 成员置为 0。

723

17.1.2 节练习

练习 17.4：编写并测试你自己版本的 findBook 函数。

练习 17.5：重写 findBook，令其返回一个 pair，包含一个索引和一个迭代器 pair。

练习 17.6：重写 findBook，不使用 tuple 或 pair。

练习 17.7：解释你更倾向于哪个版本的 findBook，为什么。

练习 17.8：在本节最后一段代码中，如果我们将 Sales_data() 作为第三个参数传递给 accumulate，会发生什么？

17.2 bitset 类型

在 4.8 节（第 135 页）中我们介绍了将整型运算对象当作二进制位集合处理的一些内置运算符。标准库还定义了 **bitset** 类，使得位运算的使用更为容易，并且能够处理超过最长整型类型大小的位集合。bitset 类定义在头文件 bitset 中。

17.2.1 定义和初始化 bitset

表 17.2 列出了 bitset 的构造函数。bitset 类是一个类模板，它类似 array 类，具有固定的大小（参见 9.2.4 节，第 301 页）。当我们定义一个 bitset 时，需要声明它包含多少个二进制位：

```
bitset<32> bitvec(1U); // 32 位；低位为 1，其他位为 0
```

大小必须是一个常量表达式（参见 2.4.4 节，第 58 页）。这条语句定义 bitvec 为一个包含 32 位的 bitset。就像 vector 包含未命名的元素一样，bitset 中的二进制位也是未命名的，我们通过位置来访问它们。二进制位的位置是从 0 开始编号的。因此，bitvec 包含编号从 0 到 31 的 32 个二进制位。编号从 0 开始的二进制位被称为**低位**（low-order），编号到 31 结束的二进制位被称为**高位**（high-order）。

表 17.2：初始化 bitset 的方法

`bitset<n> b;`	b 有 n 位；每一位均为 0。此构造函数是一个 constexpr（参见 7.5.6 节，第 267 页）
`bitset<n> b(u);`	b 是 unsigned long long 值 u 的低 n 位的拷贝。如果 n 大于 unsigned long long 的大小，则 b 中超出 unsigned long long 的高位被置为 0。此构造函数是一个 constexpr（参见 7.5.6 节，第 267 页）
`bitset<n> b(s, pos, m, zero, one);`	b 是 string s 从位置 pos 开始 m 个字符的拷贝。s 只能包含字符 zero 或 one；如果 s 包含任何其他字符，构造函数会抛出 invalid_argument 异常。字符在 b 中分别保存为 zero 和 one。pos 默认为 0，m 默认为 string::npos，zero 默认为'0'，one 默认为'1'
`bitset<n> b(cp, pos, m, zero, one);`	与上一个构造函数相同，但从 cp 指向的字符数组中拷贝字符。如果未提供 m，则 cp 必须指向一个 C 风格字符串。如果提供了 m，则从 cp 开始必须至少有 m 个 zero 或 one 字符

接受一个 string 或一个字符指针的构造函数是 explicit 的（参见 7.5.4 节，第 265 页）。在新标准中增加了为 0 和 1 指定其他字符的功能。

用 unsigned 值初始化 bitset

当我们使用一个整型值来初始化 bitset 时，此值将被转换为 unsigned long long 类型并被当作位模式来处理。bitset 中的二进制位将是此模式的一个副本。如果 bitset 的大小大于一个 unsigned long long 中的二进制位数，则剩余的高位被置为 0。如果 bitset 的大小小于一个 unsigned long long 中的二进制位数，则只使用给定值中的低位，超出 bitset 大小的高位被丢弃：

724

```
// bitvec1 比初始值小；初始值中的高位被丢弃
bitset<13> bitvec1(0xbeef);   // 二进制位序列为 1111011101111
// bitvec2 比初始值大；它的高位被置为 0
bitset<20> bitvec2(0xbeef);   // 二进制位序列为 00001011111011101111
// 在 64 位机器中，long long 0ULL 是 64 个 0 比特，因此~0ULL 是 64 个 1
bitset<128> bitvec3(~0ULL);   // 0~63 位为 1；63~127 位为 0
```

从一个 string 初始化 bitset

我们可以从一个 string 或一个字符数组指针来初始化 bitset。两种情况下，字符都直接表示位模式。与往常一样，当我们使用字符串表示数时，字符串中下标最小的字符对应高位，反之亦然：

```
bitset<32> bitvec4("1100"); // 2、3 两位为 1，剩余两位为 0
```

如果 string 包含的字符数比 bitset 少，则 bitset 的高位被置为 0。



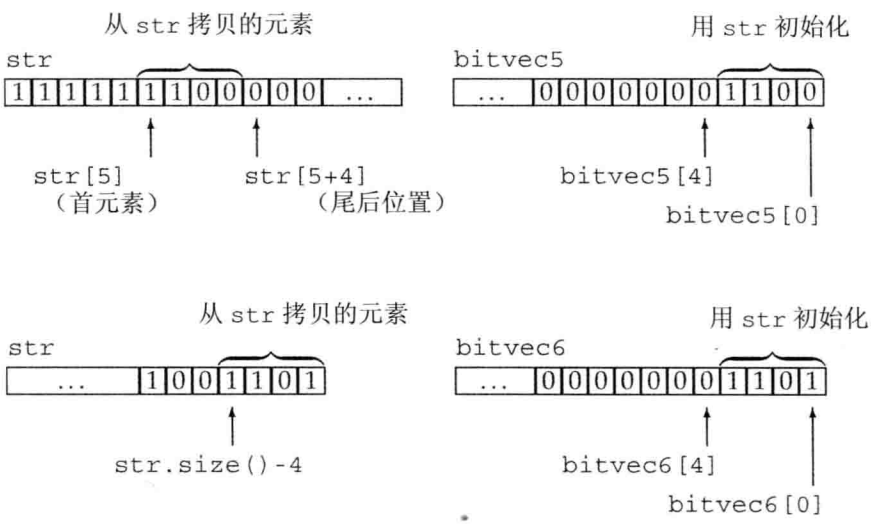
string 的下标编号习惯与 bitset 恰好相反：string 中下标最大的字符（最右字符）用来初始化 bitset 中的低位（下标为 0 的二进制位）。当你用一个 string 初始化一个 bitset 时，要记住这个差别。

725

我们不必使用整个 string 来作为 bitset 的初始值，可以只用一个子串作为初始值：

```
string str("11111110000000011001101");
bitset<32> bitvec5(str, 5, 4); // 从 str[5]开始的四个二进制位，1100
bitset<32> bitvec6(str, str.size()-4); // 使用最后四个字符
```

此处，bitvec5 用 str 中从 str[5]开始的长度为 4 的子串进行初始化。与往常一样，子串的最右字符表示最低位。因此，bitvec5 中第 3 位到第 0 位被设置为 1100，剩余位被设置为 0。传递给 bitvec6 的初始值是一个 string 和一个开始位置，因此 bitvec6 用 str 中倒数第四个字符开始的子串进行初始化。bitvec6 中剩余二进制位被初始化为 0。下图说明了这两个初始化过程



17.2.1 节练习

练习 17.9：解释下列每个 bitset 对象所包含的位模式：

(a) bitset<64> bitvec(32);
(b) bitset<32> bv(1010101);
(c) string bstr; cin >> bstr; bitset<8>bv(bstr);

17.2.2 bitset 操作

bitset 操作（参见表 17.3）定义了多种检测或设置一个或多个二进制位的方法。bitset 类还支持我们在 4.8 节(第 136 页)中介绍过的位运算符。这些运算符用于 bitset 对象的含义与内置运算符用于 unsigned 运算对象相同。

表 17.3：bitset 操作 726

b.any()	b 中是否存在置位的二进制位
b.all()	b 中所有位都置位了吗
b.none()	b 中不存在置位的二进制位吗
b.count()	b 中置位的位数
b.size()	一个 constexpr 函数（参见 2.4.4 节，第 58 页），返回 b 中的位数
b.test(pos)	若 pos 位置的位是置位的，则返回 true，否则返回 false
b.set(pos,v) b.set()	将位置 pos 处的位设置为 bool 值 v。v 默认为 true。如果未传递实参，则将 b 中所有位置位
b.reset(pos) b.reset()	将位置 pos 处的位复位或将 b 中所有位复位
b.flip(pos) b.flip()	改变位置 pos 处的位的状态或改变 b 中每一位的状态
b[pos]	访问 b 中位置 pos 处的位；如果 b 是 const 的，则当该位置位时 b[pos]返回一个 bool 值 true，否则返回 false
b.to_ulong() b.to_ullong()	返回一个 unsigned long 或一个 unsigned long long 值，其位模式与 b 相同。如果 b 中位模式不能放入指定的结果类型，则抛出一个 overflow_error 异常
b.to_string(zero, one)	返回一个 string，表示 b 中的位模式。zero 和 one 的默认值分别为 0 和 1，用来表示 b 中的 0 和 1
os << b	将 b 中二进制位打印为字符 1 或 0，打印到流 os
is >> b	从 is 读取字符存入 b。当下一个字符不是 1 或 0 时，或是已经读入 b.size()个位时，读取过程停止

count、size、all、any 和 none 等几个操作都不接受参数，返回整个 bitset 的状态。其他操作——set、reset 和 flip 则改变 bitset 的状态。改变 bitset 状态的成员函数都是重载的。对每个函数，不接受参数的版本对整个集合执行给定的操作；接受一个位置参数的版本则对指定位执行操作：

```
bitset<32> bitvec(1U);                  // 32 位；低位为 1，剩余位为 0
bool is_set = bitvec.any();             // true，因为有 1 位置位
bool is_not_set = bitvec.none();        // false，因为有 1 位置位了
bool all_set = bitvec.all();            // false，因为只有 1 位置位
size_t onBits = bitvec.count();         // 返回 1
size_t sz = bitvec.size();              // 返回 32
bitvec.flip();    // 翻转 bitvec 中的所有位
bitvec.reset();   // 将所有位复位
bitvec.set();     // 将所有位置位
```

当 bitset 对象的一个或多个位置位（即，等于 1）时，操作 any 返回 true。相反，当所有位复位时，none 返回 true。新标准引入了 all 操作，当所有位置位时返回 true。 C++11

727 操作 count 和 size 返回 size_t 类型的值（参见 3.5.2 节，第 103 页），分别表示对象中置位的位数或总位数。函数 size 是一个 constexpr 函数，因此可以用在要求常量表达式的地方（参见 2.4.4 节，第 58 页）。

成员 flip、set、reset 及 test 允许我们读写指定位置的位：

```
bitvec.flip(0);       // 翻转第一位
bitvec.set(bitvec.size() - 1); // 置位最后一位
bitvec.set(0, 0);     // 复位第一位
bitvec.reset(i);      // 复位第 i 位
bitvec.test(0);       // 返回 false，因为第一位是复位的
```

下标运算符对 const 属性进行了重载。const 版本的下标运算符在指定位置位时返回 true，否则返回 false。非 const 版本返回 bitset 定义的一个特殊类型，它允许我们操纵指定位的值：

```
bitvec[0] = 0;            // 将第一位复位
bitvec[31] = bitvec[0];   // 将最后一位设置为与第一位一样
bitvec[0].flip();         // 翻转第一位
~bitvec[0];               // 等价操作，也是翻转第一位
bool b = bitvec[0];       // 将 bitvec[0]的值转换为 bool 类型
```

提取 bitset 的值

to_ulong 和 to_ullong 操作都返回一个值，保存了与 bitset 对象相同的位模式。只有当bitset 的大小小于等于对应的大小（to_ulong 为 unsigned long，to_ullong 为 unsigned long long）时，我们才能使用这两个操作：

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```



如果 bitset 中的值不能放入给定类型中，则这两个操作会抛出一个 overflow_error 异常（参见 5.6 节，第 173 页）。

bitset 的 IO 运算符

输入运算符从一个输入流读取字符，保存到一个临时的 string 对象中。直到读取的字符数达到对应 bitset 的大小时，或是遇到不是 1 或 0 的字符时，或是遇到文件尾或输入错误时，读取过程才停止。随即用临时 string 对象来初始化 bitset（参见 17.2.1 节，第 642 页）。如果读取的字符数小于 bitset 的大小，则与往常一样，高位将被置为 0。

输出运算符打印一个 bitset 对象中的位模式：

```
bitset<16> bits;
cin >> bits; // 从 cin 读取最多 16 个 0 或 1
cout << "bits: " << bits << endl; // 打印刚刚读取的内容
```

728 ### 使用 bitset

为了说明如何使用bitset，我们重新实现4.8节（第137页）中的评分程序，用bitset代替 unsigned long 表示 30 个学生的测验结果——“通过/失败”：

```
bool status;
// 使用位运算符的版本
unsigned long quizA = 0;          // 此值被当作位集合使用
```

```
quizA |= 1UL << 27;                    // 指出第 27 个学生通过了测验
status = quizA & (1UL << 27);          // 检查第 27 个学生是否通过了测验
quizA &= ~(1UL << 27);                 // 第 27 个学生未通过测验
// 使用标准库类 bitset 完成等价的工作
bitset<30> quizB;          // 每个学生分配一位，所有位都被初始化为 0
quizB.set(27);             // 指出第 27 个学生通过了测验
status = quizB[27];        // 检查第 27 个学生是否通过了测验
quizB.reset(27);           // 第 27 个学生未通过测验
```

17.2.2 节练习

练习 17.10：使用序列 1、2、3、5、8、13、21 初始化一个 bitset，将这些位置置位。对另一个 bitset 进行默认初始化，并编写一小段程序将其恰当的位置位。

练习 17.11：定义一个数据结构，包含一个整型对象，记录一个包含 10 个问题的真/假测验的解答。如果测验包含 100 道题，你需要对数据结构做出什么改变（如果需要的话）？

练习 17.12：使用前一题中的数据结构，编写一个函数，它接受一个问题编号和一个表示真/假解答的值，函数根据这两个参数更新测验的解答。

练习 17.13：编写一个整型对象，包含真/假测验的正确答案。使用它来为前两题中的数据结构生成测验成绩。

17.3 正则表达式

正则表达式（regular expression）是一种描述字符序列的方法，是一种极其强大的计算工具。但是，用于定义正则表达式的描述语言已经大大超出了本书的范围。因此，我们重点介绍如何使用 C++正则表达式库（RE 库），它是新标准库的一部分。RE 库定义在头文件 regex 中，它包含多个组件，列于表 17.4 中。

C++ 11

表 17.4：正则表达式库组件	
regex	表示有一个正则表达式的类
regex_match	将一个字符序列与一个正则表达式匹配
regex_search	寻找第一个与正则表达式匹配的子序列
regex_replace	使用给定格式替换一个正则表达式
sregex_iterator	迭代器适配器，调用 regex_search 来遍历一个 string 中所有匹配的子串
smatch	容器类，保存在 string 中搜索的结果
ssub_match	string 中匹配的子表达式的结果



如果你还不熟悉正则表达式的使用，你应该浏览这一节，以获得正则表达式可以做什么的一些概念。

regex 类表示一个正则表达式。除了初始化和赋值之外，regex 还支持其他一些操作。表 17.6（第 647 页）列出了 regex 支持的操作。 729

函数 **regex_match** 和 **regex_search** 确定一个给定字符序列与一个给定 regex

是否匹配。如果整个输入序列与表达式匹配，则 regex_match 函数返回 true；如果输入序列中一个子串与表达式匹配，则 regex_search 函数返回 true。还有一个 regex_replace 函数，我们将在 17.3.4 节（第 657 页）中介绍。

表 17.5 列出了 regex 的函数的参数。这些函数都返回 bool 值，且都被重载了：其中一个版本接受一个类型为 **smatch** 的附加参数。如果匹配成功，这些函数将成功匹配的相关信息保存在给定的 smatch 对象中。

表 17.5：regex_search 和 regex_match 的参数

注意：这些操作返回 bool 值，指出是否找到匹配。

(*seq*, m, r, mft) (*seq*, r, mft)	在字符序列 *seq* 中查找 regex 对象 r 中的正则表达式。*seq* 可以是一个 string、表示范围的一对迭代器以及一个指向空字符结尾的字符数组的指针
	m 是一个 *match* 对象，用来保存匹配结果的相关细节。m 和 *seq* 必须具有兼容的类型（参见 17.3.1 节，第 649 页）
	mft 是一个可选的 regex_constants::match_flag_type 值。表 17.13（第 659 页）描述了这些值，它们会影响匹配过程

17.3.1 使用正则表达式库

我们从一个非常简单的例子开始——查找违反众所周知的拼写规则“i 除非在 c 之后，否则必须在 e 之前”的单词：

```
// 查找不在字符 c 之后的字符串 ei
string pattern("[^c]ei");
// 我们需要包含 pattern 的整个单词
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern);     // 构造一个用于查找模式的 regex
smatch results;       // 定义一个对象保存搜索结果
// 定义一个 string 保存与模式匹配和不匹配的文本
string test_str = "receipt freind theif receive";
// 用 r 在 test_str 中查找与 pattern 匹配的子串
if (regex_search(test_str, results, r))    // 如果有匹配子串
    cout << results.str() << endl;         // 打印匹配的单词
```

我们首先定义了一个 string 来保存希望查找的正则表达式。正则表达式[^c]表明我们希望匹配任意不是'c'的字符，而[^c]ei 指出我们想要匹配这种字符后接 ei 的字符串。此模式描述的字符串恰好包含三个字符。我们想要包含此模式的单词的完整内容。为了与整个单词匹配，我们还需要一个正则表达式与这个三字母模式之前和之后的字母匹配。

730 这个正则表达式包含零个或多个字母后接我们的三字母的模式，然后再接零个或多个额外的字母。默认情况下，regex 使用的正则表达式语言是 ECMAScript。在 ECMAScript 中，模式[[::alpha:]]匹配任意字母，符号+和*分别表示我们希望“一个或多个”或“零个或多个”匹配。因此[[::alpha:]]*将匹配零个或多个字母。

将正则表达式存入 pattern 后，我们用它来初始化一个名为 r 的 regex 对象。接下来我们定义了一个 string，用来测试正则表达式。我们将 test_str 初始化为与模式匹配的单词（如"freind"和"thief"）和不匹配的单词（如"recepit"和"receive"）。我们还定义了一个名为 results 的 smatch 对象，它将被传递给 regex_search。如果找到匹配子串，results 将会保存匹配位置的细节信息。

接下来我们调用了 regex_search。如果它找到匹配子串，就返回 true。我们用 results 的 str 成员来打印 test_str 中与模式匹配的部分。函数 regex_search 在输入序列中只要找到一个匹配子串就会停止查找。因此，程序的输出将是

```
freind
```

17.3.2 节（第 650 页）将会介绍如何查找输入序列中所有的匹配子串。

指定 regex 对象的选项

当我们定义一个 regex 或是对一个 regex 调用 assign 为其赋予新值时，可以指定一些标志来影响 regex 如何操作。这些标志控制 regex 对象的处理过程。表 17.6 列出的最后 6 个标志指出编写正则表达式所用的语言。对这 6 个标志，我们必须设置其中之一，且只能设置一个。默认情况下，ECMAScript 标志被设置，从而 regex 会使用 ECMA-262 规范，这也是很多 Web 浏览器所使用的正则表达式语言。

表 17.6：regex（和 wregex）选项

`regex r(re)` `regex r(re, f)`	*re* 表示一个正则表达式，它可以是一个 string、一个表示字符范围的迭代器对、一个指向空字符结尾的字符数组的指针、一个字符指针和一个计数器或是一个花括号包围的字符列表。f 是指出对象如何处理的标志。f 通过下面列出的值来设置。如果未指定 f，其默认值为 ECMAScript
`r1 = re`	将 r1 中的正则表达式替换为 *re*。*re* 表示一个正则表达式，它可以是另一个 regex 对象、一个 string、一个指向空字符结尾的字符数组的指针或是一个花括号包围的字符列表
`r1.assign(re, f)`	与使用赋值运算符（=）效果相同；可选的标志 f 也与 regex 的构造函数中对应的参数含义相同
`r.mark_count()`	r 中子表达式的数目（我们将在 17.3.3 节（第 654 页）中介绍）
`r.flags()`	返回 r 的标志集

注：构造函数和赋值操作可能抛出类型为 regex_error 的异常。

定义 regex 时指定的标志

定义在 regex 和 regex_constants::syntax_option_type 中

icase	在匹配过程中忽略大小写
nosubs	不保存匹配的子表达式
optimize	执行速度优先于构造速度
ECMAScript	使用 ECMA-262 指定的语法
basic	使用 POSIX 基本的正则表达式语法
extended	使用 POSIX 扩展的正则表达式语法
awk	使用 POSIX 版本的 *awk* 语言的语法
grep	使用 POSIX 版本的 grep 的语法
egrep	使用 POSIX 版本的 egrep 的语法

其他 3 个标志允许我们指定正则表达式处理过程中与语言无关的方面。例如，我们可以指出希望正则表达式以大小写无关的方式进行匹配。

作为一个例子，我们可以用 icase 标志查找具有特定扩展名的文件名。大多数操作系统都是按大小写无关的方式来识别扩展名的——可以将一个 C++程序保存在 .cc 结尾的文件中，也可以保存在 .Cc、.cC 或是 .CC 结尾的文件中，效果是一样的。如下所示，我

731

们可以编写一个正则表达式来识别上述任何一种扩展名以及其他普通文件扩展名：

```
// 一个或多个字母或数字字符后接一个'.'再接"cpp"或"cxx"或"cc"
regex r("[[:alnum:]]+\\.(cpp|cxx|cc)$", regex::icase);
smatch results;
string filename;
while (cin >> filename)
    if (regex_search(filename, results, r))
        cout << results.str() << endl; // 打印匹配结果
```

此表达式将匹配这样的字符串：一个或多个字母或数字后接一个句点再接三个文件扩展名之一。这样，此正则表达式将会匹配指定的文件扩展名而不理会大小写。

就像 C++语言中有特殊字符一样（参见 2.1.3 节，第 36 页），正则表达式语言通常也有特殊字符。例如，字符点（.）通常匹配任意字符。与 C++一样，我们可以在字符之前放置一个反斜线来去掉其特殊含义。由于反斜线也是 C++中的一个特殊字符，我们在字符串字面常量中必须连续使用两个反斜线来告诉 C++我们想要一个普通反斜线字符。因此，为了表示与句点字符匹配的正则表达式，必须写成\\.（第一个反斜线去掉 C++语言中反斜线的特殊含义，即，正则表达式字符串为\.，第二个反斜线则表示在正则表达式中去掉.的特殊含义）。

732 指定或使用正则表达式时的错误

我们可以将正则表达式本身看作用一种简单程序设计语言编写的“程序”。这种语言不是由 C++编译器解释的。正则表达式是在运行时，当一个 regex 对象被初始化或被赋予一个新模式时，才被“编译”的。与任何其他程序设计语言一样，我们用这种语言编写的正则表达式也可能有错误。



需要意识到的非常重要的一点是，一个正则表达式的语法是否正确是在运行时解析的。

如果我们编写的正则表达式存在错误，则在运行时标准库会抛出一个类型为 **regex_error** 的异常（参见 5.6 节，第 173 页）。类似标准异常类型，regex_error 有一个 what 操作来描述发生了什么错误（参见 5.6.2 节，第 175 页）。regex_error 还有一个名为 code 的成员，用来返回某个错误类型对应的数值编码。code 返回的值是由具体实现定义的。RE 库能抛出的标准错误如表 17.7 所示。

例如，我们可能在模式中意外遇到一个方括号：

```
try {
    // 错误：alnum 漏掉了右括号，构造函数会抛出异常
    regex r("[[:alnum:]+\\.(cpp|cxx|cc)$", regex::icase);
} catch (regex_error e)
  { cout << e.what() << "\ncode: " << e.code() << endl; }
```

当这段程序在我们的系统上运行时，程序会生成：

```
regex_error(error_brack):
The expression contained mismatched [ and ].
code: 4
```

表 17.7：正则表达式错误类型

定义在 regex 和 regex_constants::error_type 中	
error_collate	无效的元素校对请求
error_ctype	无效的字符类
error_escape	无效的转义字符或无效的尾置转义
error_backref	无效的向后引用
error_brack	不匹配的方括号（[或]）
error_paren	不匹配的小括号（(或)）
error_brace	不匹配的花括号（{或}）
error_badbrace	{}中无效的范围
error_range	无效的字符范围（如[z-a]）
error_space	内存不足，无法处理此正则表达式
error_badrepeat	重复字符（*、?、+或{）之前没有有效的正则表达式
error_complexity	要求的匹配过于复杂
error_stack	栈空间不足，无法处理匹配

我们的编译器定义了 code 成员，返回表 17.7 列出的错误类型的编号，与往常一样，编号从 0 开始。 733

建议：避免创建不必要的正则表达式

如我们所见，一个正则表达式所表示的“程序”是在运行时而非编译时编译的。正则表达式的编译是一个非常慢的操作，特别是在你使用了扩展的正则表达式语法或是复杂的正则表达式时。因此，构造一个 regex 对象以及向一个已存在的 regex 赋予一个新的正则表达式可能是非常耗时的。为了最小化这种开销，你应该努力避免创建很多不必要的 regex。特别是，如果你在一个循环中使用正则表达式，应该在循环外创建它，而不是在每步迭代时都编译它。

正则表达式类和输入序列类型

我们可以搜索多种类型的输入序列。输入可以是普通 char 数据或 wchar_t 数据，字符可以保存在标准库 string 中或是 char 数组中（或是宽字符版本，wstring 或 wchar_t 数组中）。RE 为这些不同的输入序列类型都定义了对应的类型。

例如，regex 类保存类型 char 的正则表达式。标准库还定义了一个 wregex 类保存类型 wachar_t，其操作与 regex 完全相同。两者唯一的差别是 wregex 的初始值必须使用 wchar_t 而不是 char。

匹配和迭代器类型（我们将在下面小节中介绍）更为特殊。这些类型的差异不仅在于字符类型，还在于序列是在标准库 string 中还是在数组中：smatch 表示 string 类型的输入序列；cmatch 表示字符数组序列；wsmatch 表示宽字符串（wstring）输入；而 wcmatch 表示宽字符数组。

重点在于我们使用的 RE 库类型必须与输入序列类型匹配。表 17.8 指出了 RE 库类型与输入序列类型的对应关系。例如：

```
regex r("[[:alnum:]]+\\.(cpp|cxx|cc)$", regex::icase);
smatch results; // 将匹配 string 输入序列，而不是 char*
if (regex_search("myfile.cc", results, r)) // 错误：输入为 char*
    cout << results.str() << endl;
```

734 这段代码会编译失败，因为 match 参数的类型与输入序列的类型不匹配。如果我们希望搜索一个字符数组，就必须使用 cmatch 对象：

```
cmatch results; // 将匹配字符数组输入序列
if (regex_search("myfile.cc", results, r))
    cout << results.str() << endl; // 打印当前匹配
```

本书程序一般会使用 string 输入序列和对应的 string 版本的 RE 库组件。

表 17.8：正则表达式库类

如果输入序列类型	则使用正则表达式类
string	regex、smatch、ssub_match 和 sregex_iterator
const char*	regex、cmatch、csub_match 和 cregex_iterator
wstring	wregex、wsmatch、wssub_match 和 wsregex_iterator
const wchar_t*	wregex、wcmatch、wcsub_match 和 wcregex_iterator

17.3.1 节练习

练习 17.14：编写几个正则表达式，分别触发不同错误。运行你的程序，观察编译器对每个错误的输出。

练习 17.15：编写程序，使用模式查找违反“i 在 e 之前，除非在 c 之后”规则的单词。你的程序应该提示用户输入一个单词，然后指出此单词是否符合要求。用一些违反和未违反规则的单词测试你的程序。

练习 17.16：如果前一题程序中的 regex 对象用"[^c]ei"进行初始化，将会发生什么？用此模式测试你的程序，检查你的答案是否正确。

17.3.2 匹配与 Regex 迭代器类型

第 646 页中的程序查找违反“i 在 e 之前，除非在 c 之后”规则的单词，它只打印输入序列中第一个匹配的单词。我们可以使用 **sregex_iterator** 来获得所有匹配。regex 迭代器是一种迭代器适配器（参见 9.6 节，第 329 页），被绑定到一个输入序列和一个 regex 对象上。如表 17.8 所述，每种不同输入序列类型都有对应的特殊 regex 迭代器类型。迭代器操作如表 17.9 所述。

表 17.9：sregex_iterator 操作

这些操作也适用于 cregex_iterator、wsregex_iterator 和 wcregex_iterator。	
sregex_iterator	一个 sregex_iterator，遍历迭代器 b 和 e 表示的 string。
it(b, e, r);	它调用 sregex_search(b, e, r) 将 it 定位到输入中第一个匹配的位置

续表

sregex_iterator end;	sregex_iterator 的尾后迭代器
*it it->	根据最后一个调用 regex_search 的结果，返回一个 smatch 对象的引用或一个指向 smatch 对象的指针
++it it++	从输入序列当前匹配位置开始调用 regex_search。前置版本返回递增后迭代器；后置版本返回旧值
it1 == it2 it1 != it2	如果两个 sregex_iterator 都是尾后迭代器，则它们相等两个非尾后迭代器是从相同的输入序列和 regex 对象构造，则它们相等

当我们将一个 sregex_iterator 绑定到一个 string 和一个 regex 对象时，迭代器自动定位到给定 string 中第一个匹配位置。即，sregex_iterator 构造函数对给定 string 和 regex 调用 regex_search。当我们解引用迭代器时，会得到一个对应最近一次搜索结果的 smatch 对象。当我们递增迭代器时，它调用 regex_search 在输入 string 中查找下一个匹配。

使用 sregex_iterator

作为一个例子，我们将扩展之前的程序，在一个文本文件中查找所有违反“i 在 e 之前，除非在 c 之后”规则的单词。我们假定名为 file 的 string 保存了我们要搜索的输入文件的全部内容。这个版本的程序将使用与前一个版本一样的 pattern，但会使用一个 sregex_iterator 来进行搜索：

735

```
// 查找前一个字符不是 c 的字符串 ei
string pattern("[^c]ei");
// 我们想要包含 pattern 的单词的全部内容
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase); // 在进行匹配时将忽略大小写
// 它将反复调用 regex_search 来寻找文件中的所有匹配
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
          it != end_it; ++it)
         cout << it->str() << endl; // 匹配的单词
```

for 循环遍历 file 中每个与 r 匹配的子串。for 语句中的初始值定义了 it 和 end_it。当我们定义 it 时，sregex_iterator 的构造函数调用 regex_search 将 it 定位到 file 中第一个与 r 匹配的位置。而 end_it 是一个空 sregex_iterator，起到尾后迭代器的作用。for 语句中的递增运算通过 regex_search 来“推进”迭代器。当我们解引用迭代器时，会得到一个表示当前匹配结果的 smatch 对象。我们调用它的 str 成员来打印匹配的单词。

我们可以将此循环想象为不断从一个匹配位置跳到下一个匹配位置，如图 17.1 所示。

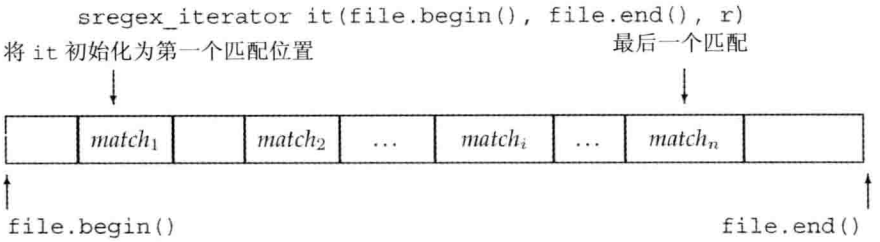


图 17.1：使用 sregex_iterator

使用匹配数据

如果我们对最初版本程序中的 test_str 运行此循环，则输出将是

```
freind
theif
```

但是，仅获得与我们的正则表达式匹配的单词还不是那么有用。如果我们在一个更大的输入序列——例如，在本章英文版的文本上运行此程序——可能希望看到匹配单词出现的上下文，如

```
hey read or write according to the type
      >>> being <<<
handled. The input operators ignore whi
```

除了允许打印输入字符串中匹配的部分之外，匹配结果类还提供了有关匹配结果的更多细节信息。表 17.10 和表 17.11 列出了这些类型支持的操作。

736

我们将在下一节中介绍更多有关 smatch 和 **ssub_match** 类型的内容。目前，我们只需知道它们允许我们获得匹配的上下文即可。匹配类型有两个名为 prefix 和 suffix 的成员，分别返回表示输入序列中当前匹配之前和之后部分的 ssub_match 对象。一个 ssub_match 对象有两个名为 str 和 length 的成员，分别返回匹配的 string 和该 string 的大小。我们可以用这些操作重写语法程序的循环。

```
// 循环头与之前一样
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
          it != end_it; ++it) {
     auto pos = it->prefix().length();          // 前缀的大小
     pos = pos > 40 ? pos - 40 : 0;             // 我们想要最多 40 个字符
     cout << it->prefix().str().substr(pos)     // 前缀的最后一部分
          << "\n\t\t>>> " << it->str() << " <<<\n" // 匹配的单词
          << it->suffix().str().substr(0, 40)       // 后缀的第一部分
          << endl;
}
```

循环本身的工作方式与前一个程序相同。改变的是循环内部，如图 17.2 所示。

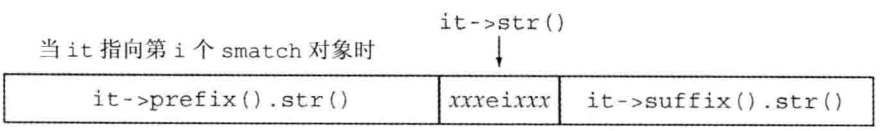


图 17.2：smatch 对象表示一个特定匹配

我们调用 prefix，返回一个 ssub_match 对象，表示 file 中当前匹配之前的部分。

我们对此 ssub_match 对象调用 length，获得前缀部分的字符数目。接下来调整 pos，使之指向前缀部分末尾向前 40 个字符的位置。如果前缀部分的长度小于 40 个字符，我们将 pos 置为 0，表示要打印整个前缀部分。我们用 substr（参见 9.5.1 节，第 321 页）来打印指定位置到前缀部分末尾的内容。

打印了当前匹配之前的字符之后，我们接下来用特殊格式打印匹配的单词本身，使得它在输出中能突出显示出来。打印匹配单词之后，我们打印 file 中匹配部分之后的前（最多）40 个字符。

737

表 17.10：smatch 操作

这些操作也适用于 cmatch、wsmatch、wcmatch 和对应的 csub_match、wssub_match 和 wcsub_match。

m.ready()	如果已经通过调用 regex_serach 或 regex_match 设置了 m，则返回 true；否则返回 false。如果 ready 返回 false，则对 m 进行操作是未定义的
m.size()	如果匹配失败，则返回 0；否则返回最近一次匹配的正则表达式中子表达式的数目
m.empty()	若 m.size() 为 0，则返回 true
m.prefix()	一个 ssub_match 对象，表示当前匹配之前的序列
m.suffix()	一个 ssub_match 对象，表示当前匹配之后的部分
m.format(…)	见表 17.12（第 657 页）

在接受一个索引的操作中，n 的默认值为 0 且必须小于 m.size()。第一个子匹配（索引为 0）表示整个匹配。

m.length(n)	第 n 个匹配的子表达式的大小
m.position(n)	第 n 个子表达式距序列开始的距离
m.str(n)	第 n 个子表达式匹配的 string
m[n]	对应第 n 个子表达式的 ssub_match 对象
m.begin(),m.end() m.cbegin(),m.cend()	表示 m 中 sub_match 元素范围的迭代器。与往常一样，cbegin 和 cend 返回 const_iterator

17.3.2 节练习

练习 17.17：更新你的程序，令它查找输入序列中所有违反“ei”语法规则的单词。

练习 17.18：修改你的程序，忽略包含“ei”但并非拼写错误的单词，如“albeit”和“neighbor”。

17.3.3 使用子表达式

738

正则表达式中的模式通常包含一个或多个**子表达式**（subexpression）。一个子表达式是模式的一部分，本身也具有意义。正则表达式语法通常用括号表示子表达式。

例如，我们用来匹配 C++文件的模式（参见 17.3.1 节，第 646 页）就是用括号来分组可能的文件扩展名。每当我们用括号分组多个可行选项时，同时也就声明了这些选项形成子表达式。我们可以重写扩展名表达式，以使得模式中点之前表示文件名的部分也形成子表达式，如下所示：

```
// r有两个子表达式：第一个是点之前表示文件名的部分，第二个表示文件扩展名
regex r("([[:alnum:]]+)\\.(cpp|cxx|cc)$", regex::icase);
```

现在我们的模式包含两个括号括起来的子表达式：

- `([[:alnum:]]+)`，匹配一个或多个字符的序列
- `(cpp|cxx|cc)`，匹配文件扩展名

我们还可以重写 17.3.1 节（第 646 页）中的程序，通过修改输出语句使之只打印文件名。

```
if (regex_search(filename, results, r))
    cout << results.str(1) << endl; // 打印第一个子表达式
```

与最初的程序一样，我们还是调用 `regex_search` 在名为 `filename` 的 `string` 中查找模式 `r`，并且传递 `smatch` 对象 `results` 来保存匹配结果。如果调用成功，我们打印结果。但是，在此版本中，我们打印的是 `str(1)`，即，与第一个子表达式匹配的部分。

匹配对象除了提供匹配整体的相关信息外，还提供访问模式中每个子表达式的能力。子匹配是按位置来访问的。第一个子匹配位置为 0，表示整个模式对应的匹配，随后是每个子表达式对应的匹配。因此，本例模式中第一个子表达式，即表示文件名的子表达式，其位置为 1，而文件扩展名对应的子表达式位置为 2。

例如，如果文件名为 `foo.cpp`，则 `results.str(0)` 将保存 `foo.cpp`；`results.str(1)` 将保存 `foo`；而 `results.str(2)` 将保存 `cpp`。在此程序中，我们想要点之前的那部分名字，即第一个子表达式，因此我们打印 `results.str(1)`。

子表达式用于数据验证

子表达式的一个常见用途是验证必须匹配特定格式的数据。例如，美国的电话号码有十位数字，包含一个区号和一个七位的本地号码。区号通常放在括号里，但这并不是必需的。剩余七位数字可以用一个短横线、一个点或是一个空格分隔，但也可以完全不用分隔符。我们可能希望接受任何这种格式的数据而拒绝任何其他格式的数。我们将分两步来实现这一目标：首先，我们将用一个正则表达式找到可能是电话号码的序列，然后再调用一个函数来完成数据验证。

739

在编写电话号码模式之前，我们需要介绍一下 ECMAScript 正则表达式语言的一些特性：

- `\{d}`表示单个数字而`\{d}{n}`则表示一个 *n* 个数字的序列。（如，`\{d}{3}`匹配三个数字的序列。）
- 在方括号中的字符集合表示匹配这些字符中任意一个。（如，`[-. ]`匹配一个短横线或一个点或一个空格。注意，点在括号中没有特殊含义。）
- 后接`'?'`的组件是可选的。（如，`\{d}{3}[-. ]?\{d}{4}`匹配这样的序列：开始是三个数字，后接一个可选的短横线或点或空格，然后是四个数字。此模式可以匹配 `555-0132` 或 `555.0132` 或 `555 0132` 或 `5550132`。）
- 类似 C++，ECMAScript 使用反斜线表示一个字符本身而不是其特殊含义。由于我们的模式包含括号，而括号是 ECMAScript 中的特殊字符，因此我们必须用`\(`和`\)`来表示括号是我们的模式的一部分而不是特殊字符。

由于反斜线是 C++中的特殊字符，在模式中每次出现`\`的地方，我们都必须用一个额外的反斜线来告知 C++我们需要一个反斜线字符而不是一个特殊符号。因此，我们用`\\{d}{3}`来表示正则表达式`\{d}{3}`。

为了验证电话号码，我们需要访问模式的组成部分。例如，我们希望验证区号部分的数字如果用了左括号，那么它是否也在区号后面用了右括号。即，我们不希望出现(908.555.1800这样的号码。

为了获得匹配的组成部分，我们需要在定义正则表达式时使用子表达式。每个子表达式用一对括号包围：

```
// 整个正则表达式包含七个子表达式：( ddd )分隔符 ddd 分隔符 dddd
// 子表达式 1、3、4 和 6 是可选的；2、5 和 7 保存号码
"(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ]?)(\\d{4})";
```

由于我们的模式使用了括号，而且必须去除反斜线的特殊含义，因此这个模式很难读（也很难写!)。理解此模式的最简单的方法是逐个剥离（括号包围的）子表达式：

1. (\\()?表示区号部分可选的左括号
2. (\\d{3})表示区号
3. (\\))?表示区号部分可选的右括号
4. ([-.])?表示区号部分可选的分隔符
5. (\\d{3})表示号码的下三位数字
6. ([-.])?表示可选的分隔符
7. (\\d{4})表示号码的最后四位数字

下面的代码读取一个文件，并用此模式查找与完整的电话号码模式匹配的数据。它会调用一个名为valid的函数来检查号码格式是否合法：

740

```
string phone =
    "(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ]?)(\\d{4})";
regex r(phone); // regex 对象，用于查找我们的模式
smatch m;
string s;
// 从输入文件中读取每条记录
while (getline(cin, s)) {
    // 对每个匹配的电话号码
    for (sregex_iterator it(s.begin(), s.end(), r), end_it;
              it != end_it; ++it)
        // 检查号码的格式是否合法
        if (valid(*it))
            cout << "valid: " << it->str() << endl;
        else
            cout << "not valid: " << it->str() << endl;
}
```

使用子匹配操作

我们将使用表17.11中描述的子匹配操作来编写valid函数。需要记住的重要一点是，我们的 pattern 有七个子表达式。与往常一样，每个 smatch 对象会包含八个ssub_match元素。位置[0]的元素表示整个匹配；元素[1]...[7]表示每个对应的子表达式。

当调用 valid 时，我们知道已经有一个完整的匹配，但不知道每个可选的子表达式是否是匹配的一部分。如果一个子表达式是完整匹配的一部分，则其对应的 ssub_match 对象的matched成员为true。

表 17.11：子匹配操作

注意：这些操作适用于 ssub_match、csub_match、wssub_match、wcsub_match。

matched	一个 public bool 数据成员，指出此 ssub_match 是否匹配了
first	public 数据成员，指向匹配序列首元素和尾后位置的迭代器。如果未匹配，则 first 和 second 是相等的
second	
length()	匹配的大小。如果 matched 为 false，则返回 0
str()	返回一个包含输入中匹配部分的 string。如果 matched 为 false，则返回空 string
s = ssub	将 ssub_match 对象 ssub 转化为 string 对象 s。等价于 s=ssub.str()。转换运算符不是 explicit 的（参见 14.9.1 节，第 515 页）

741 在一个合法的电话号码中，区号要么是完整括号包围的，要么完全没有括号。因此，valid 要做什么工作依赖于号码是否以一个括号开始：

```
bool valid(const smatch& m)
{
    // 如果区号前有一个左括号
    if(m[1].matched)
        // 则区号后必须有一个右括号，之后紧跟剩余号码或一个空格
        return m[3].matched
               && (m[4].matched == 0 || m[4].str() == " ");
    else
        // 否则，区号后不能有右括号
        // 另两个组成部分间的分隔符必须匹配
        return !m[3].matched
               && m[4].str() == m[6].str();
}
```

我们首先检查第一个子表达式（即，左括号）是否匹配了。这个子表达式在 m[1]中。如果匹配了，则号码是以左括号开始的。在此情况下，如果区号后的子表达式也匹配了（意味着区号后有右括号）则整个号码是合法的。而且，如果号码正确使用了括号，则下一个字符必须是一个空格或下一部分的第一个数字。

如果 m[1]未匹配，（即，没有左括号），则区号后的子表达式也不应该匹配。如果它为空，则整个号码是合法的。

17.3.3 节练习

练习 17.19：为什么可以不先检查 m[4]是否匹配了就直接调用 m[4].str()？

练习 17.20：编写你自己版本的验证电话号码的程序。

练习 17.21：使用本节中定义的 valid 函数重写 8.3.2 节（第 289 页）中的电话号码程序。

练习 17.22：重写你的电话号码程序，使之允许在号码的三个部分之间放置任意多个空白符。

练习 17.23：编写查找邮政编码的正则表达式。一个美国邮政编码可以由五位或九位数字组成。前五位数字和后四位数字之间可以用一个短横线分隔。

17.3.4 使用 regex_replace

正则表达式不仅用在我们希望查找一个给定序列的时候，还用在当我们想将找到的序列替换为另一个序列的时候。例如，我们可能希望将美国的电话号码转换为“ddd.ddd.dddd”的形式，即，区号和后面三位数字用一个点分隔。

当我们希望在输入序列中查找并替换一个正则表达式时，可以调用 **regex_replace**。表 17.12 描述了 regex_replace，类似搜索函数，它接受一个输入字符序列和一个 regex 对象，不同的是，它还接受一个描述我们想要的输出形式的字符串。 742

表 17.12：正则表达式替换操作

`m.format(dest, fmt, mft)` `m.format(fmt, mft)`	使用格式字符串 *fmt* 生成格式化输出，匹配在 m 中，可选的 match_flag_type 标志在 mft 中。第一个版本写入迭代器 dest 指向的目的位置（参见 10.5.1 节，第 365 页）并接受 *fmt* 参数，可以是一个 string，也可以是表示字符数组中范围的一对指针。第二个版本返回一个 string，保存输出，并接受 *fmt* 参数，可以是一个 string，也可以是一个指向空字符结尾的字符数组的指针。mft 的默认值为 format_default
`regex_replace(dest,seq, r, fmt, mft)` `regex_replace(seq,r,fmt,mft)`	遍历 *seq*，用 regex_search 查找与 regex 对象 r 匹配的子串。使用格式字符串 *fmt* 和可选的 match_flag_type 标志来生成输出。第一个版本将输出写入到迭代器 dest 指定的位置，并接受一对迭代器 *seq* 表示范围。第二个版本返回一个 string，保存输出，且 *seq* 既可以是一个 string 也可以是一个指向空字符结尾的字符数组的指针。在所有情况下，*fmt* 既可以是一个 string 也可以是一个指向空字符结尾的字符数组的指针，且 mft 的默认值为 match_default

替换字符串由我们想要的字符组合与匹配的子串对应的子表达式而组成。在本例中，我们希望在替换字符串中使用第二个、第五个和第七个子表达式。而忽略第一个、第三个、第四个和第六个子表达式，因为这些子表达式用来形成号码的原格式而非新格式中的一部分。我们用一个符号$后跟子表达式的索引号来表示一个特定的子表达式：

```
string fmt = "$2.$5.$7"; // 将号码格式改为 ddd.ddd.dddd
```

可以像下面这样使用我们的正则表达式模式和替换字符串：

```
regex r(phone); // 用来寻找模式的 regex 对象
string number = "(908) 555-1800";
cout << regex_replace(number, r, fmt) << endl;
```

此程序的输出为：

```
908.555.1800
```

只替换输入序列的一部分

正则表达式更有意思的一个用处是替换一个大文件中的电话号码。例如，我们有一个保存人名及其电话号码的文件：

743
```
morgan (201) 555-2368 862-555-0123
drew (973)555.0130
lee (609) 555-0132 2015550175 800.555-0000
```

我们希望将数据转换为下面这样：

```
morgan 201.555.2368 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0000
```

可以用下面的程序完成这种转换：

```
int main()
{
    string phone =
       "(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ])?(\\d{4})";
    regex r(phone); // 寻找模式所用的 regex 对象
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // 将号码格式改为 ddd.ddd.dddd
    // 从输入文件中读取每条记录
    while (getline(cin, s))
        cout << regex_replace(s, r, fmt) << endl;
    return 0;
}
```

我们读取每条记录，保存到 s 中，并将其传递给 regex_replace。此函数在输入序列中查找并转换所有匹配子串。

用来控制匹配和格式的标志

就像标准库定义标志来指导如何处理正则表达式一样，标准库还定义了用来在替换过程中控制匹配或格式的标志。表 17.13 列出了这些值。这些标志可以传递给函数 regex_search 或 regex_match 或是类 smatch 的 format 成员。

匹配和格式化标志的类型为 match_flag_type。这些值都定义在名为 regex_constants 的命名空间中。类似用于 bind 的 placeholders（参见 10.3.4 节，第 355 页），regex_constants 也是定义在命名空间 std 中的命名空间。为了使用 regex_constants 中的名字，我们必须在名字前同时加上两个命名空间的限定符：

```
using std::regex_constants::format_no_copy;
```

此声明指出，如果代码中使用了 format_no_copy，则表示我们想要使用命名空间 std::constants 中的这个名字。如下所示，我们也可以用另一种形式的 using 来代替上面的代码，我们将在 18.2.2 节（第 702 页）中介绍这种形式：

```
using namespace std::regex_constants;
```

744

表 17.13：匹配标志

定义在 regex_constants::match_flag_type 中	
match_default	等价于 format_default
match_not_bol	不将首字符作为行首处理
match_not_eol	不将尾字符作为行尾处理
match_not_bow	不将首字符作为单词首处理
match_not_eow	不将尾字符作为单词尾处理
match_any	如果存在多于一个匹配，则可返回任意一个匹配
match_not_null	不匹配任何空序列
match_continuous	匹配必须从输入的首字符开始
match_prev_avail	输入序列包含第一个匹配之前的内容
format_default	用 ECMAScript 规则替换字符串
format_sed	用 POSIX sed 规则替换字符串
format_no_copy	不输出输入序列中未匹配的部分
format_first_only	只替换子表达式的第一次出现

使用格式标志

默认情况下，regex_replace 输出整个输入序列。未与正则表达式匹配的部分会原样输出；匹配的部分按格式字符串指定的格式输出。我们可以通过在 regex_replace 调用中指定 format_no_copy 来改变这种默认行为：

```
// 只生成电话号码：使用新的格式字符串
string fmt2 = "$2.$5.$7 "; // 在最后一部分号码后放置空格作为分隔符
// 通知 regex_replace 只拷贝它替换的文本
cout << regex_replace(s, r, fmt2, format_no_copy) << endl;
```

给定相同的输入，此版本的程序生成

```
201.555.2368 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0000
```

17.3.4 节练习

练习 17.24：编写你自己版本的重排电话号码格式的程序。

练习 17.25：重写你的电话号码程序，使之只输出每个人的第一个电话号码。

练习 17.26：重写你的电话号码程序，使之对多于一个电话号码的人只输出第二个和后续电话号码。

练习 17.27：编写程序，将九位数字邮政编码的格式转换为 ddddd-dddd。

17.4 随机数

745

程序通常需要一个随机数源。在新标准出现之前，C 和 C++都依赖于一个简单的 C 库函数 rand 来生成随机数。此函数生成均匀分布的伪随机整数，每个随机数的范围在 0 和一个系统相关的最大值（至少为 32767）之间。

C++ 11

rand 函数有一些问题：即使不是大多数，也有很多程序需要不同范围的随机数。一些应用需要随机浮点数。一些程序需要非均匀分布的数。而程序员为了解决这些问题而试图转换 rand 生成的随机数的范围、类型或分布时，常常会引入非随机性。

定义在头文件 random 中的随机数库通过一组协作的类来解决这些问题：**随机数引擎类**（random-number engines）和**随机数分布类**（random-number distribution）。表 17.14 描述了这些类。一个引擎类可以生成 unsigned 随机数序列，一个分布类使用一个引擎类生成指定类型的、在给定范围内的、服从特定概率分布的随机数。

表 17.14：随机数库的组成	
引擎	类型，生成随机 unsigned 整数序列
分布	类型，使用引擎返回服从特定概率分布的随机数



C++程序不应该使用库函数 rand，而应使用 default_random_engine 类和恰当的分布类对象。

17.4.1 随机数引擎和分布

随机数引擎是函数对象类（参见 14.8 节，第 506 页），它们定义了一个调用运算符，该运算符不接受参数并返回一个随机 unsigned 整数。我们可以通过调用一个随机数引擎对象来生成原始随机数：

```
default_random_engine e; // 生成随机无符号数
for (size_t i = 0; i < 10; ++i)
    // e() "调用" 对象来生成下一个随机数
    cout << e() << " ";
```

在我们的系统中，此程序生成：

```
16807 282475249 1622650073 984943658 1144108930 470211272 ...
```

在本例中，我们定义了一个名为 e 的 **default_random_engine** 对象。在 for 循环内，我们调用对象 e 来获得下一个随机数。

746

标准库定义了多个随机数引擎类，区别在于性能和随机性质量不同。每个编译器都会指定其中一个作为 default_random_engine 类型。此类型一般具有最常用的特性。表 17.15 列出了随机数引擎操作，标准库定义的引擎类型列在附录 A.3.2（第 783 页）中。

表 17.15：随机数引擎操作	
Engine e;	默认构造函数；使用该引擎类型默认的种子
Engine e(s);	使用整型值 s 作为种子
e.seed(s)	使用种子 s 重置引擎的状态
e.min() e.max()	此引擎可生成的最小值和最大值
Engine::result_type	此引擎生成的 unsigned 整型类型
e.discard(u)	将引擎推进 u 步；u 的类型为 unsigned long long

对于大多数场合，随机数引擎的输出是不能直接使用的，这也是为什么早先我们称之为原始随机数。问题出在生成的随机数的值范围通常与我们需要的不符，而正确转换随机数的范围是极其困难的。

分布类型和引擎

为了得到在一个指定范围内的数，我们使用一个分布类型的对象：

```
// 生成 0 到 9 之间（包含）均匀分布的随机数
uniform_int_distribution<unsigned> u(0,9);
default_random_engine e; // 生成无符号随机整数
for (size_t i = 0; i < 10; ++i)
    // 将 u 作为随机数源
    // 每个调用返回在指定范围内并服从均匀分布的值
    cout << u(e) << " ";
```

此代码生成下面这样的输出

```
0 1 7 4 5 2 0 6 6 9
```

此处我们将 u 定义为 uniform_int_distribution<unsigned>。此类型生成均匀分布的 unsigned 值。当我们定义一个这种类型的对象时，可以提供想要的最小值和最大值。在此程序中，u(0,9)表示我们希望得到 0 到 9 之间（包含）的数。随机数分布类会使用包含的范围，从而我们可以得到给定整型类型的每个可能值。

类似引擎类型，分布类型也是函数对象类。分布类型定义了一个调用运算符，它接受一个随机数引擎作为参数。分布对象使用它的引擎参数生成随机数，并将其映射到指定的分布。

注意，我们传递给分布对象的是引擎对象本身，即 u(e)。如果我们将调用写成 u(e())，含义就变为将 e 生成的下一个值传递给 u，会导致一个编译错误。我们传递的是引擎本身，而不是它生成的下一个值，原因是某些分布可能需要调用引擎多次才能得到一个值。



当我们说随机数发生器时，是指分布对象和引擎对象的组合。

比较随机数引擎和 rand 函数

对熟悉 C 库函数 rand 的读者，值得注意的是：调用一个 default_random_engine 对象的输出类似 rand 的输出。随机数引擎生成的 unsigned 整数在一个系统定义的范围内，而 rand 生成的数的范围在 0 到 RAND_MAX 之间。一个引擎类型的范围可以通过调用该类型对象的 min 和 max 成员来获得：

747

```
cout << "min: " << e.min() << " max: " << e.max() << endl;
```

在我们的系统中，此程序生成下面的输出：

```
min: 1 max: 2147483646
```

引擎生成一个数值序列

随机数发生器有一个特性经常会使新手迷惑：即使生成的数看起来是随机的，但对一个给定的发生器，每次运行程序它都会返回相同的数值序列。序列不变这一事实在调试时非常有用。但另一方面，使用随机数发生器的程序也必须考虑这一特性。

作为一个例子，假定我们需要一个函数生成一个 vector，包含 100 个均匀分布在 0 到 9 之间的随机数。我们可能认为应该这样编写此函数：

```
// 几乎肯定是生成随机整数 vector 的错误方法
// 每次调用这个函数都会生成相同的 100 个数!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

但是，每次调用这个函数都会返回相同的 vector：

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// 将打印"equal"
cout << ((v1 == v2) ? "equal" : "not equal") << endl;
```

748 此代码会打印 equal，因为 vector v1 和 v2 具有相同的值。

编写此函数的正确方法是将引擎和关联的分布对象定义为 static 的（参见 6.1.1 节，第 185 页）：

```
// 返回一个 vector，包含 100 个均匀分布的随机数
vector<unsigned> good_randVec()
{
    // 由于我们希望引擎和分布对象保持状态，因此应该将它们
    // 定义为 static 的，从而每次调用都生成新的数
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

由于 e 和 u 是 static 的，因此它们在函数调用之间会保持住状态。第一次调用会使用 u(e) 生成的序列中的前 100 个随机数，第二次调用会获得接下来 100 个，依此类推。



一个给定的随机数发生器一直会生成相同的随机数序列。一个函数如果定义了局部的随机数发生器，应该将其（包括引擎和分布对象）定义为 static 的。否则，每次调用函数都会生成相同的序列。

设置随机数发生器种子

随机数发生器会生成相同的随机数序列这一特性在调试中很有用。但是，一旦我们的程序调试完毕，我们通常希望每次运行程序都会生成不同的随机结果，可以通过提供一个**种子**（seed）来达到这一目的。种子就是一个数值，引擎可以利用它从序列中一个新位置重新开始生成随机数。

为引擎设置种子有两种方式：在创建引擎对象时提供种子，或者调用引擎的 seed 成员：

```
default_random_engine e1;                    // 使用默认种子
default_random_engine e2(2147483646);        // 使用给定的种子值
// e3 和 e4 将生成相同的序列，因为它们使用了相同的种子
default_random_engine e3;                    // 使用默认种子值
e3.seed(32767);                              // 调用 seed 设置一个新种子值
default_random_engine e4(32767);             // 将种子值设置为 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i << endl;
    if (e3() != e4())
        cout << "seeded differs at iteration: " << i << endl;
}
```

本例中我们定义了四个引擎。前两个引擎 e1 和 e2 的种子不同，因此应该生成不同的序列。后两个引擎 e3 和 e4 有相同的种子，它们将生成相同的序列。 749

选择一个好的种子，与生成好的随机数所涉及的其他大多数事情相同，是极其困难的。可能最常用的方法是调用系统函数 time。这个函数定义在头文件 ctime 中，它返回从一个特定时刻到当前经过了多少秒。函数 time 接受单个指针参数，它指向用于写入时间的数据结构。如果此指针为空，则函数简单地返回时间：

```
default_random_engine e1(time(0)); // 稍微随机些的种子
```

由于 time 返回以秒计的时间，因此这种方式只适用于生成种子的间隔为秒级或更长的应用。



如果程序作为一个自动过程的一部分反复运行，将 time 的返回值作为种子的方式就无效了；它可能多次使用的都是相同的种子。

17.4.1 节练习

练习 17.28：编写函数，每次调用生成并返回一个均匀分布的随机 unsigned int。

练习 17.29：修改上一题中编写的函数，允许用户提供一个种子作为可选参数。

练习 17.30：再次修改你的程序，此次再增加两个参数，表示函数允许返回的最小值和最大值。

17.4.2 其他随机数分布

随机数引擎生成 unsigned 数，范围内的每个数被生成的概率都是相同的。而应用程序常常需要不同类型或不同分布的随机数。标准库通过定义不同随机数分布对象来满足这两方面的要求，分布对象和引擎对象协同工作，生成要求的结果。表 17.16 列出了分布类型所支持的操作。

生成随机实数

程序常需要一个随机浮点数的源。特别是，程序经常需要 0 到 1 之间的随机数。

最常用但不正确的从 rand 获得一个随机浮点数的方法是用 rand() 的结果除以 RAND_MAX，即，系统定义的 rand 可以生成的最大随机数的上界。这种方法不正确的原因是随机整数的精度通常低于随机浮点数，这样，有一些浮点值就永远不会被生成了。

使用新标准库设施，可以很容易地获得随机浮点数。我们可以定义一个 750

uniform_real_distribution 类型的对象，并让标准库来处理从随机整数到随机浮点数的映射。与处理 uniform_int_distribution 一样，在定义对象时，我们指定最小值和最大值：

```
default_random_engine e; // 生成无符号随机整数
// 0 到 1（包含）的均匀分布
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
     cout << u(e) << " ";
```

这段代码与之前生成 unsigned 值的程序几乎相同。但是，由于我们使用了一个不同的分布类型，此版本会生成不同的结果：

```
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
```

表 17.16：分布类型的操作

Dist d;	默认构造函数；使 d 准备好被使用。 其他构造函数依赖于 *Dist* 的类型；参见附录 A.3 节（第 781 页）。 分布类型的构造函数是 explicit 的（参见 7.5.4 节，第 265 页）
d(e)	用相同的 e 连续调用 d 的话，会根据 d 的分布式类型生成一个随机数序列；e 是一个随机数引擎对象
d.min() d.max()	返回 d(e) 能生成的最小值和最大值
d.reset()	重建 d 的状态，使得随后对 d 的使用不依赖于 d 已经生成的值

使用分布的默认结果类型

分布类型都是模板，具有单一的模板类型参数，表示分布生成的随机数的类型，对此有一个例外，我们将在 17.4.2 节（第 665 页）中进行介绍。这些分布类型要么生成浮点类型，要么生成整数类型。

每个分布模板都有一个默认模板实参（参见 16.1.3 节，第 594 页）。生成浮点值的分布类型默认生成 double 值，而生成整型值的分布默认生成 int 值。由于分布类型只有一个模板参数，因此当我们希望使用默认随机数类型时要记得在模板名之后使用空尖括号（参见 16.1.3 节，第 594 页）：

```
// 空<>表示我们希望使用默认结果类型
uniform_real_distribution<> u(0,1); // 默认生成 double 值
```

751 生成非均匀分布的随机数

除了正确生成在指定范围内的数之外，新标准库的另一个优势是可以生成非均匀分布的随机数。实际上，新标准库定义了 20 种分布类型，这些类型列在附录 A.3（第 781）中。

作为一个例子，我们将生成一个正态分布的值的序列，并画出值的分布。由于 normal_distribution 生成浮点值，我们的程序使用头文件 cmath 中的 lround 函数将每个随机数舍入到最接近的整数。我们将生成 200 个数，它们以均值 4 为中心，标准差为 1.5。由于使用的是正态分布，我们期望生成的数中大约 99%都在 0 到 8 之间（包含）。我们的程序会对这个范围内的每个整数统计有多少个生成的数映射到它：

```
default_random_engine e;          // 生成随机整数
normal_distribution<> n(4,1.5);   // 均值 4，标准差 1.5
```

```
vector<unsigned> vals(9);          // 9个元素均为0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e));     // 舍入到最接近的整数
    if (v < vals.size())           // 如果结果在范围内
        ++vals[v];                 // 统计每个数出现了多少次
}
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ": " << string(vals[j], '*') << endl;
```

我们首先定义了随机数发生器对象和一个名为 vals 的 vector。我们用 vals 来统计范围 0...8 中的每个数出现了多少次。与我们使用 vector 的大多数程序不同，此程序按需求大小为 vals 分配空间，每个元素都被初始化为 0。

在 for 循环中，我们调用 lround(n(e)) 来将 n(e) 返回的值舍入到最接近的整数。获得浮点随机数对应的整数后，我们将它作为计数器 vector 的下标。由于 n(e) 可能生成范围 0 到 8 之外的数，所以我们首先检查生成的数是否在范围内，然后再将其作为 vals 的下标。如果结果确实在范围内，我们递增对应的计数器。

当循环结束时，我们打印 vals 的内容，可能会打印出像下面这样的结果：

```
0: ***
1: ********
2: ********************
3: ************************************
4: ************************************************************
5: ******************************************
6: ************************
7: *******
8: *
```

本例中我们打印一个由星号组成的 string，有多少随机数等于此下标我们就打印多少个星号。注意，此图并不是完美对称的。如果打印出的图是完美对称的，我们反倒有理由怀疑随机数发生器的质量了。 752

bernoulli_distribution 类

我们注意到有一个分布不接受模板参数，即 bernoulli_distribution，因为它是一个普通类，而非模板。此分布总是返回一个 bool 值。它返回 true 的概率是一个常数，此概率的默认值是 0.5。

作为一个这种分布的例子，我们可以编写一个程序，这个程序与用户玩一个游戏。为了进行这个游戏，其中一个游戏者——用户或是程序——必须先行。我们可以用一个值范围是 0 到 1 的 uniform_int_distribution 来选择先行的游戏者，但也可以用伯努利分布来完成这个选择。假定已有一个名为 play 的函数来进行游戏，我们可以编写像下面这样的循环来与用户交互：

```
string resp;
default_random_engine e; // e应保持状态，所以必须在循环外定义！
bernoulli_distribution b;// 默认是50/50的机会
do {
    bool first = b(e);    // 如果为true，则程序先行
    cout << (first ? "We go first"
                   : "You get to go first") << endl;
```

```
        // 传递谁先行的指示，进行游戏
        cout << ((play(first)) ? "sorry, you lost"
                               : "congrats, you won") << endl;
        cout << "play again? Enter 'yes' or 'no'" << endl;
    } while (cin >> resp && resp[0] == 'y');
```

我们用一个 do while 循环（参见 5.4.4 节，第 169 页）来反复提示用户进行游戏。



由于引擎返回相同的随机数序列（参见 17.4.1 节，第 661 页），所以我们必须在循环外声明引擎对象。否则，每步循环都会创建一个新引擎，从而每步循环都会生成相同的值。类似的，分布对象也要保持状态，因此也应该在循环外定义。

在此程序中使用 bernoulli_distribution 的一个原因是它允许我们调整选择先行一方的概率：

```
bernoulli_distribution b(.55); // 给程序一个微小的优势
```

如果 b 定义如上，则程序有 55/45 的机会先行。

17.4.2 节练习

练习 17.31：对于本节中的游戏程序，如果在 do 循环内定义 b 和 e，会发生什么？

练习 17.32：如果我们在循环内定义 resp，会发生什么？

练习 17.33：修改 11.3.6 节（第 392 页）中的单词转换程序，允许对一个给定单词有多种转换方式，每次随机选择一种进行实际转换。

17.5 IO 库再探

在第 8 章中我们介绍了 IO 库的基本结构及其最常用的部分。在本节中，我们将介绍三个更特殊的 IO 库特性：格式控制、未格式化 IO 和随机访问。

753

17.5.1 格式化输入与输出

除了条件状态外（参见 8.1.2 节，第 279 页），每个 iostream 对象还维护一个格式状态来控制 IO 如何格式化的细节。格式状态控制格式化的某些方面，如整型值是几进制、浮点值的精度、一个输出元素的宽度等。

标准库定义了一组**操纵符**（manipulator）（参见 1.2 节，第 6 页）来修改流的格式状态，如表 17.7 和表 17.8 所示。一个操纵符是一个函数或是一个对象，会影响流的状态，并能用作输入或输出运算符的运算对象。类似输入和输出运算符，操纵符也返回它所处理的流对象，因此我们可以在一条语句中组合操纵符和数据。

我们已经在程序中使用过一个操纵符——endl，我们将它“写”到输出流，就像它是一个值一样。但 endl 不是一个普通值，而是一个操作：它输出一个换行符并刷新缓冲区。

很多操纵符改变格式状态

操纵符用于两大类输出控制：控制数值的输出形式以及控制补白的数量和位置。大多数改变格式状态的操纵符都是设置/复原成对的：一个操纵符用来将格式状态设置为一个新值，而另一个用来将其复原，恢复为正常的默认格式。



当操纵符改变流的格式状态时，通常改变后的状态对所有后续 IO 都生效。

当我们有一组 IO 操作希望使用相同的格式时，操纵符对格式状态的改变是持久的这一特性很有用。实际上，一些程序会利用操纵符的这一特性对其所有输入或输出重置一个或多个格式规则的行为。在这种情况下，操纵符会改变流这一特性就是满足要求的了。

但是，很多程序（而且更重要的是，很多程序员）期望流的状态符合标准库正常的默认设置。在这些情况下，将流的状态置于一个非标准状态可能会导致错误。因此，通常最好在不再需要特殊格式时尽快将流恢复到默认状态。

控制布尔值的格式

754

操纵符改变对象的格式状态的一个例子是 boolalpha 操纵符。默认情况下，bool 值打印为 1 或 0。一个 true 值输出为整数 1，而 false 输出为 0。我们可以通过对流使用 boolalpha 操纵符来覆盖这种格式：

```
cout << "default bool values: " << true << " " << false
     << "\nalpha bool values: " << boolalpha
     << true << " " << false << endl;
```

执行这段程序会得到下面的结果：

```
default bool values: 1 0
alpha bool values: true false
```

一旦向 cout “写入”了 boolalpha，我们就改变了 cout 打印 bool 值的方式。后续打印 bool 值的操作都会打印 true 或 false 而非 1 或 0。

为了取消 cout 格式状态的改变，我们使用 noboolalpha：

```
bool bool_val = get_status();
cout << boolalpha     // 设置 cout 的内部状态
<< bool_val
<< noboolalpha;       // 将内部状态恢复为默认格式
```

本例中我们改变了 bool 值的格式，但只对 bool_val 的输出有效。一旦完成此值的打印，我们立即将流恢复到初始状态。

指定整型值的进制

默认情况下，整型值的输入输出使用十进制。我们可以使用操纵符 hex、oct 和 dec 将其改为十六进制、八进制或是改回十进制：

```
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;
```

当编译并执行这段程序时，会得到如下输出：

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

注意，类似 boolalpha，这些操纵符也会改变格式状态。它们会影响下一个和随后所有的整型输出，直至另一个操纵符又改变了格式为止。



操纵符 hex、oct 和 dec 只影响整型运算对象，浮点值的表示形式不受影响。

755 在输出中指出进制

默认情况下，当我们打印出数值时，没有可见的线索指出使用的是几进制。例如，20 是十进制的 20 还是 16 的八进制表示？当我们按十进制打印数值时，打印结果会符合我们的期望。如果需要打印八进制值或十六进制值，应该使用 showbase 操纵符。当对流应用 showbase 操纵符时，会在输出结果中显示进制，它遵循与整型常量中指定进制相同的规范：

- 前导 0x 表示十六进制。
- 前导 0 表示八进制。
- 无前导字符串表示十进制。

我们可以使用 showbase 修改前一个程序：

```
cout << showbase; // 当打印整型值时显示进制
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase; // 恢复流状态
```

修改后的程序的输出会更清楚地表明底层值到底是什么：

```
default: 20 1024
in octal: 024 02000
in hex: 0x14 0x400
in decimal: 20 1024
```

操纵符 noshowbase 恢复 cout 的状态，从而不再显示整型值的进制。

默认情况下，十六进制值会以小写打印，前导字符也是小写的 x。我们可以通过使用 uppercase 操纵符来输出大写的 X 并将十六进制数字 a-f 以大写输出：

```
cout << uppercase << showbase << hex
     << "printed in hexadecimal: " << 20 << " " << 1024
     << nouppercase << noshowbase << dec << endl;
```

这条语句生成如下输出：

```
printed in hexadecimal: 0X14 0X400
```

我们使用了操纵符 nouppercase、noshowbase 和 dec 来重置流的状态。

控制浮点数格式

我们可以控制浮点数输出三个种格式：

- 以多高精度（多少个数字）打印浮点值
- 数值是打印为十六进制、定点十进制还是科学记数法形式
- 对于没有小数部分的浮点值是否打印小数点

756

默认情况下，浮点值按六位数字精度打印；如果浮点值没有小数部分，则不打印小数点；根据浮点数的值选择打印成定点十进制或科学记数法形式。标准库会选择一种可读性更好的格式：非常大和非常小的值打印为科学记数法形式，其他值打印为定点十进制形式。

指定打印精度

默认情况下，精度会控制打印的数字的总数。当打印时，浮点值按当前精度舍入而非截断。因此，如果当前精度为四位数字，则 3.14159 将打印为 3.142；如果精度为三位数字，则打印为 3.14。

我们可以通过调用 IO 对象的 precision 成员或使用 setprecision 操纵符来改变精度。precision 成员是重载的（参见 6.4 节，第 206 页）。一个版本接受一个 int 值，将精度设置为此值，并返回旧精度值。另一个版本不接受参数，返回当前精度值。setprecision 操纵符接受一个参数，用来设置精度。



操纵符 setprecision 和其他接受参数的操纵符都定义在头文件 iomanip 中。

下面的程序展示了控制浮点值打印精度的不同方法：

```
// cout.precision 返回当前精度值
cout << "Precision: " << cout.precision()
     << ", Value: "   << sqrt(2.0) << endl;
// cout.precision(12)将打印精度设置为 12 位数字
cout.precision(12);
cout << "Precision: " << cout.precision()
     << ", Value: "   << sqrt(2.0) << endl;
// 另一种设置精度的方法是使用 setprecision 操纵符
cout << setprecision(3);
cout << "Precision: " << cout.precision()
     << ", Value: "   << sqrt(2.0) << endl;
```

编译并执行这段程序，会得到如下输出：

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

此程序调用标准库 sqrt 函数，它定义在头文件 cmath 中。sqrt 函数是重载的，不同版本分别接受一个 float、double 或 long double 参数，返回实参的平方根。

757

表 17.17：定义在 iostream 中的操纵符

	boolalpha	将 true 和 false 输出为字符串
*	noboolalpha	将 true 和 false 输出为 1，0
	showbase	对整型值输出表示进制的前缀
*	noshowbase	不生成表示进制的前缀
	showpoint	对浮点值总是显示小数点

续表

* noshowpoint	只有当浮点值包含小数部分时才显示小数点
showpos	对非负数显示+
* noshowpos	对非负数不显示+
uppercase	在十六进制值中打印 0X，在科学记数法中打印 E
* nouppercase	在十六进制值中打印 0x，在科学记数法中打印 e
* dec	整型值显示为十进制
hex	整型值显示为十六进制
oct	整型值显示为八进制
left	在值的右侧添加填充字符
right	在值的左侧添加填充字符
internal	在符号和值之间添加填充字符
fixed	浮点值显示为定点十进制
scientific	浮点值显示为科学记数法
hexfloat	浮点值显示为十六进制（C++11 新特性）
defaultfloat	重置浮点数格式为十进制（C++11 新特性）
unitbuf	每次输出操作后都刷新缓冲区
* nounitbuf	恢复正常的缓冲区刷新方式
* skipws	输入运算符跳过空白符
noskipws	输入运算符不跳过空白符
flush	刷新 ostream 缓冲区
ends	插入空字符，然后刷新 ostream 缓冲区
endl	插入换行，然后刷新 ostream 缓冲区

* 表示默认流状态

指定浮点数记数法



除非你需要控制浮点数的表示形式（如，按列打印数据或打印表示金额或百分比的数据），否则由标准库选择记数法是最好的方式。

通过使用恰当的操纵符，我们可以强制一个流使用科学记数法、定点十进制或是十六进制记数法。操纵符 scientific 改变流的状态来使用科学记数法。操纵符 fixed 改变流的状态来使用定点十进制。

C++ 11

在新标准库中，通过使用 hexfloat 也可以强制浮点数使用十六进制格式。新标准库还提供另一个名为 defaultfloat 的操纵符，它将流恢复到默认状态——根据要打印的值选择记数法。

758

这些操纵符也会改变流的精度的默认含义。在执行 scientific、fixed 或 hexfloat 后，精度值控制的是小数点后面的数字位数，而默认情况下精度值指定的是数字的总位数——既包括小数点之后的数字也包括小数点之前的数字。使用 fixed 或 scientific 令我们可以按列打印数值，因为小数点距小数部分的距离是固定的：

```
cout << "default format: " << 100 * sqrt(2.0) << '\n'
     << "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
     << "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
     << "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
     << "use defaults: " << defaultfloat << 100 * sqrt(2.0)
```

```
        << "\n\n";
```

此程序会生成下面的输出：

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcp+7
use defaults: 141.421
```

默认情况下，十六进制数字和科学记数法中的 e 都打印成小写形式。我们可以用 uppercase 操纵符打印这些字母的大写形式。

打印小数点

默认情况下，当一个浮点值的小数部分为 0 时，不显示小数点。showpoint 操纵符强制打印小数点：

```
cout << 10.0 << endl;          // 打印 10
cout << showpoint << 10.0      // 打印 10.0000
     << noshowpoint << endl;   // 恢复小数点的默认格式
```

操纵符 noshowpoint 恢复默认行为。下一个输出表达式将有默认行为，即，当浮点值的小数部分为 0 时不输出小数点。

输出补白

当按列打印数据时，我们常常需要非常精细地控制数据格式。标准库提供了一些操纵符帮助我们完成所需的控制：

- setw 指定下一个数字或字符串值的最小空间。
- left 表示左对齐输出。
- right 表示右对齐输出，右对齐是默认格式。
- internal 控制负数的符号的位置，它左对齐符号，右对齐值，用空格填满所有中间空间。 759
- setfill 允许指定一个字符代替默认的空格来补白输出。



setw 类似 endl，不改变输出流的内部状态。它只决定下一个输出的大小。

下面程序展示了如何使用这些操纵符：

```
int i = -16;
double d = 3.14159;
// 补白第一列，使用输出中最小 12 个位置
cout << "i: " << setw(12) << i << "next col" << '\n'
     << "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，左对齐所有列
cout << left
     << "i: " << setw(12) << i << "next col" << '\n'
     << "d: " << setw(12) << d << "next col" << '\n'
     << right; // 恢复正常对齐
// 补白第一列，右对齐所有列
cout << right
     << "i: " << setw(12) << i << "next col" << '\n'
```

```
        << "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，但补在域的内部
cout << internal
        << "i: " << setw(12) << i << "next col" << '\n'
        << "d: " << setw(12) << d << "next col" << '\n';
// 补白第一列，用#作为补白字符
cout << setfill('#')
        << "i: " << setw(12) << i << "next col" << '\n'
        << "d: " << setw(12) << d << "next col" << '\n'
        << setfill(' '); // 恢复正常的补白字符
```

执行这段程序，会得到下面的输出：

```
i:           -16next col
d:       3.14159next col
i: -16           next col
d: 3.14159       next col
i:           -16next col
d:       3.14159next col
i: -           16next col
d:       3.14159next col
i:  -##########16next col
d:  #####3.14159next col
```

760

表 17.18：定义在 iomanip 中的操纵符

setfill(ch)	用 ch 填充空白
setprecision(n)	将浮点精度设置为 n
setw(w)	读或写值的宽度为 w 个字符
setbase(b)	将整数输出为 b 进制

控制输入格式

默认情况下，输入运算符会忽略空白符（空格符、制表符、换行符、换纸符和回车符）。下面的循环

```
char ch;
while (cin >> ch)
    cout << ch;
```

当给定下面输入序列时

```
a b     c
d
```

循环会执行 4 次，读取字符 a 到 d，跳过中间的空格以及可能的制表符和换行符。此程序的输出是

```
abcd
```

操纵符 noskipws 会令输入运算符读取空白符，而不是跳过它们。为了恢复默认行为，我们可以使用 skipws 操纵符：

```
cin >> noskipws; // 设置 cin 读取空白符
while (cin >> ch)
    cout << ch;
```

```
cin >> skipws; // 将 cin 恢复到默认状态，从而丢弃空白符
```

给定与前一个程序相同的输入，此循环会执行 7 次，从输入中既读取普通字符又读取空白符。此循环的输出为

```
a b      c
d
```

17.5.1 节练习

练习 17.34：编写一个程序，展示如何使用表 17.17 和表 17.18 中的每个操纵符。

练习 17.35：修改第 670 页中的程序，打印 2 的平方根，但这次打印十六进制数字的大写形式。

练习 17.36：修改上一题中的程序，打印不同的浮点数，使它们排成一列。

17.5.2 未格式化的输入/输出操作

761

到目前为止，我们的程序只使用过**格式化 IO**（formatted IO）操作。输入和输出运算符（<<和>>）根据读取或写入的数据类型来格式化它们。输入运算符忽略空白符，输出运算符应用补白、精度等规则。

标准库还提供了一组低层操作，支持**未格式化 IO**（unformatted IO）。这些操作允许我们将一个流当作一个无解释的字节序列来处理。

单字节操作

有几个未格式化操作每次一个字节地处理流。这些操作列在表 17.19 中，它们会读取而不是忽略空白符。例如，我们可以使用未格式化 IO 操作 `get` 和 `put` 来读取和写入一个字符：

```
char ch;
while (cin.get(ch))
        cout.put(ch);
```

此程序保留输入中的空白符，其输出与输入完全相同。它的执行过程与前一个使用 `noskipws` 的程序完全相同。

表 17.19：单字节低层 IO 操作

`is.get(ch)`	从 `istream is` 读取下一个字节存入字符 `ch` 中。返回 `is`
`os.put(ch)`	将字符 `ch` 输出到 `ostream os`。返回 `os`
`is.get()`	将 `is` 的下一个字节作为 `int` 返回
`is.putback(ch)`	将字符 `ch` 放回 `is`。返回 `is`
`is.unget()`	将 `is` 向后移动一个字节。返回 `is`
`is.peek()`	将下一个字节作为 `int` 返回，但不从流中删除它

将字符放回输入流

有时我们需要读取一个字符才能知道还未准备好处理它。在这种情况下，我们希望将字符放回流中。标准库提供了三种方法退回字符，它们有着细微的差别：

- `peek` 返回输入流中下一个字符的副本，但不会将它从流中删除，`peek` 返回的值仍然留在流中。

- unget 使得输入流向后移动，从而最后读取的值又回到流中。即使我们不知道最后从流中读取什么值，仍然可以调用 unget。
- putback 是更特殊版本的 unget：它退回从流中读取的最后一个值，但它接受一个参数，此参数必须与最后读取的值相同。

762 一般情况下，在读取下一个值之前，标准库保证我们可以退回最多一个值。即，标准库不保证在中间不进行读取操作的情况下能连续调用 putback 或 unget。

从输入操作返回的 int 值

函数 peek 和无参的 get 版本都以 int 类型从输入流返回一个字符。这有些令人吃惊，可能这些函数返回一个 char 看起来会更自然。

这些函数返回一个 int 的原因是：可以返回文件尾标记。我们使用 char 范围中的每个值来表示一个真实字符，因此，取值范围中没有额外的值可以用来表示文件尾。

返回 int 的函数将它们要返回的字符先转换为 unsigned char，然后再将结果提升到 int。因此，即使字符集中有字符映射到负值，这些操作返回的 int 也是正值（参见 2.1.2 节，第 32 页）。而标准库使用负值表示文件尾，这样就可以保证与任何合法字符的值都不同。头文件 cstdio 定义了一个名为 EOF 的 const，我们可以用它来检测从 get 返回的值是否是文件尾，而不必记忆表示文件尾的实际数值。对我们来说重要的是，用一个 int 来保存从这些函数返回的值：

```
int ch; // 使用一个 int，而不是一个 char 来保存 get()的返回值
// 循环读取并输出输入中的所有数据
while ((ch = cin.get()) != EOF)
        cout.put(ch);
```

此程序与第 673 页中的程序完成相同的工作，唯一的不同是用来读取输入的 get 版本不同。

多字节操作

一些未格式化 IO 操作一次处理大块数据。如果速度是要考虑的重点问题的话，这些操作是很重要的，但类似其他低层操作，这些操作也容易出错。特别是，这些操作要求我们自己分配并管理用来保存和提取数据的字符数组（参见 12.2 节，第 423 页）。表 17.20 列出了多字节操作。

表 17.20：多字节低层 IO 操作

`is.get(sink, size, delim)`
从 is 中读取最多 size 个字节，并保存在字符数组中，字符数组的起始地址由 sink 给出。读取过程直至遇到字符 delim 或读取了 size 个字节或遇到文件尾时停止。如果遇到了 delim，则将其留在输入流中，不读取出来存入 sink

`is.getline(sink, size, delim)`
与接受三个参数的 get 版本类似，但会读取并丢弃 delim

`is.read(sink, size)`
读取最多 size 个字节，存入字符数组 sink 中。返回 is

`is.gcount()`
返回上一个未格式化读取操作从 is 读取的字节数

`os.write(source, size)`
将字符数组 source 中的 size 个字节写入 os。返回 os