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

C++ 11

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。

692

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));
    // ...
}
```

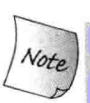
本例中我们使用 Type 作为 forward 的显式模板实参类型，它是从 arg 推断出来的。由于 arg 是一个模板类型参数的右值引用，Type 将表示传递给 arg 的实参的所有类型信息。如果实参是一个右值，则 Type 是一个普通（非引用）类型，forward<Type>将返回 Type&&。如果实参是一个左值，则通过引用折叠，Type 本身是一个左值引用类型。在此情况下，返回类型是一个指向左值引用类型的右值引用。再次对 forward<Type>的返回类型进行引用折叠，将返回一个左值引用类型。

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。