10

函 数 模 板

本章将讲述什么是函数模板（function template），并讨论怎样定义和使用函数模板。使用函数模板其实相当简单，许多C++初学者在使用库中定义的函数模板时，甚至不知道他们在使用模板。只有高级C++用户才会像本章中描述的那样定义和使用函数模板。因此，本章的内容可被用作高级C++主题的介绍性资料。我们从描述什么是函数模板以及怎样定义函数模板开始。然后说明函数模板的简单用法。在这之后。再将焦点转移到更高级的话题上。首先，我们将了解怎样以更高级的方式使用函数模板：我们将详细了解模板实参的推演过程，看看当引用一个模板实例时，怎样指定显式模板参数。然后我们会了解编译器怎样初始化模板及其对程序组织结构上的要求，并讨论怎样定义函数模板实例的特化版本。然后，本章将给出一些让函数模板设计者感兴趣的话题。我们将解释函数模板怎样被重载，以及涉及函数模板的重载解析过程如何工作。我们还会介绍函数模板定义中的名字解析。以及函数模板如何才能被定义在名字空间中。最后，本章将以一个使用函数模板的例子作为结束。

10.1 函数模板定义

有时候，强类型语言对于实现相对简单的函数似乎是个障碍。例如，虽然下面的函数min()的算法很简单，但是，强类型语言要求我们为所有希望比较的类型都实现一个实例：

```
int min( int a, int b ) {
     return a < b ? a : b;
}

double min( double a, double b ) {
     return a < b ? a : b;
}
```

有一种方法可替代这种“为每个min()实例都显式定义一个函数”的方法，这种方法很有吸引力，但是也很危险，那就是用预处理器的宏扩展设施。例如：

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

虽然该定义对于简单的 min()调用都能正常工作，如

```
min( 10, 20 );
min( 10.0, 20.0 );
```

但是，在复杂调用下，它的行为是不可预期的，这是因为它的机制并不像函数调用那样工作，只是简单地提供参数的替换。结果是，它的两个参数值都被计算两次：一次是在 a 和 b 的测试中，另一次是在宏的返回值被计算期间，例如：

```
#include <iostream>
#define min(a,b) ((a) < (b) ? (a) : (b))

const int size = 10;
int ia[size];

int main() {
    int elem_cnt = 0;
    int *p = &ia[0];

    // 计数数组元素的个数
    while ( min(p++,&ia[size]) != &ia[size] )
        ++elem_cnt;

    cout << "elem_cnt : " << elem_cnt
        << "\texpecting: " << size << endl;
    return 0;
}
```

这个程序给出了计算整型数组 ia 的元素个数的一种明显绕弯的的方法。min()的宏扩展在这种情况下会失败，因为应用在指针实参 p 上的后置递增操作随每次扩展而被应用了两次。执行该程序的结果是下面不正确的计算结果：

```
elem_cnt : 5  expecting: 10
```

函数模板提供了一种机制，通过它我们可以保留函数定义和函数调用的语义（在一个程序位置上封装了一段代码，确保在函数调用之前实参只被计算一次），而无需像宏方案那样绕过 C++的强类型检查。

函数模板提供一个种用来自动生成各种类型函数实例的算法。程序员对于函数接口（参数和返回类型）中的全部或者部分类型进行**参数化**（parameterize），而函数体保持不变。如用一个函数的实现在一组实例上保持不变，并且每个实例都处理一种惟一的数据类型，如函数 min()，则该函数就是模板的最佳候选者。例如，下面是 min()的函数模板定义：

```
template <class Type>
Type min( Type a, Type b ) {
    return a < b ? a : b;
}

int main() {
    // ok: int min( int, int );
    min( 10, 20 );
    // ok: double min( double, double );
    min( 10.0, 20.0 );
```

```
        return 0;
    }
```

如果用函数模板代替前面程序中的预处理器宏 min()，则程序的输出是正确的：

```
    elem_cnt : 10 expecting: 10
```

［C++标准库为一些常用的算法，如这里定义的 min()，提供了函数模板。这些算法将在第 12 章描述。为了介绍函数模板，我们对于 C++标准库中定义的一些算法给出了相应的简化版本。］

关键字 template 总是放在模板的定义与声明的最前面。关键字后面是用逗号分隔的**模板参数表**（template parameter list），它用尖括号（<>，一个小于号和一个大于号）括起来。该列表是模板参数表，不能为空。模板参数可以是一个**模板类型参数**（template type parameter），它代表了一种类型；也可以是一个**模板非类型参数**（template nontype parameter），它代表了一个常量表达式。

模板类型参数由关键字 class 或 typename 后加一个标识符构成。在函数的模板参数表中，这两个关键字的意义相同。它们表示后面的参数名代表一个潜在的内置或用户定义的类型。模板参数名由程序员选择。在本例中，我们用 Type 来命名 min()的模板参数，但实际上可以是任何名字。譬如：

```
template <class Glorp>
    Glorp min( Glorp a, Glorp b ) {
        return a < b ? a : b;
}
```

当模板被实例化时，实际的内置或用户定义类型将替换模板的类型参数。类型 int、double、char*、vector<int>或 list<double>*都是有效的模板实参类型。

模板非类型参数由一个普通的参数声明构成。模板非类型参数表示该参数名代表了一个潜在的值，而该值代表了模板定义中的一个常量。例如，size 是一个模板非类型参数，它代表 arr 指向的数组的长度：

```
template <class Type, int size>
    Type min( Type (&arr) [size] );
```

当函数模板 min()被实例化时，size 的值会被一个编译时刻已知的常量值代替。

函数定义或声明跟在模板参数表后。除了模板参数是类型指示符或常量值外，函数模板的定义看起来与非模板函数的定义相同。我们来看一个例子：

```
template <class Type, int size>

Type min( const Type (&r_array)[size] )
{
    /* 找到数组中元素最小值的参数化函数 */
    Type min_val = r_array[0];

    for ( int i = 1; i < size; ++i )
        if ( r_array[i] < min_val )
            min_val = r_array[i];

    return min_val;
}
```

在我们的例子中，Type 表示 min()的返回类型、参数 r_array 的类型，以及局部变量 min_val 的类型。size 表示 r_array 引用的数组的长度。在程序的运行过程中，Type 会被各种内置类型和用户定义的类型所代替，而 size 会被各种常量值所取代，这些常量值是由实际使用的 min()决定的（记住，一个函数的两种用法是调用它和取它的地址。）。类型和值的替换过程被称为**模板实例化**（template instantiation）。我们将在下一节介绍模板实例化。

我们的 min()函数模板的函数参数表看起来可能有些短。如 7.3 节所讨论的，一个数组参数总是被作为指向数组首元素的指针来传递，数组实参的第一维在函数定义内是未知的。为了缓解这个问题，我们在此处把 min()的参数声明为数组的引用。这解决了用户必须传递第二个实参来指定数组长度的问题，但是缺点是用在不同长度的 int 数组时，会生成或实例化不同的 min()实例。

当一个名字被声明为模板参数之后，它就可以被使用了，一直到模板声明或定义结束为止。模板类型参数被用作一个类型指示符，可以出现在模板定义的余下部分。它的使用方式与内置或用户定义的类型完全一样，比如用来声明变量和强制类型转换。模板非类型参数被用作一个常量值，可以出现在模板定义的余下部分。它可以用在要求常量的地方，或许是在数组声明中指定数组的大小或作为枚举常量的初始值。

```
// size 指定数组参数的大小并初始化一个 const int 值
template <class Type, int size>
Type min( const Type (&r_array)[size] )
{
     const int loc_size = size;
     Type loc_array[loc_size];
     // ...
}
```

如果在全局域中声明了与模板参数同名的对象、函数或类型，则该全局名将被隐藏。在下面的例子中，tmp 的类型不是 double，是模板参数 Type：

```
typedef double Type;
template <class Type>
Type min( Type a, Type b )
{
     // tmp 类型为模板参数 Type
     // 不是全局 typedef
     Type tmp = a < b ? a : b;
     return tmp;
}
```

在函数模板定义中声明的对象或类型不能与模板参数同名：

```
template <class Type>
Type min( Type a, Type b )
{
     // 错误：重新声明模板参数 Type
     typedef double Type;
     Type tmp = a < b ? a : b;
     return tmp;
}
```

模板类型参数名可以被用来指定函数模板的返回位：

```
// ok: T1 表示 min() 的返回类型
// T2 和 T3 表示参数类型
template <class T1, class T2, class T3>
     T1 min( T2, T3 );
```

模板参数名在同一模板参数表中只能被使用一次。例如，下面代码就有编译错误：

```
// 错误: 模板参数名 Type 的非法重复使用
template <class Type, class Type>
     Type min( Type, Type );
```

但是，模板参数名可以在多个函数模板声明或定义之间被重复使用：

```
// ok: 名字 Type 在不同模板之间重复使用
template <class Type>
     Type min( Type, Type );

template <class Type>
     Type max( Type, Type );
```

一个模板的定义和多个声明所使用的模板参数名无需相同。例如，下列三个 min()的声明都指向同一个函数模板：

```
// 三个 min() 的声明都指向同一个函数模板
// 模板的前向声明
template <class T> T min( T, T );
template <class U> U min( U, U );

// 模板的真正定义
template <class Type>
     Type min( Type a, Type b ) { /* ... */ }
```

模板参数在函数参数表中可以出现的次数没有限制。在下面的例子中，Type 用来表示两个不同函数参数的类型：

```
#include <vector>

// ok: 在模板函数的参数表中多次使用 Type
template <class Type>
     Type sum( const vector<Type> &, Type );
```

如果一个函数模板有一个以上的模板类型参数，则每个模板类型参数前面都必须有关键字 class 或 typename。

```
// ok: 关键字 typename 和 class 可以混用
template <typename T, class U>
     T minus( T*, U );

// 错误: 必须是 <typename T, class U> 或 <typename T, typename U>
template <typename T, U>
     T sum( T*, U );
```

在函数模板参数表中，关键字 typename 和 class 的意义相同，可以互换使用。它们两个都可以被用来声明同一模板参数表中的不同模板类型参数［就如前面的函数模板 minus()所做的］。看起来，好像用 typename 而不是 class 来指派模板类型参数更为直观，毕竟，关键字 typename 名字能更清楚地表明后面的名字是个类型名。但是，关键字 typename 是最近才被

加入到标准 C++中的，早期的程序员可能更习惯使用关键字 class。（更不用说关键字 class 比 typename 要短一些，人们总是希望少打几个字嘛……）

通过将关键字 typename 加入到 C++中，使得我们可以对模板定义进行分析。这个话题有些过于高深，我们只简要地解释为什么需要关键字 typename。对于想了解更多内容的读者，建议阅读 Stroustrup 的书《Design and Evolution of C++》。

为了分析模板定义，编译器必须能够区分出是类型以及不是类型的表达式。对于编译器来说，它并不总是能够区分出模板定义中的哪些表达式是类型。例如，如果编译器在模板定义中遇到表达式 Parm::name，且 Parm 这个模板类型参数代表了一个类，那么 name 引用的是 Parm 的一个类型成员吗？

```
template <class Parm, class U>
     Parm minus( Parm* array, U value )
{
     Parm::name * p;   // 这是一个指针声明还是乘法？乘法
}
```

编译器不知道 name 是否为一个类型，因为它只有在模板被实例化之后才能找到 Parm 表示的类的定义。为了让编译器能够分析模板定义，用户必须指示编译器哪些表达式是类型表达式。告诉编译器一个表达式是类型表达式的机制是在表达式前加上关键字 typename。例如如果我们想让函数模板 minus()的表达式 Parm::name 是个类型名，因而使整个表达式是一个指针声明，我们应如下修改：

```
template <class Parm, class U>
     Parm minus( Parm* array, U value )
{
     typename Parm::name * p;   // ok: 指针声明
}
```

关键字 typename 也可以被用在模板参数表中，以指示一个模板参数是一个类型。

如同非模板函数一样，函数模板也可以被声明为 inline 或 extern。应该把指示符放在模板参数表后面，而不是在关键字 template 前面。

```
// ok: 关键字跟在模板参数表之后
template <typename Type>
     inline
     Type min( Type, Type );

// 错误: inline 指示符放置的位置错误
inline
template <typename Type>
     Type min( Array<Type>, int );
```

练习 10.1

指出下列函数模板定义中哪些是错误的，并将其改正。

```
(a)  template <class T, U, class V>
     void foo( T, U, V );

(b)  template <class T>
```

```
            T foo( int *T );
(c)  template <class T1, typename T2, class T3>
            T1 foo( T2, T3 );
(d)  inline template <typename T>
            T foo( T, unsigned int* );
(e)  template <class myT, class myT>
            void foo( myT, myT );
(f)  template <class T>
            foo( T, T );
(g)  typedef char Ctype;
     template <class Ctype>
            Ctype foo( Ctype a, Ctype b );
```

练习 10.2

下列模板重复声明中哪些是错的？为什么？

```
(a)  template <class Type>
            Type bar( Type, Type );
     template <class Type>
            Type bar( Type, Type );

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

练习 10.3

将 7.3.3 小节中给出的函数 putValues()重写为模板函数。并且对函数模板进行参数化，使它有两个模板参数（一个是数组元素的类型，另一个是数组的长度）以及一个函数参数，该函数参数是一个数组的引用。同时给出函数模板定义。

10.2 函数模板实例化

函数模板指定了怎样根据一组或更多实际类型或值构造出独立的函数。这个构造过程被称为模板实例化（template instantiation）。这个过程是隐式发生的，它可以被看作是函数模板调用或取函数模板的地址的副作用。例如，在下面的程序中，min()被实例化两次：一次是针对 5 个 int 的数组类型，另一次是针对 6 个 double 的数组类型。

```
// 函数模板 min() 的定义
// 有一个类型参数 Type 和一个非类型参数 size
```

```
template <typename Type, int size>
     Type min( Type (&r_array)[size] )
{
     Type min_val = r_array[0];
     for ( int i = 1; i < size; ++i )
          if ( r_array[i] < min_val )
               min_val = r_array[i];
     return min_val;
}

// size 没有指定——ok
// size = 初始化表中的值的个数
int ia[] = { 10, 7, 14, 3, 25 };
double da[6] = { 10.2, 7.1, 14.5, 3.2, 25.0, 16.8 };

#include <iostream>
int main()
{
     // 为 5 个 int 的数组实例化 min()
     // Type => int, size => 5
     int i = min( ia );
     if ( i != 3 )
          cout << "??oops: integer min() failed\n";
     else cout << "!!ok: integer min() worked\n";

     // 为 6 个 double 的数组实例化 min()
     // Type => double, size => 6
     double d = min( da );
     if ( d != 3.2 )
          cout << "??oops: double min() failed\n";
     else cout << "!!ok: double min() worked\n";

     return 0;
}
```

调用

```
int i = min( ia );
```

被实例化为下面的 min()的整型实例，这里 Type 被 int、size 被 5 取代：

```
int min( int (&r_array)[5] )
{
     int min_val = r_array[0];
     for ( int ix = 1; ix < 5; ++ix )
          if ( r_array[ix] < min_val )
               min_val = r_array[ix];

     return min_val;
}
```

类似地，调用

```
double d = min( da );
```

也实例化了 min()的实例，这里 Type 被 double，size 被 6 取代。

类型参数 Type 和非类型参数 size 都被用作函数参数。为了判断用作模板实参的实际类型和值，编译器需要检查函数调用中提供的函数实参的类型。在我们的例子中，ia 的类型（即 5 个 int 的数组）和 da 的类型（即 6 个 double 的数组）被用来决定每个实例的模板实参。用函数实参的类型来决定模板实参的类型和值的过程被称为**模板实参推演**（template argument deduction）。我们将在下节更详细地介绍模板实参推演。我们也可以不依赖模板实参推演过程，而是显式地指定模板实参。我们将在 10.4 节了解怎样实现这种方式。）

函数模板在它被调用或取其地址时被实例化。在下面的例子中，指针 pf 被函数模板实例的地址初始化。编译器通过检查 pf 指向的函数的参数类型来决定模板实例的实参。

```
template <typename Type, int size>
     Type min( Type (&p_array)[size] ) { /* ... */ }

// pf 指向 int min( int (&)[10] )
int (*pf)(int (&)[10]) = &min;
```

pf 的类型是指向函数的指针，该函数有一个类型为 int(&)[10]的参数。当 min()被实例化时，该参数的类型决定了 Type 的模板实参的类型和 size 的模板实参的值。Type 的模板实参为 int，size 的模板实参为 10。被实例化的函数是 min(int(&)[10])，指针 pf 指向这个模板实例。

在取函数模板实例的地址时，必须能够通过上下文环境为一个模板实参决定一个惟一的类型或值。如果不能决定出这个惟一的类型或值，就会产生编译时刻错误。例如：

```
template <typename Type, int size>
     Type min( Type (&r_array)[size] ) { /* ... */ }
typedef int (&rai)[10];
typedef double (&rad)[20];

void func( int (*)(rai) );
void func( double (*)(rad) );

int main() {
     // 错误：哪一个 min() 的实例?
     func( &min );
}
```

因为函数 func()被重载了，所以编译器不可能通过查看 func()的参数类型，来为模板参数 Type 决定惟一的类型，以及为 size 的模板实参决定一个惟一值。调用 func()无法实例化下面的任何一个函数：

```
min( int (*)(int(&)[10]) )
min( double (*)(double(&)[20]) )
```

因为不可能为 func()指出一个惟一的实参的实例，所以在该上下文环境中取函数模板实例的地址会引起编译时刻错误。

如果我们用一个强制类型转换显式地指出实参的类型则可以消除编译时刻错误：

```
int main() {
     // ok: 强制转换指定实参类型
     func( static_cast< double(*)(rad) >(&min) );
}
```

更好的方案是用显式模板实参，这一点我们将在 10.4 节中说明。

10.3 模板实参推演 ※

当函数模板被调用时，对函数实参类型的检查决定了模板实参的类型和值、这个过程被称为**模板实参推演**（template argument deduction）。

函数模板 min()的函数参数是一个引用，它指向了一个 Type 类型的数组：

```
template <class Type, int size>
    Type min( Type (&r_array)[size] ) { /* ... */ }
```

为了匹配函数参数，函数实参必须也是一个表示数组类型的左值。下面的调用是个错误，因为 pval 是 int*类型而不是 int 数组类型的左值。

```
void f( int pval[9] ) {
    // 错误: Type (&)[] != int*
    int jval = min( pval );
}
```

在模板实参推演期间决定模板实参的类型时，编译器不考虑函数模板实例的返回类型。例如，对于如下的 min()调用：

```
double da[8] = { 10.3, 7.2, 14.0, 3.8, 25.7, 6.4, 5.5, 16.8 };
int i1 = min( da );
```

min()的实例有一个参数，它是一个指向 8 个 double 的数组的指针。出该实例返回的值的类型是 double 型。该返回值先被转换成 int 型，然后再用来初始化 i1。即使调用 min()的结果被用来初始化一个 int 型的对象，也不会影响模板实参的推演过程。

要想成功地进行模板实参推演，函数实参的类型不一定要严格匹配相应函数参数的类型。下列三种类型转换是允许的：左值转换、限定转换和到一个基类（该基类根据一个类模板实例化而来）的转换。让我们依次来看一看。

左值转换包括从左值到右值的转换、从数组到指针的转换或从函数到指针的转换（这些转换在 9.3 节中介绍）。为说明左值转换是怎样影响模板实参推演过程的，让我们考虑函数 min2()，它有一个名为 Type 的模板参数以及两个函数参数。min2()的第一个函数参数是一个 Type*型的指针。而 size 则不再像在 min()中定义的是个模板参数。size 变成一个函数参数当 min2()被调用时，我们必须显式地为它指定一个函数实参值：

```
template <class Type>
// 第一个参数是 Type*
Type min2( Type* array, int size )
{
    Type min_val = array[0];
    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    return min_val;
}
```

我们可以用 4 个 int 的数组来作为第一个实参调用 min2()，如下：

```
int ai[4] = { 12, 8, 73, 45 };
int main() {
    int size = sizeof (ai) / sizeof (ai[0]);

    // ok: 从数组到指针的转换
    min2( ai, size );
}
```

函数实参 ai 的类型是 4 个 int 的数组，虽然这与相应的函数参数类型 Type*并不严格匹配。但是因为允许从数组到指针的转换，所以实参 ai 在模板实参 Type 被推演之前被转换成 int*型。Type 的模板实参接着被推演为 int，最终被实例化的函数模板是 min2(int*,int)。

限定修饰转换把 const 或 volatile 限定修饰符加到指针上（限定修饰转换在 9.3 节中介绍）。为说明限定修饰转换是怎样影响模板实参推演过程的，我们考虑函数 min3()，它的第一个函数参数的类型是 const Type*：

```
template <class Type>
    // 第一个参数是 const Type*
    Type min3( const Type* array, int size ) {
        // ...
    }
```

我们可以用 int*型的第一个参数调用 min3()，如下：

```
int *pi = &ai;

// ok: 到 const int* 的限定修饰转换
int i = min3( pi, 4 );
```

函数实参 pi 的类型是 int 指针，虽然与相应的函数参数类型 const Type*并不完全匹配。但是因为允许限定修饰转换，所以函数实参在模板实参被推演之前，就先被转换为 const Type*型了。然后 Type 的模板实参被椎演为 int，被实例化的函数模板是 min3(const int*, int)。

现在，再让我们来看看到一个基类（该基类根据一个类模板实例化而来）的转换。如果函数参数的类型是一个类模板，且如果实参是一个类，它有一个从被指定为函数参数的类模板实例化而来的基类，则模板实参的推演就可以进行。为说明这个转换，我们使用一个新的被称为 min4()的函数模板，它有一个类型为 Array<Type>&的参数（这里的 Array 是在 2.5 节中定义的类模板）。（第 16 章将给出类模板的完全讨论）。

```
template <class Type>
    class Array { /* ... */ };

template <class Type>
    Type min4( Array<Type>& array )
{
    Type min_val = array[0];
    for ( int i = 1; i < array.size(); ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    return min_val;
}
```

我们可以用类型为 ArrayRC<int>的第一个实参调用 min4()（ArrayRC 也是第 2 章定义的

类模板。类继承将在 17 章和 18 章中讨论）。如下：

```
template <class Type>
    class ArrayRC : public Array<Type> { /* ... */ };

int main() {
    ArrayRC<int> ia_rc( ia, sizeof(ia)/sizeof(int) );
    min4( ia_rc );
}
```

函数实参 ia_rc 的类型是 ArrayRC<int>，它与相应的函数参数类型 Array<Type>&并不完全匹配。因为类 ArrayRC<int>有一个 Array<int>的基类，而 Array<int>是一个从被指定为函数参数的类模板实例化而来的类，并且派生类类型的函数实参还可以被用来推演一个模板实参，所以函数实参 ArrayRC<int>在模板实参被推演之前首先被转换成 Array<int>型，然后 Type 的模板实参再被推演为 int，被实例化的函数模板是 min4(Array<int>&)。

多个函数实参可以参加同一个模板实参的推演过程。如果模板参数在函数参数表中出现多次，则每个推演出来的类型都必须与根据模板实参推演出来的第一个类型完全匹配。例如

```
template <class T> T min5( T, T ) { /* ... */ }
unsigned int ui;

int main() {
    // 错误: 不能实例化 min5( unsigned int, int )
    // 必须是: min( unsigned int, unsigned int ) 或
    // min( int, int )
    min5( ui, 1024 );
}
```

mins()的函数实参必须类型相同（要么都是 int，要么都是 unsigned int)，这是因为模板参数 T 必须被绑定在一个类型上。从第一个函数实参推演出的 T 的模板实参是 int，而从第二个函数实参推演出的是 unsigned int。因为对于两个函数实参，模板实参 T 的类型被推演成不同类型，所以模板实参推演将失败，并且模板实例化也会出错误。（一种解决办法是在调用 min5()时显式指定模板实参。我们将在 10.4 节介绍怎样实现它。）

这些可能的类型转换的限制只适用于参加模板实参推演过程的函数实参。对于所有其他实参，所有的类型转换都是允许的。下面的函数模板 sum()有两个参数。针对第一个参数的实参 op1 参与模板实参推演过程，而针对第二个参数的实参 op2 则没有参与。

```
template <class Type>
    Type sum( Type op1, int op2 ) { /* ... */ }
```

因为第二个实参不参与模板实参推演过程，所以当函数模板 sum()的实例被调用时。可以在第二个实参上应用任何类型转换。（9.3 节描述了可以应用在函数实参上的类型转换。）例如：

```
int ai[] = { ... };
double dd;

int main() {
    // sum( int, int ) 被实例化
    sum( ai[0], dd );
}
```

第一个函数实参 dd 的类型与相应的函数参数类型 int 不匹配。但是，对函数模板 sum() 实例的调用不是错的，这是因为第二个实参的类型是固定的，不依赖于模板参数。对于该调用，函数 sum(int,int)被实例化。实参 dd 被通过浮点—有序标准转换转换成类型 int。

所以模板实参推演的通用算法如下：

1．依次检查每个函数实参，以确定在每个函数参数的类型中出现的模板参数。

2．如果找到模板参数，则通过检查函数实参的类型，推演出相应的模板实参。

3．函数参数类型和函数实参类型不必完全匹配。下列类型转换可以被应用在函数实参上，以便将其转换成相应的函数参数的类型：

- 左值转换。
- 限定修饰转换。
- 从派生类到基类类型的转换。假定函数参数具有形式 T<args>、T<args>&或 T<args>*，则这里的参数表 args 至少含有一个模板参数。

4．如果在多个函数参数中找到同一个模板参数，则从每个相应函数实参推演出的模板实参必须相同。

练习 10.4

指出在模板实参推演过程中涉及到的函数实参时两种可行的类型转换。

练习 10.5

已知下列模板定义：

```
template <class Type>
     Type min3( const Type* array, int size ) { /* ... */ }
template <class Type>
     Type min5( Type p1,Type p2 ) { /* ... */ }
```

下列哪些调用是错误的？为什么？

```
double dobj1, dobj2;
float fobj1, fobj2;
char cobj1, cobj2;
int ai[5] = { 511, 16, 8, 63, 34 };
(a) min5( cobj2, 'c' );
(b) min5( dobj1, fobj1 );
(c) min3( ai, cobj1 );
```

10.4 显式模板实参 ※

在某些情况下编译器不可能推演出模板实参的类型。如在上节函数模板 min5()的例子中所看到的，如果模板实参推演过程为同一模板实参推演出两个不同的类型，则编译器会给出一个错误，指出模板实参推演失败。

在这种情况下，我们需要改变模板实参推演机制，并使用**显式指定**（explicitly specify）模板实参。模板实参被显式指定在逗号分隔的列表中，用尖括号（<>，一个小于号和一个

大于号）括起来，紧跟在函数模板实例的名字后面。例如，在我们前面使用的 min5()中，假定希望 T 的模板实参是 unsigned int，则函数模板实例 min5()的调用可以重写如下：

```
// min5( unsigned int, unsigned int ) 被实例化
min5< unsigned int >( ui, 1024 );
```

在这种情况下，模板实参表<unsigned int>显式地指定了模板实参的类型。因为模板实参已知，所以函数调用不再是一个错误。

注意，在调用函数 min5()时，第二个函数实参是 1024，它的类型是 int。因为第二个函数参数的类型通过显式模板实参已被固定为 unsigned int，所以第二个函数实参通过有序标准转换被转换成类型 unsigned int。

在上节中我们看到，能在函数实参上进行的只是类型转换是有限的。从 int 到 unsigned int 的有序标准转换就不允许。但是当模板实参被显式指定时，就没有必要推演模板实参了。函数参数的类型已经固定。当函数模板实参被显式指定时，把函数实参转换成相应函数参数的类型可以应用任何隐式类型转换。

除了允许在函数实参上的类型转换，显式模板参数还为解决其他的程序设计问题提供了方案。考虑下面的问题。我们希望定义一个名为 sum()的函数模板，以便从该模板实例化的函数可以返回某一种类型的值，该类型足够大，可以装下两种以任何顺序传递来的任何类型的两个值的和。我们该怎样做呢？应该指定 sum()的返回类型吗？

```
// 以 T 或 U 作为返回类型?
template <class T, class U>
    ??? sum( T, U );
```

在我们的例子中，答案是两个参数都不用。因为使用它们中的任何一个都会导致在某点上失败：

```
char ch; unsigned int ui;

// T 和 U 都不用作返回类型
sum( ch, ui );    // ok: U sum( T, U );
sum( ui, ch );    // ok: T sum( T, U );
```

一种方案是通过引入第三个模板参数来指明函数模板的返回类型。

```
// T1 不出现在函数模板参数表中
template <class T1, class T2, class T3>
    T1 sum( T2, T3 );
```

因为返问类型可能与函数实参类型不同，所以 T1 在函数参数表中不再被提起。这是一个潜在的问题，因为 T1 的模板实参不能从函数实参中被推演出来。但是，如果在 sum()的一个实例调用中给出一个显式模板实参。我们就能避免编译器错误地指出 T1 的模板实参不能被推演出来。例如：

```
typedef unsigned int ui_type;
ui_type calc( char ch, ui_type ui ) {

    // ...

    // 错误: T1 不能被推演出来
    ui_type loc1 = sum( ch, ui );
```

```
        // ok: 模板实参被显式指定
        // T1 和 T3 是 unsigned int, T2 是 char
        ui_type loc2 = sum< ui_type, char, ui_type >( ch, ui );
    }
```

我们真正期望的是，为 T1 指定一个显式模板实参，而省略 T2 和 T3 的显式模板实参。T2、T3 的模板实参可以从该调用的函数实参中推演出来。

在显式特化（explicit specification）中，我们只需列出不能被隐式推演的模板实参，如同缺省实参一样，我们只能省略尾部的实参。例如：

```
    // ok: T3 是 unsigned int
    // T3 从 ui 的类型中推演出来
    ui_type loc3 = sum< ui_type, char >( ch, ui );

    // ok: T2 是 char, T3 是 unsigned int
    // T2 和 T3 从 pf 的类型中推演出来
    ui_type (*pf)( char, ui_type ) = &sum< ui_type >;

    // 错误: 只能省略尾部的实参
    ui_type loc4 = sum< ui_type, , ui_type >( ch, ui );
```

在其他情形下，编译器不可能从使用函数模板实例的上下文中推演出模板实参。没有显式模板实参。在这种上下文环境中就不可能使用函数模板实例。我们必须意识到需要支持这些清形，这导致了标准 C++对显式模板实参提供了支持。在下面的例子中，sum()实例的地址被取出，并作为重载函数 manipulate()调用的实参被传递。如在 10.2 节中所述，想只通过查看 manipulate()的参数表就能选择出作为实参传递的 sum()实例，这是不可能的。sum()的两个不同实例都可以被实例化，并满足该调用。但 manipulate()的调用是二义的。消除调用二义性的一个方案是提供一个显式强制类型转换，而更好的方案是使用显式模板实参。显式模板实参指明 sum()的哪个实例被使用，以及哪个 manipulate()被调用。例如：

```
    template <class T1, class T2, class T3>
    T1 sum( T2 op1, T3 op2 ) { /* ... */ }
    void manipulate( int (*pf)( int,char ) );
    void manipulate( double (*pf)( float,float ) );
    int main( )
    {
        // 错误: 哪一个 sum 的实例?
        // int sum( int, char ) 还是
        // double sum( float, float ) ?
        manipulate( &sum );

        // 取实例: double sum( float, float )
        // 调用: void manipulate( double (*pf)( float, float ) );
        manipulate( &sum< double, float, float > );
    }
```

我们必须指出，显式模板实参应该只被用在完全需要它们来解决二义性，或在模板实参不能被推演出来的上下文中使用模板实例时。首先，让编译器来决定模板实参的类型和值是比较容易的。其次，如果我们通过修改程序中的声明来改变在函数模板实例调用中的函数实参的类型，则编译器会自动用不同的模板实参实例化函数模板，而无需我们做任何事情。另

一方面，如果我们指定了显式模板参数，则必须检查显式模板实参对于函数实参的新类型是否仍然合适。所以建议在可能的时候省略显式模板实参。

练习 10.6

指出两种必须使用显式模板实参的情形。

练习 10.7

已知下面 sum()的模板定义

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

下列哪些调用是错误的，为什么？

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

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

10.5 模板编译模式 ※

函数模板的定义可用来作为一个无限个函数实例集合定义的规范描述（prescription）。模板本身不能定义任何函数。例如，当编译器实现看到下面的模板定义时

```
template <typename Type>
     Type min( Type t1, Type t2 )
{
     return t1 < t2 ? t1 : t2;
}
```

它就保存了 min()的内部表示形式，但是不会使任何其他事情发生。后来，当它看到 min()被实际使用时，如

```
int i, j;
double dobj = min( i, j );
```

才根据模板定义为 min()实例化一个整型的定义。

这带来了几个问题：如果希望编译器能够实例化函数模板，那么函数模板 min()的定义必须在实例被调用之前就可见吗？例如，在上面的例子中，min()的整型实例被用在 dobj 的定义中，那么在此之前函数模板 min()的定义必须先出现吗？我们把函数模板定义放在头文件中（就好像对内联函数定义的做法一样），在使用函数模板实例的地方包含它们？或者我们只在头文件中给出函数模板声明，而把模板定义放在文本文件中（就好像对非内联函数的做法一样）？

为了回答这些问题，我们必须解释 **C++模板编译模式**（template compilation model），它

指定了对于定义和使用模板的程序的组织方式的要求。C++支持两种模板编译模式：**包含模式**（Inclusion Model）和**分离模式**（Separation Model）。本节余下部分将分别描述这两种模式，并具体说明它们的用法。

10.5.1 包含编译模式

在包含编译模式下，我们在每个模板被实例化的文件中包含函数模板的定义，并且往往把定义放在头文件中，像对内联函数所做的那样。我们已经把这种模式选为本书使用的模式，例如：

```
// model1.h
// 包含模式：模板定义放在头文件中
template <typename Type>
     Type min( Type t1, Type t2 ) {
          return t1 < t2 ? t1 : t2;
}
```

在每个使用 min()实例的文件中都包含了该头文件，例如：

```
// 在使用模板实例之前包含模板定义
#include "model1.h"
int i, j;
double dobj = min( i, j );
```

该头文件可以被包含在许多程序文本文件中。这意味着编译器必须在每个调用该实例的文件中实例化 min()的整型实例吗？不。该程序必须表现得好像 min()的整型实例只被实例化一次。但是，真正的实例化动作发生在何时何地，要取决于具体的编译器实现。现在就我们所关心的来说，我们只需要知道 min()的整型实例在程序中的某个地方被实例化。（如我们在本节结束时将看到的，用显式实例化声明可指定在何时何地进行模板实例化。这样的声明有时候必须在产品开发的后期被使，以来改善应用程序的性能。）

在头文件中提供函数模板定义有几个缺点。函数模板体（body）描述了实现细节，对于这些细节，用户可能想忽略，或者我们希望隐藏起来不让用户知道。实际上，如果函数模板的定义非常大，那么在头文件中给出的细节层次有可能是不可接受的。而且，在多个文件之间编译相同的函数模板定义增加了不必要的编译时间。分离编译模式允许我们分离函数模板的声明和定义，下面让我们看一下怎样使用它。

10.5.2 分离编译模式

在分离编译模式下，函数模板的声明被放在头文件中。在这种模式下，函数模板声明和定义的组织方式与程序中的非内联函数的声明和定义组织方式相同。例如：

```
// model2.h

// 分离模式：只提供模板声明
template <typename Type> Type min( Type t1, Type t2 );

// model2.C
// the template definition
export template <typename Type>
```

```
Type min( Type t1, Type t2 ) { /* ...*/ }
```

使用函数模板 min()实例的程序只需在使用该实例之前包含这个头文件：

```
// user.C
#include "model2.h"

int i, j;
double d = min( i, j ); // ok: 用法，需要一个实例
```

即使 min()的模板定义在 user.C 中不可见，仍然可以在这个文件中调用模板实例 min(int,int)。但是，为了实现它，模板 min()必须以一种特殊的方式被定义。你知道怎样做吗？如果仔细看一下定义了函数模板 min()的文件 model2.C，你就会注意到在模板定义中有一个关键字 export。模板 min()被定义成一个可导出的（exported）模板。关键字 export 告诉编译器在生成被其他文件使用的函数模板实例时可能需要这个模板定义。编译器必须保证，在生成这些实例时，该模板定义是可见的。

我们通过在模板定义中的关键字 template 之前加上关键字 export，来声明一个可导出的函数模板。当函数模板被导出时，我们就可以在任意程序文本文件中使用模板的实例，而我们所需要做的就是在使用之前声明该模板。如果省略了模板 min()定义中的关键字 export，则编译器实现可能不能实例化函数模板 min()的整型实例，而我们将不能正确链接我们的程序。

注意，有些编译器实现可能不要求用关键字 export。有些实现可能支持下列语言扩展：非导出的函数模板定义可能只出现在一个程序文本文件中，在其他程序文本文件中用到的实例仍然被正确地实例化。但是，这种行为只是一个扩展。如果在模板被实例化之前只有函数模板的声明在程序文本文件中可见，那么，标准 C++要求用户把函数模板定义标记为 export。

关键字 export 不需要出现在头文件的模板声明中。在 model2.h 中的 min()的声明中没有指定关键字 export。此关键字也可以出现在该声明中，但不是必需的。

在程序中，一个函数模板只能被定义为 export 一次。不幸的是，因为编译器每次只处理一个文件，所以它不能检测到一个函数模板在多个文本文件中被定义为 export 的情况。如果发生了这样的事情，下列行为就有可能随之发生：

1．可能产生一个链接错误，指出函数模板在多个文件中被定义。

2．编译器可能不只一次地为同一个模板实参集合实例化该函数模板，由于函数模板实例的重复定义，这会引起链接错误。

3．编译器可能用其中一个 export 函数模板定义来实例化函数模板，而忽略其他定义。

所以，在程序中提供多个 export 函数模板的定义不一定会产生错误。我们必须小心谨慎地组织程序，以便把 export 函数模板定义只放在一个程序文本文件中。

分离模式使我们能够很好地将函数模板的接口同其实现分开，进而组织好程序，以便把函数模板的接口放到头文件中，而把实现放在文本文件中。但是，并不是所有的编译器都支持分离模式，即使支持也未必总能支持得很好。支持分离模式需要更复杂的程序设计环境，所以它们不能在所有 C++编译器实现中提供。[21]

对于本书的目的而言，因为模板例子非常小，而且我们希望这些例子在许多 C++实现中

[21] 本书的姊妹篇《Inside the C++ Object Model》，描述了一个 C++编译器“the Edison Design Group compiler”支持的模板实例化机制。该书的中文简体版已经出版。

都能够方便地被编译，所以我们只使用包含模式。

10.5.3 显式实例化声明

当我们使用包含模式时，每个使用模板实例的程序文本文件都要包含函数模板的定义。我们已经看到：尽管程序不能确切地知道编译器在何时何地实例化函数模板，但是它必须表现得好像对一个特定的模板实参集合只实例化一次模板。在实际中，有些编译器（尤其是老的 C++编译器）对特殊的模板实参集合会多次实例化函数模板。在这种模式下，程序会选择这些实例中的一个作为最终的实例（当程序被链接或在某个预链接阶段）。而其他实例只是被简单地忽略掉。

无论函数模板被实例化一次还是多次，程序结果都不会受到影响，因为最终只有一个模板实例被程序使用。但是，如果函数模板被多次实例化，则程序的编译时间性能可能会受到很大影响。如果应用程序由许多文件构成，并且所有这些文件中的模板都被实例化，那么编译应用程序所需要的时间会显著地增加。

早期编译器的实例化问题使得模板用起来非常困难。为了解决这个问题，标准 C++提供了显式实例化声明来帮助程序员控制模板实例化发生的时间。

在显式实例化声明中，关键字 template 后面是函数模板实例的声明，其中显式地指定了模板实参。在下面的例子中，提供了 sum(int*,int)的显式实例化声明。

```
template <typename Type>
     Type sum( Type op1, int op2 ) { /* ... */ }

// 显式实例化声明
template int* sum< int* >( int*, int );
```

该显式实例化声明要求用模板实参 int*实例化模板 sum()。对于给定的函数模板实例。显式实例化声明在一个程序中只能出现一次。

在显式实例化声明所在的文件中，函数模板的定义必须被给出。如果该定义不可见，则该显式实例化声明是错误的。

```
#include <vector>
template <typename Type>
     Type sum( Type op1, int op2 ); // declaration only

// 声明一个 typedef 引用 vector< int >
typedef vector< int > VI;

// 错误: sum() 没有定义
template VI sum< VI >( VI , int );
```

当一个显式实例化声明在程序文本文件中出现时，在其他使用该函数模板实例的文件中又发生了什么事情？我们怎样告诉编译器，一个显式实例化声明已在另一个文件中出现，而在程序其他文件使用该模板函数时不能再对它实例化？

显式实例化声明是与另外一个编译选项联合使用的，该选项压制了程序中模板的隐式实例化。选项的名称随着编译器不同而小同。例如，对 IBM 编译器 Visual Age for C++ for Windows 版本 3.5，压制模板隐式实例化的选项为/ft-。当我们用这个选项编译应用程序时，

编译器假定我们将会用显式实例化声明处理模板实例，所以它不会隐式地实例化应用程序用到的模板。

当然，如果我们不为一个函数实例提供显式实例化声明，则在编译程序时指定选项/ft-就会产生一个链接错误，认为缺少函数模板实例化的定义。在这种情况下，模板不会被隐式地实例化。

练习 10.8

指出 C++支持的两种模板编译模式，并解释在这些模板编译模式下怎样组织含有函数模板定义的程序。

练习 10.9

给出下列 sum()的模板定义：

```
template <typename Type>
     Type sum( Type op1, char op2 );
```

怎样为 string 类型的模板实参声明一个显式实例化声明？

10.6 模板显式特化 ※

我们并不总是能够写出对所有可能被实例化的类型都是最合适的函数模板。在某些情况下，我们可能想利用类型的某些特性，来编写一些比模板实例化的函数更高效的函数。在有些时候，一般性的模板定义对于某种类型来说并不适用。例如，假设我们有函数模板 max()的定义：

```
// 通用的模板定义
template <class T>
     T max( T t1, T t2 ) {
          return (t1 > t2 ? t1 : t2);
}
```

如果函数模板用 const char*型的模板实参实例化，并且我们还想让每个实参都被解释为 C 风格的字符串，而不是字符的指针，则通用模板定义给出正确的语义就不正确了。为了获得正确的语义，我们必须为函数模板实例化提供特化的定义。

在**模板显式特化定义**（explicit specialization definition）中，先是关键字 template 和一对尖括号（<>，一个小于号和一个大于号），然后是函数模板特化的定义。该定义指出了模板名、被用来特化模板的模板实参，以及函数参数表和函数体。在下面的例子中，为 max(const char*, const char*)定义了一个显式特化：

```
#include <cstring>

// const char* 显式特化:
// 覆盖了来自通用模板定义的实例

typedef const char *PCC;
template<> PCC max< PCC >( PCC s1, PCC s2 ) {
```

```
    return ( strcmp( s1, s2 ) > 0 ? s1 : s2 );
}
```

由于有了这个显式特化，当在程序中调用函数 max(const char*,const char*)时，模板不会用类型 const char*来实例化。对所有用两个 const char*型实参进行调用的 max()，都会调用这个特化的定义。而对于其他的调用，根据通用模板定义实例化一个实例，然后再调用它。这些函数可能的调用如下：

```
#include <iostream>

// 函数模板 max() 的定义以及对 const char* 的特化

int main() {
    // 调用实例: int max< int >( int, int );
    int i = max( 10, 5 );

    // 调用显式特化: const char* max< const char* >( const char*, const char* );
    const char *p = max( "hello", "world" );

    cout << "i: " << i << " p: " << p << endl;
    return 0;
}
```

我们也可以声明一个函数模板的显式特化而不定义它。例如，函数 max(const char*,const char*)的显式特化可以被声明如下：

```
// 函数模板显式特化的声明
template<> PCC max< PCC >( PCC, PCC );
```

在声明或定义函数模板显式特化时，我们不能省略显式特化声明中的关键字 template 及其后的尖括号。类似地，函数参数表也不能从特化声明中省略掉。

```
// 错误: 无效的特化声明

// 缺少 template<>
PCC max< PCC >( PCC, PCC );

// 缺少函数参数表
template<> PCC max< PCC >;
```

但是，如果模板实参可以从函数参数中推演出来，则模板实参的显式特化可以从显式特化声明中省略。

```
// ok: 模板实参 const char* 可以从参数类型中推演出来
template<> PCC max( PCC , PCC );
```

在下面的例子中，函数模板 sum()被显式特化：

```
template <class T1, class T2, class T3>
    T1 sum( T2 op1, T3 op2 );

// 显式特化声明

// 错误: T1 的模板实参不能被推演出来
//       它必须显式指定
```

```
template<> double sum( float, float );

// ok: T1 的实参被显式指定
// T2 和 T3 可以从 float 推演出来
template<> double sum<double>( float, float );

// ok: 所有实参都显式指定
template<> int sum<int,char,char>( char , char );
```

省略显式特化声明中的 template<>并不总是错的。例如：

```
// 通用模板定义
template <class T>
T max( T t1, T t2 ) { /* ... */ }

// ok: 普通函数定义
const char* max( const char*, const char* );
```

但是，max()的声明并没有声明函数模板特化。它只是用与模板实例相匹配的返回值和参数表声明了一个普通函数。声明一个与模板实例相匹配的普通函数并不是个错误。

为什么我们要声明一个与模板实例相匹配的普通函数而不声明一个显式特化呢？如在 9.3 节所见到的，如果该实参参与模板实参的推演过程，则只能应用有限的一些类型转换把函数模板实例的实参转换成相应的函数参数类型。函数模板被显式特化的情形也是这样，只有 10.3 节描述的那些类型转换才可以被应用在函数模板显式特化的实参上。显式特化并没有帮助我们越过类型转换的限制。如果想进行该类型转换集合之外的转换，那么就必须定义一个普通函数而不是一个函数模板的特化。10.8 节将更详细地介绍，并给出怎样解析一个与普通函数和函数模板实例都匹配的调用。

即使函数模板显式特化所指定的函数模板只有声明而没有定义，我们仍然可以声明函数模板显式特化。在前面的例子中，函数模板 sum()在被特化之前只是被声明了一下。尽管不需要函数定义，但是模板声明也还是需要的。在名字 sum()被特化之前，编译器必须知道它是个模板。

在源文件中，使用函数模板显式特化之前，必须先进行声明。例如：

```
#include <iostream>
#include <cstring>

// 通用模板定义
template <class T>
T max( T t1, T t2 ) { /* ... */ }
int main() {
     // const char* max<const char*>( const char*, const char* ) 的实例
     // 使用通用模板定义
     const char *p = max( "hello", "world" );

     cout << " p: " << p << endl;
     return 0;
}
```

```
// 无效程序: const char* 显式特化覆盖了通用模板函数
typedef const char *PCC;
     template<> PCC max< PCC >( PCC s1, PCC s2 ) { /* ... */ }
```

因为上面的例子在声明显式特化之前使用了 max(const char*,const char*)的实例。所以，编译器只好假定该函数需要从通用模板定义中实例化。但是，一个程序不能对相同的模板实参集的同一模板同时有一个显式特化和一个实例。当在程序文本文件中再遇到 max(const char*,const char*)的显式特化时，编译器会提示一个编译时刻错误。

如果程序由一个以上的文件构成，则模板显式特化的声明必须在使用该特化的每个文件中都可见。像下面这样的情况是不允许的：在有些文件中，函数模板被根据通用模板定义实例化，而在其他文件中，对同一模板实参的集合却被特化。考虑下面的例子：

```
// ---- max.h ----
// 通用模板定义
template <class Type>
     Type max( Type t1, Type t2 ) { /* ... */ }

// ---- File1.C ----
#include <iostream>
#include "max.h"
void another();
int main() {
     // const char* max<const char*>( const char*, const char* )
     // 的实例
     const char *p = max( "hello", "world" );

     cout << " p: " << p << endl;
     another();
     return 0;
}

// ---- File2.C ----
#include <iostream>
#include <cstring>
#include "max.h"

// const char* 的模板显式特化
typedef const char *PCC;
     template<> PCC max< PCC >( PCC s1, PCC s2 ) { /*... */ }

void another() {
     // const char* max< const char* >( const char*, const char* )
     // 的显式特化;
     const char *p = max( "hi", "again" );

     cout << " p: " << p << endl;
     return 0;
}
```

上面的程序由两个文件构成。在 File1.C 中，没有显式特化 max(const char*,const char*)

的声明，函数模板被根据通用模板定义实例化。在 File2.C 中声明了显式特化，调用 max("hi", "again")就会调用该显式特化。因为同一程序在一个文件中实例化了函数模板实例 max(const char*,const char*)，而在另一个文件中又调用了这显式特化，所以这个程序是非法的。为了补救这个问题，显式特化的声明必须在文件 File1.C 中 max(const char*,const char*)调用之前被给出。

为了防止出现这样的错误，并确保模板显式特化 max(const char*,const char*)的声明被包含在每个用类型 const char*型实参调用函数模板 max()的文件中，显式特化的声明应该被放在头文件“max.h”中，并在所有使用函数模板 max()的程序中包含这个文件：

```
// ---- max.h ----
// 通用模板定义
template <class Type>
     Type max( Type t1, Type t2 ) { /* ... */ }

// const char* 模板显式特化的声明
typedef const char *PCC;
     template<> PCC max< PCC >( PCC s1, PCC s2 );

// ---- File1.C ----
#include <iostream>
#include "max.h"

void another();
int main() {
     // const char* max<const char*>( const char*, const char* ) 的特化;
     const char *p = max( "hello", "world" );

     //....
}
```

练习 10.10

请定义一个函数模板 count()，以记录数组中某个值出现的次数。写个程序调用它，并按顺序向其传递 double、int 和 char 型的数组，以及引入 count()函数的一个特化模板实例来处理字符串。最后，再重新运行调用该函数模板实例的程序。

10.7 重载函数模板 ※

函数模板可以被重载。例如，下面给出函数模板 min()的三个有效的重载声明：

```
// 类模板 Array 的定义
// (introduced in Section 2.4)
template <typename Type>
     class Array{ /* ... */ };

// min() 的三个函数模板声明
```

```
template <typename Type>
     Type min( const Array<Type>&, int ); // #1

template <typename Type>
     Type min( const Type*, int );        // #2

template <typename Type>
     Type min( Type, Type );              // #3
```

下面的 main()定义说明了这三个 min()声明可以被怎样调用：

```
#include <cmath>

int main()
{
     Array<int> iA(1024);// 类实例
     int ia[1024];

     // Type == int; min( const Array<int>&, int )
     int ival0 = min( iA, 1024 );

     // Type == int; min( const int*, int )
     int ival1 = min( ia, 1024 );

     // Type == double; min( double, double )
     double dval0 = min( sqrt( iA[0] ), sqrt( ia[0] ) );

     return 0;
}
```

当然，成功地声明一组重载函数模板并不能保证它们可以被成功地调用。在调用一个模板实例时，重载的函数模扳可能会导致二义性。下面是将出现这种二义性的一个例子。我们前面曾讲到，对于下列 min5()的模板定义：

```
template <typename T>
     int min5( T, T ) { /* ... */ }
```

即使用不同类型的实参调用 min5()，编译器也不能根据模板定义实例化函数。因为函数实参推演过程为 T 推演出两个不同的类型，所以模板实参推演失败，调用是错误的。

```
int i;
unsigned int ui;

// ok: 为 T 推演出: int
min5( 1024, i );

// 模板实参推演失败
// 为 T 推演出两个不同的类型
min5( i, ui );
```

为了解析第二个调用，我们可以重载 min5()，允许两个不同的实参类型：

```
template <typename T, typename U>
     int min5( T, U );
```

下列函数调用则调用了这个新函数模板的实例：

```
// ok: int min5( int, unsigned int )
min5( i, ui );
```

不幸的是，原先的调用现在已变成二义的了：

```
// 错误：二义性：来自 min5( T, T ) 和 min5( T, U ) 的两个可能的实例
min5( 1024, i );
```

min5()的第二个声明允许两个不同类型的函数实参。但是，它没有要求它们一定是不同的。在这种情况下，T 和 U 都可以是 int 型。对于两个实参类型相同的调用，这两个模板声明都可以被实例化。要指明哪个函数模板比较好、并且消除调用的二义性的惟一方法是显式指定模板实参（关于显式模板实参的讨论见 10.4 节）。例如：

```
// ok: 从 min5( T, U ) 实例化
min5<int, int>( 1024, i );
```

但是，在这种情况下，我们其实可以取消重载函数模板。因为 min5(T,U)处理的调用集是由 min5(T,T)处理的超集，所以应该只提供 min5(T,U)的声明，而 min5(T,T)应该被删除。因此，正如我们在第 9 章开始时所说的，尽管重载是可能的，但是我们在设计重载函数时，仍然必须小心确保重载是必需的。这些设计限制也同样适用于定义重载函数模板。

在某些情况下，即使对于一个函数调用，两个不同的函数模板都可以实例化，但是该函数调用仍然可能不是二义的。已知 sum()的下列两个模板定义，下面就是一种情况：虽然从这两个函数模板的任一个都可以生成一个实例，但是第一个模板定义比较好。

```
template <typename Type>
     Type sum( Type*, int );

template <typename Type>
     Type sum( Type, int );

int ia[1024];

// Type == int ; sum<int>( int*, int ); or
// Type == int*; sum<int*>( int*, int ); ??
int ival1 = sum<int>( ia, 1024 );
```

真让人吃惊，上面的调用居然没有二义性，该模板是用第一个模板定义实例化的。为该实例选择的模板函数是**最特化的**（most specialized）。因此，Type 的模板实参是 int 而不是 int*。

一个模板要比另一个更特化，两个模板必须有相同的名字、相同的参数个数，对于不同类型的相应函数参数，如上面的 T*和 T，一个参数必须能接受另一个模板中相应参数能够接受的实参的超集。例如，对模板 sum(Type*,int)，第一个函数参数只能匹配指针类型的实参。对于模板 sum(Type,int)，第一个函数参数可以匹配指针类型以及任意其他类型的实参。第二个模板接受第一个模极所能够接受的类型的超集，接受更有限的实参集合的模板被称为是更特化的。在我们的例子中，模板 sum(Type*,int)是更特化的，它是为了例子中的函数调用而被实例化的模板。

10.8 考虑模板函数实例的重载解析 ※

如上节所述，函数模板可以被重载，函数模板可以与一个普通非模板函数同名。例如：

```
// 函数模板
template <class Type>
     Type sum( Type, int ) { /* ... */ }
// 普通（非模板）函数
double sum( double, double );
```

当程序调用 sum()时，该调用可以被解析为函数模板的实例，或者被解析为普通函数。到底调用哪个函数取决于这些函数中的哪一个与函数实参类型匹配得最好。在第 9 章中介绍的函数重载解析过程被用来决定哪个函数与函数调用中的实参最匹配。例如，考虑下列代码：

```
void calc( int ii, double dd ) {
     // 调用模板实例还是普通函数?
     sum( dd, ii );
}
```

sum(dd,ii)调用由模板实例化的函数。还是调用普通非模板函数？为了回答这个问题，让我们逐步分析函数重载解析过程。函数重载解析的第一步是构造可以被调用的候选函数集。该集合由与被调用函数同名的函数构成，在调用点上，这些函数都有一个声明是可见的。

当存在一个函数模板时，如果用该函数调用的实参可以实例化一个函数，则从该模板实例化的实例也是一个候选函数。一个函数是否能被实例化，取决于模板实参推演过程是否能进行。（模板实参推演的过程在 10.3 节中说明。）在前面的例子中，函数实参 dd 被用来推演 Type 的模板实参。被推演出来的模板实参是 double，而模板实例 sum(double,int)被加入到候选函数集中。因此，该调用有两个候选函数。模板实例 sum(double,int)和普通函数 sum(double,double)。

一旦模板实例被加入到候选函数集中，函数重载解析过程就会像以前一样进行。

函数重载解析的第二步是从候选函数集中选择可行函数集。可行函数是这样的候选函数：对它而言，存在着能够把每个函数实参转换成相应的函数参数类型的类型转换。（9.3 节给出了可以被应用在函数实参上的类型转换。）对实例 sum(double,int)和非模板函数 sum(double,double)都存在类型转换，所以，这两个函数都是可行函数。

函数重载解析的第三步是为应用在实参上的类型转换划分等级，以便选择最佳可行函数。针对我们的例子，等级划分如下：

- 对于函数模板实例 sum(double,int)：

1．因为第一个实参的实参和参数类型都是 double，该转换是精确匹配。

2．因为第二个实参的实参和参数的类型都是 int，该转换也是精确匹配。

- 对非模板函数 sum(double,double)：

1．因为第一个实参的实参和参数类型都是 double，该转换也是精确匹配。

2．因为第二个实参的实参类型是 int，参数类型是 double，应用的转换是浮点——有序标准转换。

当只考虑第一个实参时。两个函数一样好。但是，函数模板实例对于第二个参数更适合

一些。所以，对该调用选择得到的最佳可行函数是实例 sum(double,int)。

只有当模板实参推演成功时，函数模板实例才能进入候选函数集。所以即使模板实参推演失败，它也不是一个错误。在这种情况下，没有函数实例被加入到候选函数集中。例如，假设函数模板 sum()被声明如下：

```
// 函数模板
template <class T>
     int sum( T*, int ) { .... }
```

如果与前面相同的函数调用，模板实参推演过程将会失败，因为对于一个 double 型的函数实参来说不可能有 T*型的相应参数。因为对该调用来说并没有实例可以从该函数模板产生，所以也就不会有实例被加入到候选函数集中。那么在候选函数集中惟一的函数就是非模板函数 sum(double,double)，则它就是此调用选择出来的函数，第二个实参将转换成 double 型。

如果模板实参推演成功，但是被推演的模板实参被显式特化，又会怎么样呢？进入候选函数集合的是显式特化，它将代替从通用模板定义实例化的函数。例如：

```
// 函数模板定义
template <class Type> Type sum( Type, int ) { /* ... */ }

// Type == double 的显式特化
template<> double sum<double>( double,int );

// 普通（非模板）函数
double sum( double, double );
void manip( int ii, double dd ) {
     // 调用模板显式特化 sum<double> ()
     sum( dd, ii );
}
```

对 manip()中的 sum()的调用，模板实参推演过程发现从通用模板定义生成的实例 sum(double,int)应该进入候选函数集合。但是，sum(double,int)有一个显式特化，所以进入候选函数集的是显式特化。实际上，同为后来发现该特化是该调用的最好匹配，所以它是被函数重载解析过程选中的函数。

模板显式特化不会自动进入候选函数集，只有模板实参推演成功的模板特化才被考虑。例如：

```
// 函数模板定义
template <class Type>
     Type min( Type, Type ) { /* ... */ }

// Type == double 的显式特化
template<> double min<double>( double,double );

void manip( int ii, double dd ) {
     // 错误：模板实参推演失败
     // 该调用没有候选函数
     min( dd, ii );
}
```

这里函数模板 min()将针对实参 double 进行特化。但是，这个特化并没有进入候选函数

集、manip()中对 min()调用时，模板实参推演会失败，因为从每个函数实参为 Type 推演出来的模板实参是不同的。对于第一个实参，为 Type 推演出来的类型是 double。而对于第二个实参，为 Type 推演出来的类型是 int。因为模板实参推演失败，所以没有实例进入候选函数集，并且特化 min(double,double)也被忽略。又因为该调用没有其他的候选函数，所以调用是错误的。

正如 10.6 节所提到的，一个普通函数也可以具有与“一个模板的实例化函数”完全匹配的返回类型和参数表。在下列例子中，函数 min(int,int)只是一个普通函数，而不是函数模板 min()的特化。你可能还记得，特化声明必须以符号 template<>开始：

```
// 函数模板声明
template <class T>
     T min( T, T );

// 普通函数声明
int min( int, int ) { }
```

一个函数调用可以与普通函数以及函数模板的实例化函数都匹配。在下列例子中，调用 min(ai[0],99)的两个实参类型都是 int。该调用有两个可行函数：普通函数 min(int,int)和从函数模板实例化的、带有相同返回类型和参数的函数。

```
int ai[4] = { 22, 33, 44, 55 };
int main() {
     // ／调用普通函数 min( int, int )
     min( ai[0], 99 );
}
```

但是，这样的调用不是二义的。当非模板函数存在时，因为该函数被显式实现，所以它将被给予更高的优先级。函数重载解析过程为该调用选择普通函数 min(int,int)。

一旦某个调用被函数重载解析过程解析为一个普通函数，如果该程序不包含该函数的定义，也不能回头了。如果不存在函数体，编译器也不能将函数模板实例化，为此函数生成函数体。取而代之的是产生一个链接时刻错误。在下面的例子中，程序调用了一个普通函数 min(int,int)，但是没有定义它。该程序产生了一个链接错误：

```
// 函数模板
template <class T>
     T min( T, T ) { .... }

// 这个普通函数在该程序中没有被定义
int min( int ,int );
int ai[4] = { 22, 33, 44, 55 };
int main() {
     // 链接错误: min(int, int) 被调用，但是没有被定义
     min( ai[0], 99 );
}
```

定义一个普通函数，让它的返回类型和参数表与另一个从模板实例化的函数的相同，又有什么用处呢？记住，当调用从模板实例化的函数时，只有有限的类型转换可以被应用在模板实参推演过程使用的函数实参上、如果声明一个普通函数，则可以考虑用所有的类型转换来转换实参，这是因为普通函数参数的类型是固定的。让我们看一个例子，它给出了声明一

个普通函数的原因。

假设我们想定义一个函数模板特化 min<int>(int，int)并希望：当用任何整型类型的实参调用 min()时，无论实参类型是否相同都调用这个函数。因为类型转换上的限制，所以用不同类型的整型实参的调用不会直接调用函数模板实例 min<int>(int,int)。我们可以通过指定显式模板实参直接调用该实例。但是，我们更喜欢一个不要求修改每个调用点的方案。通过定义一个普通函数，无论何时使用整型实参，我们的程序都会调用 min(int,int)的特化版本，而无需我们为每个调用都使用显式模板实参。例如：

```
// 函数模板定义
template <class Type>
     Type min( Type t1, Type t2 ) { ... }

int ai[4] = { 22, 33, 44, 55 };
short ss = 88;
void call_instantiation() {
     // 错误: 这个调用没有候选函数
     min( ai[0], ss );
}
// 普通函数
int min( int a1, int a2 ) {
     min<int>( a1, a2 );
}
int main() {
     call_instantiation();
     // 调用普通函数
     min( ai[0], ss );
}
```

在 call_instantiation()中的调用 min(ai[0],ss)没有候选函数。因为，想要从函数模板 min()生成一个候选函数肯定要失败从函数实参将为 Type 推演出不同的模板实参，所以该调用是错误的。但是对于 main()中的调用 min(ai[0],ss)来说，普通函数 min(int,int)的声明是可见的。这个普通函数是该调用的可行函数：第一个实参的类型与相应参数的类型精确匹配第二个实参可以用提升转换为相应参数的类型。由于该普通函数是第二个调用的可行函数集中的惟一函数，所以它将被选中。

我们已经了解当涉及到同名的函数模板实例、函数模板特化以及普通函数时，函数重载解析是怎样进行的，现在让我们来总结一下，对于一个调用，考虑普通函数和函数模板的函数重载解析步骤：

1．生成候选函数集。

考虑与函数调用同名的函数模板。如果对于该函数调用的实参，模板实参推演能够成功则实例化一个函数模板，或者对于推演出来的模板实参存在一个模板特化，则该模板特化就是一个候选函数。

2．生成可行函数集，如 9.3 节所描述。

只保留候选函数集中可以用函数调用实参调用的函数。

3．对类型转换划分等级，如 9.3 节中描述。

a．如果只选择了一个函数，则调用该函数。

b．如果该调用是二义的，则从可行函数集中去掉函数模板实例。

4．只考虑可行函数集中的普通函数，完成重载解析过程，如 9.3 节中描述。

a．如果只选择了一个函数，则调用该函数。

b．否则，该调用是二义的。

让我们一步步地来看一个例子。下面是两个声明：一个函数模板声明，一个带有两个 double 型实参的普通函数。

```
template <class Type>
    Type max( Type, Type ) { .... }

// 普通函数
double max( double, double );
```

下面是三个 max()调用。你能分辨出每个调用分别会调用哪个实例吗？

```
int main() {
    int ival;
    double dval;
    float fd;

    // 向 ival, dval, 和 fd 赋某些值
    max( 0, ival );
    max( 0.25, dval );
    max( 0, fd );
}
```

让我们按顺序查看每个调用。

- max(o,ival)：两个实参的类型都是 int。对该调用存在两个候选函数：函数模板实例 max(int,int)以及普通函数 max(double,double)。用于函数模板实例对函数实参来说是精确匹配的，所以它将是被调用的函数。
- max(0.25,dval)：这两个实参都是 double 型。对该调用存在两个候选函数：函数模板实例 max(double,double)以及普通函数 max(double,double)。因为调用与两个函数都完全匹配，所以该调用存在二义。根据规则 3b 可知在这种情况下应选择普通函数。
- max(0,fd)：实参的类型分别是 int 和 float。对该调用只存在一个候选函数：普通函数 max(double,double)。因为从两个函数实参为 Type 推演出的模板实参不是一种类型，所以模板实参推演会失败，故也没有模板实例进入候选函数集。因为存在类型转换能把函数实参转换成相应函数参数的类型，该普通函数是个可行函数。因此，选择普通函数。如果该普通函数没有被定义，则该调用将是个错误。

如果我们已经为 max()定义了第二个普通函数，又会怎么样呢？例如：

```
template <class T> T max( T, T ) { .... }

// 两个普通函数
char max( char, char );
double max( double, double );
```

第三个调用的解析会不同吗？是的。

```
int main() {
     float fd;
     // 解析为哪个函数?
     max( 0, fd );
}
```

规则 3b 说明，因为该调用是二义的，所以只考虑普通函数。这些函数都没有被选为最佳可行函数，因为在实参上的类型转换对两个函数都是一样的不好：两个实参都要求标准转换以便匹配可行函数中的两个相应的参数。所以该调用是二义的，将被编译器标记为错误。

练习 10.11

让我们回到前面给出的例子

```
template <class Type>
     Type max( Type, Type ) { .... }

double max( double, double );

int main() {
     int ival;
     double dval;
     float fd;
     max( 0, ival );
     max( 0.25, dval );
     max( 0, fd );
}
```

下面的函数模板特化被加入到全局域的声明集中：

```
template <> char max<char>( char, char ) { .... }
```

重新考虑 main()中的函数调用，并为每个调用列出候选函数和可行函数。假定下面的函数调用被加到 main()中，则该调用将被解析为哪个函数？为什么？

```
int main() {
     // ...
     max (0, 'J' );
}
```

练习 10.12

假设有下列模板定义和特化，以及变量、函数声明的集合：

```
int i; unsigned int ui;
char str[24]; int ia[24];
template <class T> T calc( T*, int );
template <class T> T calc( T, T );
template<> char calc( char*, int );
double calc( double, double );
```

指出下列每个调用将会调用哪个模板实例或函数。为每个调用，列出候选函数、可行函数，并解释选择最佳可行函数的原因。

```
(a) calc( str, 24 );   (d) calc( i, ui );
(b) calc( ia, 24 );    (e) calc( ia, ui );
(c) calc( ia[0], i );  (f) calc( &i, i );
```

10.9 模板定义中的名字解析 ※

在模板定义中有些结构在两个模板实例之间有不同的意义，而另外一些结构在模板的所有实例之间意义相同。这取决于该结构是会涉及模板参数。例如：

```
template<typename Type>
Type min( Type* array, int size )
{
    Type min_val = array[0];

    for ( int i = 1; i < size; ++i )
        if ( array[i] < min_val )
            min_val = array[i];
    print( "Minimum value found: " );
    print( min_val );
    return min_val;
}
```

在 min()中，array 和 min_val 的类型取决于模板被实例化时取代 Type 的实际类型，而 size 的类型总是 int，与模板参数的类型无关。array 和 min_val 的类型随不同的模板实例而不同。因此，我们说这些变量的类型依赖于模板参数（depend on a template parameter），而 size 的类型则不依赖于模板参数。

因为不知道 min_val 的类型，所以当 min_val 出现在表达式中时，究竟哪个操作将会被用到也是未知的。例如，函数调用 print(min_val)将会调用哪个 print()函数？应该是 int 类型的 print()函数或是 float 类型的 print()函数？会不会因为没有 print()函数可以用 min_val 的类型的实参来调用而使得这个调用是错误的？只有当实例化该模板、知道了 min_val 的实际类型时，我们才能回答这些问题。因此我们也称调用 print(min_val)依赖于模板参数。

在 min()中，不依赖于模板参数的结构就没有这样的问题。例如，调用 print("Minimum value found")总知道应该调用哪个函数。这个函数被用来输出字符串，被调用的 print()函数不会随着模板实例的不同而不同。因此，我们称该调用不依赖于模板参数。

正如我们在第 7 章所了解的，在 C++中，函数必须在调用前被声明。在模板定义中，被调用的函数必须在模板定义出现之前被声明吗？在前面的例子中，在 min()的模板定义中，函数 print()必须在调用之前先声明吗？答案取决于我们引用了哪种名字。不依赖于模板参数的结构必须在使用之前先声明。因此前面给出的函数模板 min()的定义是不正确的。因为调用

```
print( "Minimum value found: " );
```

不依赖于模板参数，所以针对字符串的函数 print()必须在被模板定义使用它之前先被声明。为了解决这个问题，print()函数的声明必须在 min()的定义之前给出，如下所示：

```
// ---- primer.h ----
// 这个声明是必需的
// print( const char * ) 在 min() 中被调用
void print( const char* );

template<typename Type>
Type min( Type* array, int size ) {
     // ....
     print( "Minimum value found: " );
     print( min_val );
     return min_val;
}
```

另一方面，调用 print(min_val)使用的 print()函数的声明则是不需要的，因为我们还不知道要寻找的是哪个 print()。只有当 min_val 的类型是已知的时候，才可能知道应该调用哪个 print()函数。

那么应该在什么时候声明 print(min_val)调用的 print()函数呢？必须在实例化模板之前声明。例如：

```
#include <primer.h>

void print( int );
int ai[4] = { 12, 8, 73, 45 };

int main() {
     int size = sizeof(ai) / sizeof(int);

     // min( int*, int ) 的实例
     min( &ai[0], size );
}
```

函数 main()调用函数模板实例 min(int*,int)。在 min()的这个实例中，Type 被 int 取代，而变量 min_val 的类型是 int。所以 print(min_val)调用一个可以用 int 类型的实参来调用的函数。在实例化 min(int*,int)的时候，我们知道 print()函数的第二个调用有一个 int 型的实参。也正是在这个时候。要求可以用 int 型实参调用的 print()函数必须可见。在我们的例子中，被选择的函数是 print(int)。如果函数 print(int)在 min(int*,int)被实例化之前没有被声明，则该实例化会导致编译时刻错误。

所以，模板定义中的名字解析分两个步骤进行。首先：不依赖于模板参数的名字在模板定义时被解析；其次，依赖于模板参数的名字在模板被实例化时被解析。你可能会问，为什么要分两步呢？为什么不是所有的名字都在模板被实例化时被解析呢？

如果我们是函数模板的设计者，我们可以控制模板定义中的名字解析方式。假设函数模板 min()位于一个函数库中，该库还定义了其他一些模板和函数。我们希望 min()的实例在所有可能的时候都使用库中的其他组件。在前面的例子中，库的接口定义位于头文件<primer.h>中。函数 print(const char*)的声明和函数模板 min()的定义都是库的接口的一部分。我们希望 min()的实例调用该库提供的 print()函数。名字解析的第一步保证了这件事情。当用在模板定义中的名字不依赖于模板参数时，此名字肯定会指向库中定义的另一个组件——即，它会

指向和函数模板定义在一起、被打包在头文件<primer.h>中的声明。

事实上，函数模板的设计者必须确保为模板定义中用到的、所有不依赖于模板参数的名字提供声明。如果用在模板定义中的名字不依赖于模板参数，并且在定义该模板时没有找到该名字的声明，则模板定义是错误的。当模板被实例化时，编译器就不再考虑这样的错误。例如：

```
// ---- primer.h ----
template<typename Type>
Type min( Type* array, int size )
{
     Type min_val = array[0];
     // ...
     // 错误: 没有找到 print ( const char* )
     print( "Minimum value found: " );
     // ok: 依赖于模板参数
     print( min_val );
     // ...
}

// ---- user.C ----
#include <primer.h>

// print( const char* ) 的这个声明被忽略
void print( const char* );

void print( int );
int ai[4] = { 12, 8, 73, 45 };
int main() {
     int size = sizeof(ai) / sizeof(int);
     // min( int*, int ) 的实例
     min( &ai[0], size );
}
```

在 user.C 中的 print(const char*)的声明，在模板定义出现的地方是不可见的。但是，这样的声明在模板 min(int*,int)被实例化时是可见的，但该声明并不被调用 print("minimum value found:")所考虑，因为该调用不依赖于模板参数。除非模板定义中的结构依赖于模板参数，否则一个名字在模板定义的上下文中被解析，此解析过程不会在该模板实例化的上下文中被重新考虑。因此，确保被用在模板定义中的名字的声明和模板定义被正确地包含为库接口的一部分，这是模板设计者的责任。

让我们换个思路，假设该库是由其他人编写的，我们是头文件<primer.h>中的定义的用户。有这样一种情况，我们希望当程序在实例化库中的模板时考虑程序中定义的对象和函数。例如，假设我们的程序定义了一个称为 SmallInt 的类。我们希望实例化库<primer.h>中的函数 min()，以获得 SmallInt 型的对象的数组中的最小值。

当函数模板 min()被 SmallInt 型对象的数组实例化时，Type 的模板实参是 class 类型的 SmallInt。这意味着，在 min()的实例中，min_val 的类型是 SmallInt。在 min()的实例中，调用 print(min_val)应该被解析为哪个函数？

```
// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
SmallInt asi[4];
int main() {
     // 设置 asi 的元素
     // min( SmallInt*, int ) 的实例
     int size = sizeof(asi) / sizeof(SmallInt);
     min( &asi[0], size );
}
```

对，我们希望考虑我们的函数 print(const SmallInt&)。只考虑在库<primer.h>中定义的函数是远远不够的。名字解析的第二步保证这样的事情可以发生。当用在模板定义中的名字依赖于模板参数时，在实例上下文中声明的名字被考虑。所以，对于函数模板中的该操作，如果模板实参是 SmallInt 型，那么可以处理 SmallInt 型的对象的函数肯定会被考虑。

在源代码中模板被实例化的位置被称为模板的**实例化点**（point of instantiation）。知道模板的实例化点是很重要的，因为它决定了对依赖于模板参数的名字所考虑的声明。函数模板的实例化点总是在名字空间域中，并且跟在引用该实例的函数后。例如，min(SmallInt*,int)的实例化点就紧跟在名字空间域中的函数 main()后：

```
// ...
int main() {
     // ...
     // 使用 min( SmallInt*, int )
     min( &asi[0], size );
}
// min( SmallInt*, int ) 的实例化点
// 好像实例定义如下出现
SmallInt min( SmallInt* array, int size )
     { /* ... */ }
```

但是，如果在一个源文件中，一个模板实例不只被使用一次，又该怎么办呢？实例化点在哪儿呢？可能你会问：为什么这很重要？在我们的例子中，它的重要性是因为函数 print(const SmallInt&)的声明必须出现在 min(SmallInt*,int)的实例化点之前。例如：

```
#include <primer.h>

void another();
SmallInt asi[4];
int main() {
     // 设置 asi 的元素
     int size = sizeof(asi) / sizeof(SmallInt);
     min( &asi[0], size );
     another();

     // ...
```

```
}
// 实例化点在这儿?

void another() {
     int size = sizeof(asi) / sizeof(SmallInt);
     min( &asi[0], size );
}
// 还是这儿?
```

当模板实例要多次使用时，在每个使用该实例的函数定义之后都有一个实例化点。编译器自由选择这些实例化点之一来真正实例化该函数模板。这意味着在组织代码时，我们必须小心地把解析依赖于模板参数的名字所需要的声明放置在模板的第一个实例化点之前。因此，在模板的任何一个实例被使用之前，应该头文件中给出所有必需的声明，并包含这个头文件。例如：

```
#include <primer.h>

// user.h 含有实例所需的声明
#include "user.h"

void another();
SmallInt asi[4];
int main() {
     // ...
}
// min( SmallInt*, int ) 的第一个实例化点
void another() {
     // ...
}
// min( SmallInt*, int ) 的第二个实例化点
```

如果不只在一个文件中使用模板实例，该怎样办呢？例如，如果函数 another()函数 main()在不同的文件中，该怎样办呢？那么在每个使用该模板的实例的文件中都存在一个实例化点。编译器自由选择这些文件中的任一个实例化点来实例化该函数模板。所以在组织代码时，我们必须也要小心地把头文件“user.h”包含在每个使用该函数模板实例的文件中。这保证 min(SmallInt*,int)的实例指向我们的函数 print(const SmallInt&)，这是我们所期望的，与编译器选择哪个实例化点无关。

练习 10.13

列出模板定义中名字解析的两个步骤。解释第一步如何解决库设计者关心的事情，以及第二步怎样提供模板用户所需要的灵活性。

练习 10.14

在 max(LongDouble*,SIZE)实例中的名字 display 和 SIZE 引用哪个声明？

```
// ---- exercise.h ----
void display( const void* );
typedef unsigned int SIZE;
template<typename Type>
```

```
        Type max( Type* array, SIZE size )
{
        Type max_val = array[0];
        for ( SIZE i = 1; i < size; ++i )
        if ( array[i] > max_val )
        max_val = array[i];
        display( "Maximum value found: " );
        display( max_val );
        return max_val;
}

// ---- user.h ----
class LongDouble { /* ... */ };
void display( const LongDouble & );
void display( const char * );
typedef int SIZE;

// ---- user.C ----
#include <exercise.h>
#include "user.h"
LongDouble ad[7];
int main() {
        // 设置 ad 的元素
        // max( LongDouble*, SIZE ) 的实例化
        SIZE size = sizeof(ad) / sizeof(LongDouble);
        max( &ad[0], size );
}
```

10.10 名字空间和函数模板 ※

与其他全局域定义一样，函数模板定义也可以被放在名字空间中。（关于名字空间的讨论见 8.5 节和 8.6 节。）这种模板定义的意义与在全局域中定义的一样，除了该模板的名字被隐藏在名字空间中。当在名字空间之外使用该模板时，该模板名必须被限定，或提供一个 usign 声明。

```
// ---- primer.h ----
namespace cplusplus_primer {
        // 模板定义被隐藏在名字空间中
        template<class Type>
            Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.C ----
#include <primer.h>

int ai[4] = { 12, 8, 73, 45 };
int main() {
        int size = sizeof(ai) / sizeof(ai[0]);
```

```
        // 错误: 没有找到函数 min()
        min( &ai[0], size );
        using cplusplus_primer::min;  // using 声明

        // ok: 指向名字空间 cplusplus_primer 中的 min()
        min( &ai[0], size );
    }
```

如果我们的程序使用了一个在名字空间中定义的模板，而且我们希望为其提供一个特化，又会怎么样？（关于模板显式特化在 10.6 节介绍）。例如，我们想用名字空间 cplusplus_primer 中定义的模板 min()来找到 SmallInt 型的对象数组中的最小值。但是，我们意识到，在名字空间 cplusplus_primer 中提供的模板定义不能完全奏效。函数模板中的比较操作如下：

```
    if ( array[i] < min_val )
```

该语句用小于（<）操作符比较两个 SmallInt 型的类对象。除非为类 SmallInt 定义了一个重载的 operator<()，否则该操作符不能被应用在两个类对象上。（我们将在 15 章看到怎样定义重载操作符。）假设我们为 min()函数模板定义一个特化，以便用一个名为 compareLess()的函数找到 SmallInt 类对象的数组中的最小值。下面是 compareLess()函数的声明：

```
    // SmallInt 对象的比较函数
    // 如果 param1 小于 parm2, 返回 true
    bool compareLess( const SmallInt &parm1, const SmallInt &parm2 );
```

这个函数的定义看起来会是什么样的呢？为了回答这个问题，我们需要更详细地看看我们的 SmallInt 的定义。这个 SmallInt 可以用来一些对象，能够存放定义与 8 位 unsigned char 范围相同的值，即 0－255。它的附加功能是能够捕捉到上溢和下溢错误。除此之外，我们还希望它的工作方式与 unsigned char 相同。类 SmallInt 的定义看起来如下：

```
    class SmallInt {
    public:
        SmallInt( int ival ) : value( ival ) { }
        friend bool compareLess( const SmallInt &, const SmallInt & );

    private:
        int value;    // 数据成员
    };
```

在这个类定义中，有些事情我们应该讨论一下、首先，该类有一个私有数据成员：value。它是存储 SmallInt 型对象值的数据成员。该类还包含一个构造函数：

```
    // 类 SmallInt 的构造函数
    SmallInt( int ival ) : value( ival ) { }
```

该构造函数有一个参数：ival。它执行的惟一动作是用参数 ival 的值初始化类数据成员 value。

现在，我们可以回答前面的问题了。函数 compareLess()怎样定义？该函数将比较它的两个 SmallInt 型参数的 value 数据成员，如下：

```
    // 如果 parm1 小于 parm2, 则返回 true
    bool compareLess( const SmallInt &parm1, const SmallInt &parm2 ) {
        return parm1.value < parm2.value;
    }
```

但是要注意，数据成员 value 是 SmallInt 类的私有数据成员。这个全局函数怎样引用该私有成员而不破坏类 SmallInt 的封装化且不会引起编译错误呢？如果看看 SmallInt 的定义，你会注意到这个类定义把全局函数 compareLess()声明为 friend。当一个函数是一个类的 friend 时，它就可以引用该类的私有成员，譬如我们的函数 compareLess()在 15.2 节中对类的 fiend 有进一步介绍）。

现在我们已经准备好为 min()定义我们的模板特化了。它如下使用 compareLess()函数：

```
// 针对 SmallInt 对象数组的 min() 特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
{
    SmallInt min_val = array[0];
    for ( int i = 1; i < size; ++i )
    // 使用函数 compareLess() 比较
    if ( compareLess( array[i], min_val ) )
        min_val = array[i];
    print( "Minimum value found: " );
    print( min_val );
    return min_val;
}
```

我们应该在哪里声明这个特化？看看下面的做法怎么样：

```
// ---- primer.h ----
namespace cplusplus_primer {
    // 模板定义被隐藏在名字空间中
    template<class Type>
        Type min( Type* array, int size ) { /* ... */ }
}

// ---- user.h ----
class SmallInt { /* ... */ };
void print( const SmallInt & );
bool compareLess( const SmallInt &, const SmallInt & );

// ---- user.C ----
#include <primer.h>
#include "user.h"
// 错误: 不是 cplusplus_primer::min() 的特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
    { /* ... */ }
// ...
```

不幸的是，该代码不能完成任务。函数模板的显式特化声明必须被声明在该通用模板被定义的名字空间内。因此，我们必须在名字空间 cplusplus_primer 中定义 min()的特化。在我们的程序中，有两种实现它的方式。

请回忆一下，由于名字空间的定义可以是非连续的，所以我们可以重新打开名字空间 cplusplus_primer 的定义并加上该特化的定义，如下：

```
// ---- user.C ----
#include <primer.h>
#include "user.h"
```

```
namespace cplusplus_primer {
// cplusplus_primer::min() 的特化
template<> SmallInt min<SmallInt>( SmallInt* array, int size )
     { /* ... */ }
}
SmallInt asi[4];

int main() {
     // 用 set() 成员函数设置 asi 的元素
     using cplusplus_primer::min; // using declaration
     int size = sizeof(asi) / sizeof(SmallInt);
     // min(SmallInt*,int) 的实例化
     min( &asi[0], size );
}
```

或者我们可以用“在名字空间定义之外定义任何名字空间成员”的方式定义该特化：通过用外围名字空间名来限定修饰名字空间成员名。

```
// ---- user.C ----
#include <primer.h>
#include "user.h"

// cplusplus_primer::min() 的特化
// 此特化的名字被限定修饰
template<> SmallInt cplusplus_primer::
     min<SmallInt>( SmallInt* array, int size )
          { /* ... */ }

// ...
```

所以，作为包含模板定义的库的用户，如果我们想为库中的模板提供特化，那么我们就必须确保它们的定义被合适地放置在含有原始模板定义的名字空间内。

练习 10.15

现在我们把练习 10.14 中给出的头文件<exercise.h>的内容放到名字空间 cplusplus_primer 中。怎样改变函数 main()使其能够实例化位于名字空间 cplusplus_primer 中的函数模板 max()?

练习 10.16

再次参考练习 10.14，已知头文件<exercise.h>的内容被放在名字空间 cplusplus_primer 中，我们想为类 LongDouble 对象的数组定义函数模板 max()的特化。并且让该模板特化使用如下定义的函数 compareGreater()，来比较两个 LongDouble 型的对象：

```
// 比较两个 LongDouble 对象的函数
// 如果 parm1 大于 parm2, 则返回 true
bool compareGreater( const LongDouble &parm1,
          const LongDouble &parm2 );
```

类 LongDouble 的定义如下：

```
class LongDouble {
```

```
public:
    LongDouble(double dval) : value(dval) { }
    void set(double dval) { value = dval; }

    friend bool compareGreater( const LongDouble &,
        const LongDouble & );
private:
    double value;
};
```

给出函数 compareGreater()以及使用该函数的 max()特化的定义。写一个用于设置数组 ad 元素的 main()函数，然后调用 max()的特化取得 ad 中的最大值：初始化数组 ad 的元素的值应该通过读取标准输入 cin 获得。

10.11 函数模板示例

本节将给出一个程序示例，用来说明怎样定义和使用函数模板。这个示例定义了一个函数模板 sort()，它对数组的元素进行排序。数组本身由在 2.5 节介绍的 Array 类模板表示。因此，函数模板 sort()可以被用来排序任意类型元素的数组。

我们在第 6 章中看到，C++标准库定义了一个容器类型被称为 Vector，它的行为和 2.5 节定义的 Array 非常相像。第 12 章将介绍泛型算法，它可以处理第 6 章描述的容器类型。其中一个算法被称为 sort()，它可以被用来排序 vector 的内容。在本节中，我们将定义自己的“泛型 sort()算法”，以处理我们的 Array 类。你在本节中所看到的是 C++标准库中的算法的简化版本。

我们针对 Array 类模板的 sort()函数模板定义如下：

```
#include "Array.h"

template <class elemType>
void sort( Array<elemType> &array, int low, int high ) {
    if ( low < high ) {
        int lo = low;
        int hi = high + 1;
        elemType elem = array[lo];

        for (;;) {
            while ( min( array[++lo], elem ) != elem && lo < high ) ;
            while ( min( array[--hi], elem ) == elem && hi > low ) ;
            if (lo < hi)
                swap( array, lo, hi );
            else break;
        }
        swap( array, low, hi );
        sort( array, low, hi-1 );
        sort( array, hi+1, high );
    }
}
```

函数 sort()使用了两个辅助函数：min()和 swap()。这两个函数都被定义为函数模板，而且能够处理可用来实例化 sort()的全部实参类型。min()被定义为函数模板，以便我们可以找到任意类型的两个数组元素的最小值：

```
template <class Type>
    Type min( Type a, Type b ) {
        return a < b ? a : b;
}
```

swap()被定义为函数模板，以便我们可以交换任意类型的两个数组元素：

```
#include "Array.h"
template <class elemType>
    void swap( Array<elemType> &array, int i, int j )
{
    elemType tmp = array[ i ];
    array[ i ] = array[ j ];
    array[ j ] = tmp;
}
```

为了确保我们的 sort()函数模板能够真正地工作，我们需要在数组被排序后显示数组的内容。因为 display()函数必须能够处理 Array 类模板被实例化后的任何数组，所以我们也必须把 display()定义为函数模板：

```
#include <iostream>

template <class elemType>
    void display( Array<elemType> &array )
{ // display format: < 0 1 2 3 4 5 >
    cout << "< ";

    for ( int ix = 0; ix < array.size(); ++ix )
        cout << array[ix] << " ";

    cout << ">\n";
}
```

在这个例子中，我们使用包含编译模式，并把函数模板放在头文件 Array.h 中，跟在 Array 类模板后面。

下一步是写一个函数练习这些函数模板。依次被传递给函数 sort()的数组为 double 型的数组、int 型的数组和 string 型的数组。下面是程序：

```
#include <iostream>
#include <string>

#include "Array.h"

double da[10] = {
    26.7, 5.7, 37.7, 1.7, 61.7, 11.7, 59.7,
    15.7, 48.7, 19.7 };

int ia[16] = {
    503, 87, 512, 61, 908, 170, 897, 275, 653,
```

```
        426, 154, 509, 612, 677, 765, 703 };

    string sa[11] = {
        "a", "heavy", "snow", "was", "falling", "when",
        "they", "left", "the", "police", "station" };

    int main() {
        // 调用构造函数初始化 arrd
        Array<double> arrd( da, sizeof(da)/sizeof(da[0]) );

        // 调用构造函数初始化 arri
        Array<int> arri( ia, sizeof(ia)/sizeof(ia[0]) );

        // 调用构造函数初始化 arrs
        Array<string> arrs( sa, sizeof(sa)/sizeof(sa[0]) );

        cout << "sort array of doubles (size == "
             << arrd.size() << ")" << endl;
        sort( arrd, 0, arrd.size()-1 );
        display(arrd);

        cout << "\sort array of ints (size == "
             << arri.size() << ")" << endl;
        sort( arri, 0, arri.size()-1 );
        display(arri);

        cout << "\sort array of strings (size == "
             << arrs.size() << ")" << endl;
        sort( arrs, 0, arrs.size()-1 );
        display(arrs);

        return 0;
    }
```

编译并运行该程序，产生下列输出（数组的输出已经被手工调整以适应篇幅）：

```
    sort array of doubles (size == 10)
    < 1.7 5.7 11.7 14.9 15.7 19.7 26.7
    37.7 48.7 59.7 61.7 >

    sort array of ints (size == 16)
    < 61 87 154 170 275 426 503 509 512
    612 653 677 703 765 897 908 >

    sort array of strings (size == 11)
    < "a" "falling" "heavy" "left" "police" "snow"
    "station" "the" "they" "was" "when" >
```

在 C++标准库定义的泛型算法中（在第 12 章中），你还会找到一个 min()函数和一个 swap()函数。在第 12 章中我们将了解怎样在程序中使用它们。