9

重 载 函 数

我们已经知道怎样声明和定义函数，以及怎样在程序中使用函数，在本章中我们将了解 C++支持的一种特殊函数：重载函数。如果两个函数名字相同，并且在相同的域中被声明，但是参数表不同，则它们就是重载函数（overloaded function）。在本章中，我们将首先了解怎样声明一组重载函数，以及这样做的好处。然后，再看看函数重载解析过程是怎样进行的——即，一个函数调用怎样被解析为一组重载函数中的某一个函数。函数重载解析过程是 C++中最复杂的内容之一。本章的结尾将为那些希望进一步详细了解重载函数的人提供了两个小节的高级主题。它们将更完整地描述参数类型转换和函数重载的解析。

本文中的“域”指名字的作用域

9.1 重载函数声明

我们已经知道怎样声明和定义函数，以及怎样在程序中使用函数，现在我们将了解 C++支持的另一种函数新特性：重载函数。**函数重载**（function overloading）允许多个函数共享同一个函数名，但是针对不同参数类型提供共同的操作。

如果你曾经用一种程序设计语言写过算术表达式，那么你就已经使用过预定义的重载函数。例如，如下表达式：

```
1 + 3
```

调用了针对整数操作数的加法操作，而表达式：

```
1.0 + 3.0
```

调用了另外一个专门处理浮点操作数的不同的加法操作。实际被使用的操作对用户而言是透明的。加法操作被重载，以便处理不同的操作数类型。根据操作数的类型来区分不同的操作并应用适当的操作，是编译器的责任，而不是程序员的事情。

本章我们将了解怎样定义自己的重载函数。

9.1.1 为什么要重载一个函数名

正如内置加法操作的情形一样，我们可能希望定义一组函数，它们执行同样的一般性动

作，但是应用在不同的参数类型上。例如，假设我们希望定义一个函数，它返回参数中的最大值。

如果没有重载一个函数名的能力，那么我们就必须为每个函数给出一个惟一的名字。例如，我们可能如下定义一组 max()函数：

```
int i_max( int, int );
int vi_max( const vector<int> & );
int matrix_max( const matrix & );
```

但是，这些函数都执行了相同的一般性动作；都返回参数集合中的最大值。从用户的角度来看，只有一种操作，就是判断最大值。至于怎样完成其细节，函数的用户一点也不关心。

这种词汇上的复杂性不是“判断一组数中最大值”问题本身固有的，而是反映了程序设计环境的一种局限性：在同一个域中出现的名字必须指向一个唯实体（惟一的对象、函数、class 类型等等）。这种复杂性给程序员带来了一个实际问题，他们必须记住或查找每一个名字。函数重载把程序员从这种词汇复杂性中解放出来。

通过函数重载，程序员可以简单地这样写：

```
int ix = max( j, k );
vector<int> vec;
// ...
int iy = max( vec );
```

这项技术可以获得各种条件下的最大值。

9.1.2 怎样重载一个函数名

在 C++中，可以为两个或多个函数提供相同的名字，只要它们的每个参数表惟一就行：或者是参数的个数不同，或者是参数类型不同。下面是重载函数 max()的声明：

```
int max( int, int );
int max( const vector<int> & );
int max( const matrix & );
```

参数集惟一的每个重载声明都要求一个独立的 max()定义。

当一个函数名在一个特殊的域中被声明多次时，编译器按如下步骤解释第二个（以及后续的）的声明。

- 如果两个函数的参数表中参数的个数或类型不同，则认为这两个函数是重载的。例如：

```
// 重载函数
void print( const string & );
void print( vector<int> & );
```

- 如果两个函数的返回类型和参数表精确匹配，则第二个声明被视为第一个的重复声明。例如：

```
// 声明同一个函数
void print( const string &str );
void print( cons t string & );
```

参数表的比较过程与参数名无关。

- 如果两个函数的参数表相同，但是返回类型不同，则第一个声明被视为第一个的错误重复声明，会被标记为编译错误。例如：

```
unsigned int max( int i1, int i2 );
int max( int , int );  // 错误：只有返回类型不同
```

函数的返回类型不足以区分两个重载函数。

- 如果在两个函数的参数表中，只有缺省实参不同，则第二个声明被视为第一个的重复声明。例如：

```
// 声明同一函数
int max( int *ia, int sz );
int max( int *, int = 10 );
```

typedef 名为现有的数据类型提供了一个替换名：它并没有创建一个新类型。因此，如果两个函数参数表的区别只在于一个使用了 typedef，而另一个使用了与 typedef 相应的类型。则该参数表不被视为不同的。下列 calc()的两个函数声明被视为具有相同的参数表。第二个声明导致编译时刻错误，因为虽然它声明了相同的参数表，但是它声明了与第一个不同的返回类型。

```
// typedef 并不引入一个新类型
typedef double DOLLAR;

// 错误：相同参数表，不同返回类型
extern DOLLAR calc( DOLLAR );
extern int calc( double );
```

当一个参数类型是 const 或 volatile 时，在识别函数声明是否相同时，并不考虑 const 和 volatile 修饰符。例如，下列两个声明声明了同一个函数：

```
// 声明同一函数
void f( int );
void f( const int );
```

参数是 const，这只跟函数的定义有关系：它意味着，函数体内的表达式不能改变参数的值。但是，对于按值传递的参数，这对函数的用户是完全透明的：用户不会看到函数对按值传递的实参的改变。（按值传递的实参以及参数的其他传递方式在 7.3 节中讨论。），当实参被按值传递时，将参数声明为 const 不会改变可以被传递给该函数的实参种类。任何 int 型的实参都可以被用来调用函数 f(const int)。因为两个函数接受相同的实参集，所以刚才给出的两个声明并没有声明一个重载函数。函数 f()可以被定义为：

```
void f( int i ) { }
```

或：

```
void f( const int i ) { }
```

然而。在同一个程序中同时提供这两个定义将产生错误，因为这些定义把一个函数定义了两次。

但是，如果把 const 或 volatile 应用在指针或引用参数指向的类型上，则在判断函数声明

是否相同时，就要考虑 const 和 volatile 修饰符。

```
// 声明了不同的函数
void f( int* );
void f( const int* );

// 也声明了不同的函数
void f( int& );
void f( const int& );
```

9.1.3 何时不重载一个函数名

什么时候重载一个函数名没有好处？如果不同的函数名所提供的信息可使程序更易于理解的话，则再用重载函数就没有什么好处了。下面是一个例子，下列函数集合在一个公共数据抽象上进行操作，它们可能首先会被看作重载的对象：

```
void setDate( Date&, int, int, int );
Date &convertDate( const string & );
void printDate( const Date& );
```

这些函数在同一个数据类型（类 Date）上执行操作，但是并不共享同样的操作。在这种情况下，与函数名相关的词汇复杂性来自于程序员的习惯，他用这一组操作集和公共数据类型来命名函数。C++的类机制使得这种习惯变得不再必要。相反，这些函数应该成为类 Date 的成员，因为每个成员函数执行不同的操作，所以成员函数的名字应该表示它的操作。例如：

```
#include <string>
class Date {
public:
     set( int, int, int );
     Date &convert( const string & );
     void print();

     // ...
};
```

下面是另外一个例子，下列 Screen 类的五个成员函数在 Screen 的光标上执行各种移动操作，或许我们首先认为最好把这些函数以名字 move()重载：

```
Screen& moveHome();
Screen& moveAbs( int, int );
Screen& moveRel( int, int, char *direction );

Screen& moveX( int );
Screen& moveY( int );
```

最后两个实例并不能被重载，因为它们的参数表完全相同。为了提供一个惟一的标识，我们把两个函数如下压缩成一个：

```
// moveX() 和 moveY() 组合后的函数
Screen& move( int, char xy );
```

现在，每个函数都有了一个惟一的参数表，这样就能够用名字 move()重载该函数集合。

但是，根据我们的准则，重载函数是个坏主意：不同的函数名所提供的信息会被丢失，这使程序更难于理解。尽管光标移动是所有这些函数共享的通用操作，但是，这些函数之间移动的特性是惟一的。例如，moveHome()代表了光标移动的一个特殊实例。对程序的读者来说下面两个调用哪一个更易于理解？对 Screen 类的用户来说下面两个调用哪个更容易记忆？

```
// 哪一个更易于理解?
myScreen.home();  // 我们认为是这个
myScreen.move();
```

有时候，没有必要重载，可能也不需要不同的函数定义。在某些情况下，缺省实参可以把多个函数声明压缩为一个函数中。例如，两个光标函数：

```
moveAbs(int,int);
moveAbs(int,int,char*);
```

可以通过第三个 char*型参数的有无来区分。如果这两个函数的实现十分类似，并且在向函数传递参数时，如果能够找到一个 char*型缺省实参可以表示实参不存在时的意义，则这两个函数就可以被合并。现在，正好有个这样的缺省实参——值为 0 的指针：

```
move( int, int, char* = 0 );
```

程序员最好抱这样的观点：并不是每个语言特性都是你要攀登的下一座山峰。使用语言的特性应该遵从应用的逻辑，而不是简单地因为它的存在就必须要使用它。程序员不应该勉强使用重载函数。只有在必要的地方使用它们，才会让人感觉自然。

9.1.4 重载与域 ※

重载函数集合中的全部函数都应在同一个域中声明。例如，一个声明为局部的函数将隐藏而不是重载一个全局域中声明的函数。例如：

```
#include <string>
void print( const string & );
void print( double ); // overloads print()
void fooBar( int ival )
{
    // 独立的域: 隐藏 print()的两个实例
    extern void print( int );

    // 错误: print( const string & )在这个域中被隐藏
    print( "Value : " );
    print( ival ); // ok: print( int ) 可见
}
```

我们也可以在一个类中声明一组重载函数。因为每个类都维持着自己的一个域，所以两个不同类的成员函数不能相互重载。类成员函数将在第 13 章描述，而类成员函数的重载解析将在第 15 章描述。

我们也可以在一个名字空间内声明一组重载函数。每个名字空间也都维持着自己的一个域，作为不同名字空间成员的函数不能相互重载。例如：

```
#include <string>
namespace IBM {
```

```
        extern void print( const string < );
        extern void print( double );  // 重载 print()
    }

    namespace Disney {
        // 独立的域:
        // 没有重载 IBM 的 print()
        extern void print( int );
    }
```

using 声明和 using 指示符可以使一个名字空间的成员在另一个中可见，这些机制对于重载函数的声明有一些影响。关于 using 声明和 using 指示符在 8.6 节介绍。

using 声明怎样影响重载函数呢？using 声明为一个名字空间的成员在该声明出现的域中提供了一个别名。下面程序中的 using 声明会怎么样呢？

```
    namespace libs_R_us {
        int max( int, int );
        int max( double, double );
        extern void print( int );
        extern void print( double );
    }
    // using 声明
    using libs_R_us::max;
    using libs_R_us::print( double );  // 错误
    void func()
    {
        max( 87, 65 );          // 调用 libs_R_us::max( int, int )
        max( 35.5, 76.6 );      // 调用 libs_R_us::max( double, double )
    }
```

第一个 using 声明向全局域中引入了两个 libs_R_us::max()函数。于是，我们便可以在 func()中调用这两个 max()函数。函数调用时的实参类型将决定哪个函数会被调用。第二个 using 声明是个错误。用户不能在 using 声明中为一个函数指定参数表。对于 libs_R_us::pring()惟一有效的 using 声明是：

```
    using libs_R_us::pring;
```

using 声明总是为重载函数集合的所有函数声明别名。为什么这个限制是有必要的呢？这个限制可以确保名字空间 libs_R_us 的接口不会被破坏。很清楚，对如下的函数调用：

```
    print( 88 );
```

名字空间的作者希望调用函数 libs_R_us::pring(int)。由于某种原因，库的作者给出了几个不同的函数。若允许用户有选择地把一组重载函数中的一个函数、而不是全部函数加入到一个域中，那么这将导致令人吃惊的程序行为。

如果 using 声明向一个域中引入了一个函数，而该域中已经存在一个同名的函数，又会怎样呢？记住，using 声明只是一个声明。由 using 声明引入的函数就好像在该声明出现的地方被声明一样。因此，由 using 声明引入的函数重载了在该声明所出现的域中同名函数的其他声明。例如：

```
#include <string>
namespace libs_R_us {
     extern void print( int );
     extern void print( double );
}
extern void print( const string & );

// libs_R_us::print( int ) 和 libs_R_us::print( double )
// 重载 print( const string & )
using libs_R_us::print;

void fooBar( int ival )
{
     print( "Value: " ); // 调用全局 print( const string & )
     print( ival );    // 调用 libs_R_us::print( int )
}
```

using 声明向全局域中加入了两个声明：一个是 print(int)，一个是 print(double)。这些声明为名字空间 libs_R_us 中的函数提供了别名。这些声明被加入到 print()的重载函数集合中，它已经包含了全局函数 print(const string&)。当 fooBar()调用函数时，所有的 print()函数都将被考虑。

如果 using 声明向一个域中引入了一个函数，而该域中已经有同名函数且具有相同的参数表，则该 using 声明就是错误的。如果在全局域中已经存在一个名为 print(int)的函数，则 using 声明不能为名字空间 libs_R_us 中的函数声明别名 print(int)。例如：

```
namespace libs_R_us {
     void print( int );
     void print( double );
}

void print( int );
using libs_R_us::print;  // 错误: print(int) 的重复声明
void fooBar( int ival )
{
     print( ival );    // 哪一个 print? ::print 还是 libs_R_us::print?
}
```

我们已经知道了 using 声明是怎样影响重载函数的，现在让我们来了解一下 using 指示符又是怎样影响重载函数的。using 指示符使名字空间成员就像在名字空间之外被声明的一样。通过去掉名字空间的边界，using 指示符把所有声明加入到当前名字空间被定义的域中。如果在当前域中声明的函数与某个名字空间成员函数名字相同，则该名字空间成员函数被加入到重载函数集合中。例如：

```
#include <string>

namespace libs_R_us {
     extern void print( int );
     extern void print( double );
}
```

```
extern void print( const string & );

// using 指示符:
// print(int), print(double) 和 print(const string &)
// 是重载函数集的一部分
using namespace libs_R_us;
void fooBar( int ival )
{
     print( "Value: " ); // 调用 global print(const string &)
     print( ival ); // 调用 libs_R_us::print(int)
}
```

如果使用多个 using 指示符，情况也是这样。具有相同的名字、但是来自不同名字空间的成员函数都将被加到同一重载函数集合中。例如：

```
namespace IBM {
     int print(int);
}
namespace Disney {
     double print(double);
}

// using 指示符:
// 从不同的名字空间形成函数的重载集合
using namespace IBM;
using namespace Disney;
long double print(long double);
int main() {
     print(1);      // 调用 IBM::print(int)
     print(3.1);    // 调用 Disney::print(double)
     return 0;
}
```

在全局域中的函数 print()的重载集合含有函数 print(double)、print(int)以及 print(long double)。这些函数是 main()中调用该函数时需要被考虑的重载函数集，尽管这些函数最初是在不同的名字空间域中被声明的。

因此，同一重载函数集合中的函数都是在同一个域中被声明的，即使这些声明可能是用“使名字空间成员好像在其他域中声明的一样可见的 using 声明或 using 指示符”引入的。

9.1.5 extern "c" 和重载函数 ※

如 7.7 节所示，我们可以用链接指示符 extern "C"，来表示 C++程序中的某一个函数是用程序设计语言 C 编写的。链接指示符 extern "C"对重载函数声明的影响又会怎样呢？重载函数集合中的某些函数可以是 C++函数，而另外一些是 C 函数吗？

链接指示符只能指定重载函数集中的一个函数，例如，包含下列两个声明的程序是非法的：

```
// 错误: 在一个重载函数集中有两个 extern "C" 函数
extern "C" void print( const char* );
extern "C" void print( int );
```

下面 calc()的重载说明了在一个重载函数集合上的典型的链接指示符的用法：

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };

// 这个 C 函数可以在 C 和 C++程序中调用
// C++函数可以处理 C++类参数
extern "C" double calc( double );
extern SmallInt calc( const SmallInt& );
extern BigNum calc( const BigNum& );
```

C 语言的 calc()函数可以被 C 程序调用，也可以被 C++程序调用。其他函数是 C++函数，它们含有类参数，只能在 C++程序中被调用。声明的顺序并不重要。

链接指示符并不影响函数调用时对于函数的选择：只用参数类型来选择将被调用的函数。被选中的函数是与实参类型精确匹配的那个。例如：

```
SmallInt si = 8;
int main() {
    calc( 34 ); // 调用 C 写的 calc( double )
    calc( si ); // 调用 C++ 写的 calc( const SmallInt & )

    // ...
    return 0;
}
```

9.1.6 指向重载函数的指针 ※

我们可以声明一个指向重载函数集合里的某一个函数的指针。怎样做呢？例如：

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// pf1 指向哪个函数?
void ( *pf1 )( unsigned int ) = &ff;
```

因为 ff()是一个重载函数，所以只看初始化表达式&ff，编译器并不知道该选择哪个函数。为选择初始化该指针的函数，编译器要查找重载函数集合里与指针指向的函数类型只有相同的返回类型和参数表的函数。在上个例子中，选择的是 ff(unsigned int)。

如果没有函数与指针类型匹配，又该怎么办？如果是这样，将导致编译错误。例如：

```
extern void ff( vector<double> );
extern void ff( unsigned int );

// 错误: 无匹配: 无效参数表
void ( *pf2 )( int ) = &ff;

// 错误: 无匹配: 无效返同类型
double ( *pf3 )( vector<double> ) = &ff;
```

赋值的工作方式类似。如果一个重载函数的地址被赋值给一个函数指针，则该函数指针的类型被用来选择赋值符号右边的函数。如果编译器没有找到与指针类型匹配的函数，则赋值就是错误的，也就是说，在两个函数指针类型之间不能进行类型转换：

```
matrix calc( const matrix & );
```

```
int calc( int, int );
int ( *pc1 )( int, int ) = 0;
int ( *pc2 )( int, double ) = 0;

// ...
// ok: 匹配 int calc( int, int );
pc1 = &calc;

// 错误: 无匹配:  无效的第二个参数类型
pc2 = &calc;
```

9.1.7 类型安全链接 ※

重载允许同一个函数名以不同参数表出现多次，这是程序源代码层次上的词法便利。但是，大多数编译系统的底层组件要求每个函数名必须惟一。这是因为大多数链接编辑器都是按照函数名来解析外部引用的。如果链接编辑器看到两个以上的名为 print 的实例，它就不能通过分析类型来区分不同的实体（在编译到这一点时，类型信息通常已经不存在了）。链接编辑器会标记 print 被定义多次，并退出。

为处理这个问题，每个函数名及其相关参数表都被作为一个惟一的内部名**编码**（encoded）。编译系统的底层组件只能看到编码后的名字。名字转换的细节并不重要：在不同的编译器实现中，它们可能不同。一般的做法是把参数的个数和类型都进行编码，然后再将其附在函数名后面。

正如在 8.2 节关于全局函数的介绍中我们所看到的，这种特殊的编码可确保同名函数的两个声明（它们有不同的参数表，处于不同的文件中）不会被链接编辑器当作同一个函数的声明。因为这种编码帮助链接阶段区分程序中的重载函数，所以我们把它称作**类型安全链接**（type-safe linkage）。

这种特殊编码不适用于用链接指示符 extern "C"声明的函数。这就是为什么在重载函数集合中只有一个函数可以被声明为 extern "C"的原因，具有不同的参数表的两个 extern "C" 的函数会被链接编辑器视为同一函数。

练习 9.1

为什么我们要声明重载函数？

练习 9.2

应该怎样声明下面 error()函数的重载函数集合以处理下列调用？

```
int index;
int upperBound;
char selectVal;

// ...
error( "Array out of bounds: ", index, upperBound );
error( "Division by zero" );
error( "Invalid selection", selectVal );
```

练习 9.3

说出下列声明集合中第二个声明所造成的影响。

```
(a)  int calc( int, int );
     int calc( const int, const int );
(b)  int get();
     double get();
(c)  int *reset( int * );
     double *reset( double * );
(d)  extern "C" int compute( int *, int );
     extern "C" double compute( double *, double );
```

练习 9.4

下列哪些初始化是错误的？为什么？

```
(a)  void reset( int * );
     void (*pf)( void * ) = reset;
(b)  int calc( int, int );
     int (*pf1)( int, int ) = calc;
(c)  extern "C" int compute( int *, int );
     int (*pf3)( int*, int ) = compute;
(d)  void (*pf4)( const matrix & ) = 0;
```

9.2 重载解析的三个步骤

函数重载解析（function overload resolution）是把函数调用与重载函数集合中的一个函数相关联的过程。在存在多个同名函数的情况下，根据函数调用中指定的实参选择其中一个函数。考虑下面的例子：

```
T t1, t2;
void f( int, int );
void f( float, float );
int main() {
     f( t1,t2 );
     return 0;
}
```

这里，根据给出的类型 T，函数重载解析过程将决定 f(t1,t2)调用的是 f(int, int)还是 f(float, float)，还要决定是因为用实参 t1 和 t2 不能调用任何一个函数，还是由于调用中指定的实参与两个函数都精确匹配引起了**二义性**（ambiguous），使调用出错了。

函数重载解析过程是 C++程序设计语言中最复杂的部分之一。C++初学者在开始时可能会被它的全部细节吓倒。因此，本节只大概地浏览一下重载函数解析的过程，使你对发生的

事情有个感性认识。希望进一步了解的读者将在下两节中看到有关函数重载解析的更详细地描述。

函数重载解析的过程有三个步骤，我们将用下面的例子解释这三步：

```
void f(); void f( int );
void f( double, double = 3.4 );
void f( char*, char* );
int main() {
    f( 5.6 );
    return 0;
}
```

函数重载解析的步骤如下：

1．确定函数调用考虑的重载函数的集合，确定函数调用中实参表的属性。

2．从重载函数集合中选择函数，该函数可以在（给出实参个数和类型）的情况下用调用中指定的实参进行调用。

3．选择与调用最匹配的函数。

下面我们将按顺序查看每一步。

函数重载解析的第一步是确定对该调用所考虑的重载函数集合。该集合中的函数被称为**候选函数**（candidate function）。候选函数是与被调用函数同名的函数，并且在调用点上，它的声明可见。

在这个例子中，有四个候选函数：f()、f(int)、f(double, double)以及 f(char*, char*)。

函数重载解析的第一步还要确定函数调用中的参数表的属性，即实参的数目和类型。在本例中。实参表由一个 double 型的实参构成。

函数重载解析的第二步是从第一步找到的候选函数中选择一个或多个函数，它们能够用该调用中指定的实参来调用。因此，选出来的函数被称为**可行函数**（viable function）。可行函数的参数个数与调用的实参表中的参数数目相同，或者可行函数的参数个数多一些，但是每个多出来的参数都要有相关的缺省实参。对于每个可行函数，调用中的实参与该函数的对应的参数类型之间必须存在**转换**（conversion）。

在这个例子中，有两个可行函数，它们能够用调用中指定的实参表进行调用。

- f(int)是一个可行函数，因为它只有一个参数而且存在从实参类型 double 到参数类型 int 之间的转换。
- f(double, double)也是一个可行函数，因为它的第二个参数给出了缺省值，而第一个参数类型是 double，与实参类型精确匹配。

如果函数重载解析过程的第二步没有找到可以用给定的实参表调用的可行函数，则该调用就是错误的。没有函数与调用匹配，则说是**无匹配情况**（no match situation）。

函数重载解析的第三步选择与调用最匹配的函数，该函数被称为**最佳可行函数**（best viable function）[通常也称为**最佳匹配函数**（best match function）]。为了选择这个函数，从实参类型到相应可行函数参数所用的转换都被划分**等级**（ranked）。最佳可行函数是被适用于如下规则的函数：

1．应用在实参上的转换不比调用其他可行函数所需的转换差。

2．在某些实参上的转换要比其他可行函数对该参数的转换好。

类型转换及其等级划分将在 9.3 节详细讨论。这里我们只简要地查看一下本例子中转换的等级。当考虑可行函数 f(int)时，应用的转换是个标准转换，它将 double 型的实参转换成 int 型。当考虑可行函数 f(double)时，实参的类型 double 与相应的参数精确匹配。因为精确匹配比标准转换好（不做转换比任何转换都好），所以该调用的最佳可行函数是 f(double, double)。

如果函数重载解析的第三步没有找到最佳可行函数，则该函数调用是有二义的，即没有找到一个比其他可行函数都好的函数。

有关函数重载解析步骤的详细情况可在 9.4 节中找到，当一个重载的类成员函数被调用时，或一个重载的操作符函数被调用时函数重载解析也是适用的。15.10 节将讨论类成员函数重载解析的规则，而 15.11 节将讨论重载操作符的函数重载解析规则。函数重载解析过程还必须考虑函数模板生成的函数，10.8 节将讨论函数模板如何影响函数重载解析过程。

练习 9.5

函数重载解析过程的最后一步（第三步）发生的是什么？

9.3 参数类型转换 ※

在函数重载解析的第二步中，编译器确定“可以应用在函数调用的实参上的、将其转换成每个可行函数中相应参数类型”的转换，并将其划分等级。这种等级有二种可能：

1．**精确匹配**（exact match）：实参与函数参数的类型精确匹配。例如，给出重载函数集中的下列三个 printo 函数，则后面三个 print()调用都导致精确匹配：

```
void print( unsigned int );
void print( const char* );
void print( char );

unsigned int a;
print( 'a' ); // 匹配 print( char );
print( "a" ); // 匹配 print( const char* );
print( a );   // 匹配 print( unsigned int );
```

2．与一个**类型转换**（type conversion）匹配。实参不直接与参数类型匹配，但是它能转换成这样的类型：

```
void ff( char );
ff( 0 ); // 从 int 到 char 转换实参
```

3．**无匹配**（no match）：实参不能与声明的函数的参数匹配，因为在实参与相应的函数参数之间无法进行类型转换。下列两个 print()调用导致无匹配：

```
// print() 声明如下
int *ip;
class SmallInt { /* ... */ };
SmallInt si;
```

```
print( ip );  // 错误: 无匹配
print( si );  // 错误: 无匹配
```

精确匹配的实参并不一定与参数的类型完全一致，有一些最小转换可以被应用到实参上。在精确匹配的等级类别中可能存在的转换如下：

- 从左值到右值的转换
- 从数组到指针的转换
- 从函数到指针的转换
- 限定修饰转换

我们会在后面更详细地介绍这些转换。

“与一个类型转换匹配”的等级类别是三个等级中最复杂的一个，几种类型转换都必须考虑到。可能的转换被分成三组：**提升**（promotion）、**标准转换**（standard conversion）和**用户定义的转换**（user-defined conversions）。提升和标准转换在本节后面介绍。用户定义的转换将在详细讨论类（class）之后介绍。用户定义的转换由**转换函数**（conversion function）来执行，它是类的一个成员函数，允许一个类定义自己的“标准”转换。在第 15 章我们将看到类的转换函数以及涉及用户定义的转换的函数重载解析过程。

为一个函数调用选择最佳可行函数时，编译器会选择在实参的类型转换方面“最好”的一个函数。函数转换被划分等级如下：精确匹配比提升好，提升比标准转换好，标准转换比用户定义的转换好。我们将在 9.4 节进一步了解类型转换的等级。但是现在，我们描述的是各种可能的类型转换，本节中的某些例子会给出简单的情况，即怎样用这种等级划分来选择最佳可行函数。

9.3.1 精确匹配的细节

精确匹配最简单的例子是实参与函数参数类型精确匹配。例如，已知下面 max()重载函数集中的两个函数，则后面两个 max()调用中的实参与重载集合的特定函数的参数精确匹配

```
int max( int, int );
double max( double, double );

int i1;

void calc( double d1 ) {
     max( 56, i1 );    // 精确匹配 max( int, int );
     max( d1, 66.9 );  // 精确匹配 max( double, double );
}
```

枚举类型定义了一个惟一的类型，它只与枚举类型中的枚举值以及被声明为该枚举类型的对象精确匹配。例如：

```
enum Tokens { INLINE = 128; VIRTUAL = 129; };
Tokens curTok = INLINE;

enum Stat { Fail, Pass };

extern void ff( Tokens );
extern void ff( Stat );
```

```
extern void ff( int );
int main() {
     ff( Pass );    // 精确匹配 ff( Stat )
     ff( 0 );       // 精确匹配 ff( int )
     ff( curTok );  // 精确匹配 ff( Tokens )
     // ...
}
```

正如前面所提到的，即使一个实参必须应用一些最小的类型转换才能将其转换为相应函数参数的类型，它仍然是精确匹配的。

这些转换的第一个就是从左值到右值的转换。左值代表了一个可被程序寻址的对象，可以从该对象读取一个值，除非该对象被声明为 const，否则它的值也可以被修改。相对来说，右值只是一个表达式，它表示了一个值，或一个引用了临时对象的表达式，用户不能寻址该对象，也不能改变它的值。下面是一个简单的例子：

```
int calc( int );
int main() {
     int lval, res;

     lval = 5;      // 左值: lval; 右值: 5
     res = calc( lval );
                    // 左值: res;
                    // 右值: 存放 calc() 的返回值的临时对象
     return 0;
}
```

在第一个赋值表达式中，lval 是个左值，文字常量 5 是个右值。在第二个赋值表达式中，res 是个左值，函数 calc()调用返回值的临时对象是个右值。

在某些情况下，当预计出现一个值的时候，我们也可以用一个左值表达式来实现。例如：

```
int obj1;
int obj2;
int main() {
     // ...
     int local = obj1 + obj2;
     return 0;
}
```

obj1 和 obj2 是左值表达式。但是 main()中的加法只需要存贮在 obj1 和 obj2 中的值。在执行加法前，从 obj1 和 obj2 中把这些值抽取出来。从一个左值表达式所表示的对象中抽取值的动作就是一个“从左值到右值的转换”。

当一个函数期望一个按值传递的实参，而该实参又是一个左值的时候，就会执行从左值到右值的转换。例如：

```
#include <string>

string color( "purple" );
void print( string );

int main() {
     print( color );   // 精确匹配: 从左值到右值的转换
```

```
        return 0;
    }
```

因为 print()调用中的实参是按值传递的，所以发生了从左值到右值的转换，它从 color 中抽取出一个值，将其传递给 print(string)。即使发生了从左值到右值的转换，实参 color 也还是 print(string)的精确匹配。

不是所有函数调用都要求实参进行从左值到右值的转换。一个引用表示一个左值，所以当一个函数有一个引用参数时，被调用的函数接受一个左值。因此，不会有从左值到右值的转换被应用到相应的引用参数的实参上。例如，已知函数：

```
#include <list>
void print( list<int> & );
```

下列调用中的 li 是一个左值，代表被传递给函数 print()的 list<int>对象。

```
list<int> li(20);

int main() {
    // ...
    print( li ); // 精确匹配：没有从左值到右值的转换
    return 0;
}
```

li 与引用参数的绑定是个精确匹配。

精确匹配允许的第二种转换是从数组到指针的转换。如在 7.3 节中所提到的，函数参数没有数组类型，取而代之的是参数被转换成指向数组首元素的指针。类似地，类型为 NT 数组（这里 N 是数组元素的个数，T 是数组元素的类型）的实参总是被转换成 T 型的指针。实参类型的转换是从数组到指针的转换。即使发生了转换，实参仍然被看作是 T 型指针参数的精确匹配。例如：

```
int ai[3];
void putValues(int *);

int main() {
    // ...
    putValues(ai); // 精确匹配：从数组到指针的转换
    return 0;
}
```

在函数 putValues()被调用之前，发生了从数组到指针的转换，将实参 ai 从三个 int 的数组转换成 int 型的指针。即使 putValues()有一个指针参数，且在实参上发生了从数组到指针的转换，但是该实参也仍是 putValues()调用的精确匹配。

精确匹配允许的下一种转换是从函数到指针的转换，该转换在 7.9 节中已简要介绍过了。和数组类型参数一样，函数类型的参数自动被转换成指向函数的指针。函数类型的实参也自动被转换成函数指针类型。这种实参类型的转换被称为从函数到指针的转换。即使发生了这种转换，该实参仍被看作是函数指针类型参数的精确匹配。例如：

```
int lexicoCompare( const string &, const string & );

typedef int (*PFI)( const string &, const string & );
void sort( string *, string *, PFI );
```

```
string as[10];
int main()
{
    // ...
    sort( as,
        as + sizeof(as) / sizeof(as[0] - 1),
        lexicoCompare// 精确匹配：从函数到指针的转换
    );

    return 0;
}
```

在函数 sort()被调用之前，发生了从函数到指针的转换，它将实参 lexicoCompare 从函数类型转换成函数指针类型。即使该函数期望接收的是一个指针而实参是一个函数名，即使发生了从函数到指针的转换，该实参也仍然是 sort()的第三个参数的精确匹配。

精确匹配的最后一种转换是限定修饰转换，这种转换只影响指针。它将限定修饰符 const 或 volatile（或两者）加到指针指向的类型上。例如：

```
int a[5] = { 4454, 7864, 92, 421, 938 };
int *pi = a;
bool is_equal( const int * , const int * );
int func( int *parm ) {

    // pi 和 parm 的精确匹配：限定修饰转换
    if ( is_equal( pi, parm ) )

    // ...
    return 0;
}
```

在函数 is_equal()被调用之前，实参 pi 和 parm 被从 int 型指针转换成指向 const int 型的指针。该转换把 const 限定修饰符加到指针指向的类型上，所以是限定修饰转换。即使函数期望两个 const int 的指针，而两个实参是指向 int 型的指针，这两个实参也仍然是 is_equal()的参数的精确匹配。

限定修饰转换只应用在指针指向的类型上。当参数是 const 或 volatile 类型，而实参不是时，没有类型转换发生：

```
extern void takeCI( const int );
int main() {
    int ii = ...;

    takeCI(ii); // 无转换发生
    return 0;
}
```

在 takedCI()调用中，即使参数是 const int 型，也不会有限定修饰转换被应用在 int 型的实参 li 上 。该实参是函数参数类型的精确匹配。

如果实参是指针，且有 const 或 volatile 限定符应用在指针上，也是这样：

```
extern void init( int *const );
```

```
extern int *pi;
int main() {

    // ...
    init(pi); // 没有限制转换
    return 0;
}
```

由于 init()参数上的 const 限定修饰符只应用在指针本身上，而并没有应用在指针指向的类型上。因此，编译器在考虑应用在实参上的转换时不会考虑 const 限定修饰符。因为没有限定修饰转换被应用在实参 pi 上，所以该实参与函数参数类型精确匹配。

精确匹配类别中的前三种转换（从左值到右值、从数组到指针以及从函数到指针的转换）通常被称为**左值转换**（lvalue transformation）。正如在 9.4 节中即将看到的那样，虽然左值转换和限定修饰转换都属于精确匹配类别，但是只需要左值转换的精确匹配比需要限定修饰转换的要好。我们将在下节中更详细地讨论这些。

精确匹配可以用一个显式强制转换强行执行。例如，已知重载函数集合：

```
extern void ff(int);
extern void ff(void *);
```

如下调用；

```
ff( 0xffbc ); // 调用 ff(int)
```

与 ff(int)精确匹配，因为 0xffbc 是十六进制形式的 int 型文字常量。程序员可以如下提供一个显式转换来强制调用 ff(void*)。

```
ff( reinterpret_cast<void *>(0xffbc) );   // 调用 ff(void*)
```

显式强制转换应用在实参上时，实参的类型就变成强制转换的结果。使用显式强制转换的类型转换可以帮助指导函数重载解析。例如，如果因为实参与两个以上可行函数匹配，使得函数重载解析的结果是二义的，则可以用显式强制转换来打破二义性，使函数调用被解析为一个特殊的可行函数。

9.3.2 提升的细节

提升实际上就是下列转换之一：

- char、unsigned char 或 short 型的实参被提升为 int 型。如果机器上 int 型的字长比 short 整型的长，则 unsigned short 型的实参被提升到 int 型；否则，它被提升到 unsigned int 型。
- float 型的实参被提升到 double 类型。
- 枚举类型的实参被提升到下列第一个能够表示其所有枚举常量的类型：int、unsigned int、long 或 unsigned long。
- 布尔型的实参被提升为 int 型。

当实参的类型是上面描述的源类型之一，而函数参数的类型是相应被提升的类型时，则应用该提升。例如：

```
extern void manip( int );
```

```
int main() {
    manip( 'a' ); // 类型 char 被提升为 int
    return 0;
}
```

字符文字的类型是 char，它的提升类型是 int。因为提升的类型与函数 manip()的参数类型匹配，所以我们说函数调用要求提升它的实参。

假设有下列例子：

```
extern void print( unsigned int );
extern void print( int );
extern void print( char );

unsigned char uc;
print( uc ); // print( int ): uc 只需要提升
```

在 unsigned char 类型只占一个字节、而 int 型占四个字节的机器上，由于类型 int 可以表示 unsigned char 型的全部值，所以提升就是将一个 unsigned char 的实参变成 int 型。在上面给定的重载函数声明，以及刚刚描述的结构中，与 unsigned char 型实参最匹配的函数是 print(int)，要匹配其他两个函数则要求应用标准转换。

下面的例子说明了枚举型实参的提升：

```
enum Stat { Fail, Pass };

extern void ff( int );
extern void ff( char );

int main() {
    // ok: 枚举常量 Pass 被提升到 int
    ff( Pass );       // ff( int )
    ff( 0 );          // ff( int )
    return 0;
}
```

枚举类型的提升有时候会使人惊奇，编译器经常根据枚举常量的值来选择枚举类型的表示。例如，假设有前面描述的结构（char 有一个字节，int 有四个字节）以及下面的枚举类型：

```
enum e1 { a1, b1, c1 };
```

因为只有三个枚举常量——a1、b1 和 c1——它们的值分别为 0、1、2，该枚举类型的所有值都可以用 char 型表示，所以编译器常常会选择 char 型作为 e1 的表示。但是，假设我们有另外一个枚举类型 e2，它有不同的枚举常量值：

```
enum e2 { a2, b2, c2=0x80000000 };
```

因为有一个枚举常量的值是 0x80000000，所以该编译器被迫为 e2 选择一个能够表示值 0x80000000 的表示，这个表示就是 unsigned int。

因此，即使 e1 和 e2 都是枚举类型，它们的表示也并不相同。这使 e1 和 e2 被提升为不同的类型，例如：

```
#include <string>

string format( int );
string format( unsigned int );
```

```
int main() {
     format(e1);   // 调用 format( int )
     format(e2);   // 调用 format( unsigned int )
     return 0;
}
```

在第一个 format()的调用中，因为实参的类型是 char 型表示的类型 e1，所以实参被提升为 int 型，为该调用选择的函数是 format(int)。在 format()的第二个调用中，因为实参的类型是 unsigned int 型表示的 e2 型，所以实参被提升为类型 unsigned int。这使得函数 format(unsigned int)被选择给第二个调用。因此，你应该知道：两个枚举类型在重载函数解析期间的行为可能完全不同，解析过程根据枚举常量的值来决定它们被提升的类型。

9.3.3 标准转换的细节

有五种转换属于标准转换：

1．整值类型转换：从任何整值类型或枚举类型向其他整值类型的转换（不包括前面提升部分中列出的转换）。

2．浮点转换：从任何浮点类型到其他浮点类型的转换（不包括前面提升部分中列出的转换）。

3．浮点—整值转换：从任何浮点类型到任何整值类型或从任何整值类型到任何浮点类型的转换。

4．指针转换：整数值 0 到指针类型的转换和任何类型的指针到类型 void*的转换。

5．bool 转换：从任何整值类型、浮点类型、枚举类型或指针类型到 bool 型的转换。

下面是一些例子：

```
extern void print( void* );
extern void print( double );
int main() {
     int i;
     print( i );   // 匹配 print( double );
                   // i 被一个标准转换从 int 转换到 double
     print( &i );  // 匹配 print( void* );
                   // &i 被标准转换从 int* 转换到 void*

     return 0;
}
```

类别 1、2 和 3 中的转换是有潜在危险的转换，这是因为转换的目标类型不能表示源类型的全部值。例如，类型 float 不能表示出 int 类型的所有值的精度。这也是这些类别中的转换是标准转换而不是提升转换的原因。

```
int i;
void calc( float );

int main() {
     calc( i );    // 浮点—整值标准转换
                   // 潜在危险，取决于 i 值
     return 0;
```

当用户调用函数 calc()时，浮点—整值标准转换把实参从 int 型转换成 float 型。根据存储在 i 中值的情况，可能无法保证把 i 的值存储在类型 float 的参数内并且不损失精度。

所有的标准转换都被视为是等价的。例如，从 char 到 unsigned char 的转换并不比从 char 到 double 的转换优先级高。类型之间的接近程度不被考虑。即，如果有两个可行函数要求对实参进行标准转换以便匹配各自参数的类型，则该调用就是二义的，将被标记为编译错误。例如，下面给出的一对重载函数：

```
extern void manip( long );
extern void manip( float );
```

下列调用是二义的：

```
int main() {
     manip( 3.14 );      // 错误: 二义性
                         // manip( float ) 也不会好到那里
     return 0;
}
```

文字常量 3.14 是 double 型的，通过标准转换两个函数都能匹配。因为可能存在有两种标准转换，所以该调用被标记为二义的。没有一个标准转换比其他的标准转换更为优先。程序员可以用显式强制转换来解决二义性的问题，比如：

```
manip( static_cast<long>( 3.14 ) );    // manip( long )
```

或通过用 float 常量后缀：

```
manip( 3.14F );    // manip( float )
```

下面是一些其他函数调用的例子，因为它们都与重载函数集中的多个函数匹配，所以它们都是二义的，并都被标记为错误：

```
extern void farith( unsigned int );
extern void farith( float );
int main() {
     // 每个调用都是二义的
     farith( 'a' );      // 实参类型为 char
     farith( 0 );        // 实参类型为 int
     farith( 2uL );      // 实参类型为 unsigned long
     farith( 3.14159 );  // 实参类型为 double
     farith( true );     // 实参类型为 bool
     return 0;
}
```

有时标准指针转换看起来有些违反直觉。尤其是，0 可以被转换成任何指针类型：这样创建的指针值被称为**空指针值**（null pointer value）。同时，值 0 也可以是任何整型常量表达式。例如：

```
void set(int*);

int main() {
     // 从 0 到 int* 的指针转换应用到两个实参上
     set( 0L );
     set( 0x00 );
     return 0;
```

```
    return 0;
}
```

常量表达式 0L（long int 型的 0）以及常量表达式 0x00（十六进制的 0）都属于整型类型因此能够被转换成 int*型的空指针值。

但是，因为枚举类型不是整型，仍为 0 的枚举型值不能被转换成指针类型。例如：

```
enum EN { zr = 0 };
set( zr );    // 错误: zr 不能被转换到 int*
```

对 set()的调用是错的，因为在枚举值 zr 和 int*型的参数之间不存在可能的转换，即使该枚举值为 0。

还有一些事情要注意，常量表达式 0 属于类型 int。把这常量表达式转换成指针类型的标准转换是必需的。如果重载函数集中有一个函数，它的参数是 int 型，则对于实参 0，该函数会被优先考虑。例如：

```
void print( int );
void print( void * );
void set( const char* );
void set( char* );

int main() {
    print( 0 );  // 调用 print( int )
    set( 0 ); // 二义
    return 0;
}
```

实参对 print(int)的调用是精确匹配。但是，为了调用 print(void*)，需要一个标准转换将 0 转换成指针类型。因为精确匹配比标准转换要好，所以该调用选择了函数 print(int)。对 set()的调用是二义的，因为通过应用标准转换，0 与两个 set()函数的参数都匹配。且两个函数对该调用一样好，所以它是二义的。

最后一种指针转换允许将任何指针类型的实参转换成 void*型的参数，因为 void*是通用的数据类型指针，所以它可以存放任何数据类型的指针值。下面是一些例子：

```
#include <string>

extern void reset( void * );
int func( int *pi, string *ps ) {
    // ...
    reset( pi ); // 指针转换: int* 到 void*
    // ...
    reset( ps ); // 指针转换: string* 到 void*
    return 0;
}
```

只有指向数据类型的指针才可以用指针标准转换将其转换成类型 void*，函数指针不能用标准转换转换成类型 void*。例如：

```
typedef int (*PFV)();
extern PFV testCases[10];    // 函数指针数组
extern void reset( void * );
```

```
int main() {
    // ...
    reset( testCases[0] );  // 错误: 在 int(*)() 之间不存在标准转换

    return 0;
}
```

9.3.4 引用

函数调用的实参或函数参数都可以是引用。那么，引用又是怎样影响类型转换规则的呢？

首先，我们来看一下如果实参是一个引用时会发生什么情况。实参的类型永远不会是引用类型。当实参是一个引用时，该实参是一个左值，它的类型是引用所指的对象的类型。考虑下列例子：

```
int i;
int & ri = i;
void print( int );
int main() {
    print( i );   // int 型的左值实参
    print( ri ); // 同样
    return 0;
}
```

两个函数调用的实参都是 int 型，而在第二个调用中引用被用作实参，对实参类型没有任何影响。

当一个实参是类型 T 的引用时，所考虑的标准转换和提升与该实参是 T 型对象时的一样。例如：

```
int i;
int& ri = i;
void calc( double );
int main() {
    calc( i );    // 浮点–整值标准转换
    calc( ri );   // 同样
    return 0;
}
```

那么，引用参数是怎样影响应用在实参上的转换的呢？实参与引用参数的匹配结果有下面的两种可能。

1．实参是引用参数的合适的初始值。在这种情况下，我们说该实参是参数的精确匹配。例如：

```
void swap( int &, int & );
int manip( int i1, int i2 ) {
    // ...
    swap( i1, i2 );   // ok: 调用 swap( int &, int & )
    // ...
    return 0;
```

```
}
```

2．实参不能初始化引用参数。在这种情况下，没有匹配情况发生，实参不能被用来调用该函数。例如：

```
int obj;
void frd( double & );
int main() {
     frd( obj );  // 错误: 参数必须是 const double &
     return 0;
}
```

对 frd()的调用是错误的。实参类型是 int，必须被转换成 double 以匹配引用参数的类型。该转换的结果是个临时值。因为这种引用不是 const 型的，所以临时值不能被用来初始化该引用。

下面是在引用参数与实参之间没有匹配的另外一个例子。

```
class B;
void takeB( B& );
B giveB();

int main() {
     takeB( give() );  // 错误: 参数必须是 const B&
     return 0;
}
```

对 takeB()的调用是错误的。实参是函数调用的返回值，它是一个临时值，不能被用来初始化非 const 型的引用。

在这两种情况下，如果引用参数是 const 型的引用，则实参就是参数的精确匹配。针对下面给出的代码：

```
void print( int );
void print(int&);

int iobj;
int &ri = iobj;

int main() {
     print( iobj );    // 错误: 二义
     print( ri );      // 错误: 二义
     print( 86 );      // ok: 调用 print( int )
     return 0;
}
```

第一个函数调用是错误的。因为对象 iobj 是与两个函数 print()都精确匹配的实参，所以函数调用是二义的。对第二个函数调用也一样。引用 ri 指向一个对象，它是两个函数的精确匹配。但是，第三个调用是正确的。函数 print(int&)不是该调用的可行函数。整型常量是个右值，不是非 const 引用参数的有效的初始值。在调用 print(86)的可行函数集中只有一个函数 print(int)。因为它是惟一的可行函数，所以它是该调用选择的函数。

简而言之，对于引用参数来说，如果实参是该引用的有效初始值，则该实参是精确匹配。如果该实参不是引用的有效初始值，则不匹配。

练习 9.6

指出精确匹配中允许的两个最小转换。

练习 9.7

在下列函数调用中，实参上的每个转换的等级是什么？

```
(a)  void print( int *, int);
     int arr[6];
     print( arr, 6 );              // 函数调用
(b)  void manip( int, int );
     manip( 'a', 'z' );            // 函数调用
(c)  int calc( int, int);
     double dobj;
     double = calc( 55.4, dobj );  // 函数调用
(d)  void set( const int * );
     int *pi;
     set( pi );                    // 函数调用
```

练习 9.8

下列哪个函数调用是因为实参与函数参数之间不存在类型转换而发生错误？

```
(a)  enum Stat { Fail, Pass };
     void test( Stat );
     test( 0 );                    // 函数调用
(b)  void reset( void * );
     reset( 0 );                   // 函数调用
(c) void set( void * );
     int *pi;
     set( pi );                    // 函数调用
(d) #include <list>
     list<int> oper();
     void print( list<int> & );
     print( oper() );              // 函数调用
(e)  void print( const int );
     int iobj;
     print( iobj );                // 函数调用
```

9.4 函数重载解析细节 ※

如 9.2 节所述，函数重载解析过程有三个步骤。这些步骤可以总结如下：

1. 确定为该调用而考虑的候选函数，以及函数调用中的实参表属性。
2. 从候选函数中选出可行函数。也就是说；根据调用中指定的实参、实参数目和类型，

选择可以被调用的函数。

3．对于“被用来将实参转换成可行函数参数类型的转换”划分等级，以便选出与调用最匹配的函数。

下面，我们来详细讨论这三个步骤。

9.4.1 候选函数

候选函数与被调用的函数具有同样的名字。可以用下面两种方式找到候选函数：

1．该函数的声明在调用点上可见。给出下列例子：

```
void f();
void f( int );
void f( double, double = 3.4 );
void f( char*, char* );
int main() {
     f( 5.6 );          // 这个调用有四个候选函数
     return 0;
}
```

因为在全局域中声明的四个 f()在调用点上都可见。所以它们都是候选函数集的一部分。

2．如果函数实参的类型是在一个名字空间中被声明的，则该名字空间中与被调用函数同名的成员函数也将被加入到候选函数集中。例如：

```
namespace NS {
     lass C { /* ... */ };
     oid takeC( C& );
}
// cobj 的类型是在名字空间 NS 中被声明的类 C
NS::C cobj;

int main() {
                          // 在调用点没有 takeC()可见
     takeC( cobj );       // ok: 调用 NS::takeC( C& )
                          // 因为实参类型是 NS::C
                          // 所以考虑在名字空间 NS 中声明的函数 takeC()
     return 0;
}
```

因此，候选函数是“在调用点上可见的函数”以及“在实参类型所在的名字空间中声明的同名函数”的集合。

当我们确定在调用点上可见的重载函数集合时，我们在前面看到的关于怎样生成重载函数集的规则仍然适用。

在嵌套的域中被声明的函数隐藏了而不是重载了外围域中的同名函数。这种情况下的候选函数是在嵌套域中被声明的函数，即没有被该函数调用隐藏的函数。在下面的例子中，在调用点上可见的候选函数是 format(double)和 format(char*)：

```
char* format( int );
void g() {
     char* format( double );
```

```
        char* format( char* );
        format(3);          // 调用 format( double )
    }
```

因为在全局域中声明的函数 format(int)被隐藏，所以它没有被包含在候选函数集中。

在调用点上可见的 using 声明也可以引入候选函数。考虑下列例子：

```
    namespace libs_R_us {
        int max( int, int );
        double max( double, double );
    }

    char max( char, char );

    void func()
    {
        // 名字空间的函数不可见
        // 这三个调用分别调用全局函数 max( char, char )
        max( 87, 65 );
        max( 35.5, 76.6);
        max( 'J', 'L' );
    }
```

名字空间 libs_R_us 中定义的函数 max()在调用点上不可见，惟一可见的是全局域中声明的函数 max()。该函数是候选函数集中惟一的一个函数：它是 func()中三个调用所调用的函数。我们可以用 using 声明使名字空间 libs_R_us 中声明的函数 max()变为可见。那么，using 声明应该放在哪儿呢？如果把 using 声明放在全局域中，

```
    char max( char, char );
    using libs_R_us::max;     // using 声明
```

那么，来自名字空间 libs_R_us 中的函数 max()就将被加到重载函数集中，该集合同时还包含全局域中声明的函数 max()。现在，三个函数在 func()中都可见，并且都成为侯选函数集中的一部分。随着三个函数在调用点上可见，func()中的调用被解析如下：

```
    void func()
    {
        max( 87, 65 );       // 调用 libs_R_us::max( int, int )
        max( 35.5, 76.6 );   // 调用 libs_R_us::max( double, double )
        max( 'J', 'L' );     // 调用 max( char, char )
    }
```

但是，如果我们在函数 func()的局部域中如下引入了 using 声明又会怎么样呢？

```
    void func()
    {
        // using 声明
        using libs_R_us::max;

        // 函数调用如上
    }
```

候选函数集中会包含哪些 max()？请回忆一下 using 声明的嵌套。由于局部域中的 using 声明，全局函数 max(char, char)被隐藏。在调用点上可见的函数只是：

```
libs_R_us::max( int, int )
libs_R_us::max( double, double )
```

这两个函数是候选函数集中的函数，func()中的调用被解析如下：

```
void func()
{
    // using 声明
    // 全局 max( char, char ) 被隐藏
    using libs_R_us::max;

    max( 87, 65 );        // 调用 libs_R_us::max( int, int )
    max( 35.5, 76.6 );    // 调用 libs_R_us::max( double, double )
    max( 'J', 'L' );      // 调用 libs_R_us::max( int, int )
}
```

using 指示符也会影响候选函数集的构成。假设我们决定用 using 指示符而不是 using 声明使名字空间 lib_R_us 中的函数 max()在 func()中可见。例如，使用下面全局域中的 using 指示符，候选函数集就将包含全局函数 max(char,char)以及在名字空间 libs_R_us 中声明的函数 max(int, int)和 max(double, double)：

```
namespace libs_R_us {
    int max( int, int );
    double max( double, double );
}
char max( char, char );

using namespace libs_R_us; // using 指示符

void func()
{
    max( 87, 65 );        // 调用 libs_R_us::max( int, int )
    max( 35.5, 76.6 );    // 调用 libs_R_us::max( double, double )
    max( 'J', 'L' );      // 调用 ::max( char, char )
}
```

假若像下面这样，将 using 指示符放到 func()的局部域内，又会怎么样呢？

```
void func()
{
    // using 指示符
    using namespace libs_R_us;

    // 函数调用如上
}
```

哪些 max()会成为候选函数？记住，using 指示符使名字空间成员可见就好像它们是在名字空间之外、在定义名字空间的位置上被声明的一样。在我们的例子中，名字空间 libs_R_us 的成员在 func()的局部域内可见，就好像该成员已经在名字空间之外、全局域内被声明的样。这暗示着在 func()内可见的重载函数集与前面包含下列三个函数的一样：

```
max( char, char )
libs_R_us::max( int, int )
libs_R_us::max( double, double )
```

无论 using 指示符出现在全局域还是 func()的局部域内，都不会影响 func()中的调用的解析过程：

```
void func()
{
    using namespace libs_R_us;
    max( 87, 65 );       // 调用 libs_R_us::max( int, int )
    max( 35.5, 76.6 );   // 调用 libs_R_us::max( double, double )
    max( 'J', 'L' );     // 调用 ::max( char, char )
}
```

所以，候选函数集是在调用点上可见的函数（包括 using 声明和 using 指示符引入的函数）以及在与实参类型相关的名字空间内被声明的成员函数。例如：

```
namespace basicLib {
    int print( int );
    double print( double );
}
namespace matrixLib {
    class matrix { /* ... */ };
    void print( const matrix & );
}

void display()
{
    using basicLib::print;
    matrixLib::matrix mObj;
    print( mObj );       // 调用 matrixLib::print( const matrix& )
    print( 87 );         // 调用 basicLib::print( int )
}
```

哪些函数是调用 print(mObj)的候选函数？因为由函数 display()中的 using 声明引入的函数 basicLib::print(int)和 basicLib::print(double)在调用点上可见，所以它们都是候选函数。因为函数调用实参的类型是 matrixLib::matrix，所以在名字空间 matrixLib 中声明的函数 print()也是个候选函数。调用 print(87)的候选函数是哪些呢？在调用点上只有函数 basicLib::print(int)和 basicLib::print(double)可见，所以它们是候选函数。因为实参的类型是 int，所以编译器不会在其他名字空间中寻找其他候选函数。

9.4.2 可行函数

可行函数是候选函数集合中的函数。它的参数表或者与调用中的实参数目相同，或者有更多的参数。在后一种情况下，额外的参数会被给出缺省实参，以便可以用实参表中指定的实参调用该函数。可行函数是这样的函数：对于每个实参，都存在到函数参数表中相应的参数类型之间的转换。可被考虑的转换是 9.3 节中介绍的转换。

在下面的例子中，对调用 f(5.6)来说有两个可行函数：它们是 f(int)和 f(double)。

```
void f();
void f( int );
void f( double );
```

```
void f( char*, char* );
int main() {
     f( 5.6 );        // 两个可行函数: f( int ) 和 f( double )
     return 0;
}
```

f(int)是可行函数。因为它只有一个参数，这与函数调用中实参的数目匹配，并且存在着把实参从 double 型转换成 int 型的标准转换。f(double)也是个可行函数。这个可行函数只有一个参数，类型为 double，是调用中实参的精确匹配。候选函数 f()和 f(char*, char*)被排除在可行函数集合之外，是因为这些函数不能用一个实参调用。

在下面的例子中，调用 format(3)的唯一可行函数是函数 format(double)。虽然候选函数 format(char*)也可以用一个实参调用，但是在 int 型的实参和 char*型的参数之间不存在转换。就因为不存在该类型转换，所以该函数被排除在可行函数集合之外。

```
char* format( int );

void g() {
     // 全局函数 format( int ) 被隐藏
     char* format( double );
     char* format( char* );
     format(3);       // 只有一个可行函数: format( double )
}
```

在下面的例子中，三个候选函数都在 func()中 max()调用的可行函数集合中。这三个函数都可以用两个实参来调用。因为实参类型是 int，它是 libs_R_us::max(int,int)的参数的精确匹配，所以这两个实参可以通过“浮点—整值标准转换”转换成 libs_R_us::max(double,double)的参数，以及通过“整值标准转换”转换成 max(char,char)的参数。

```
namespace libs_R_us {
     int max( int, int );
     double max( double, double );
}

// using 声明
using libs_R_us::max;
char max( char, char );

void func()
{
     // 这三个 max() 都是可行函数
     max( 87, 65 );   // 调用 libs_R_us::max( int, int )
}
```

注意，对于有多个参数的候选函数来说，只要函数调用中的一个实参不能被转换成候选函数参数表中相应的参数，它就将马上被排除在可行函数集合之外，即使其他实参都存在转换。在下面的例子中，函数 min(char*,int)被排除在可行函数之外，因为在第一个实参 int 类型与相应函数参数 char*类型之间不存在转换。即使第二个实参是函数的第二个参数的精确匹配，该函数也会被排除：

```
extern double min( double, double );
extern int min( char*, int );
```

```
void func()
{
     // 候选函数 min( double, double )
     min( 87, 65 );    // 调用 min( double, double )
}
```

如果在去掉参数个数不同的候选函数（或去掉不存在合适的类型转换的候选函数）之后，没有可行函数存在，则该调用就会导致编译时刻错误。在这种情况下，我们就说没有找到匹配。

```
void print( unsigned int );
void print( char* );
void print( char );

int *ip;
class SmallInt { /* ... */ };
SmallInt si;

int main() {
     print( ip );      // 错误：没有可行函数：没有匹配
     print( si );      // 错误：没有可行函数：没有匹配
     return 0;
}
```

9.4.3 最佳可行函数

最佳可行函数是具有与实参类型匹配最好的参数的可行函数。对于每个可行函数来说，每个实参的类型转换都被划分了等级，以决定每个实参与其相应参数的匹配程度（9.2 节描述了得到支持的类型转换。）。最佳可行函数是满足下列条件的可行函数：

1．用在实参上的转换不比调用其他可行函数所需的转换更差

2．在某些实参上的转换要比其他可行函数对该参数的转换更好。

将实参转换成相应的函数参数时可能不只应用一种类型转换。例如，在下面的例子中：

```
int arr[3];
void putValues(const int *);

int main() {
     putValues(arr);   // 在转换序列中有 2 个转换
                       // 数组到指针、限定修饰转换
     return 0;
}
```

将实参 arr 从三个 int 元素的数组转换成 const int 指针类型应用了一个转换序列，该转换序列由下列转换构成：

1．从数组到指针的转换。将实参从三个 int 元素的数组转换成 int 型的指针。

2．限定修饰转换。把 int 型的指针转换成 const int 型的指针。

因此说一个**转换序列**（conversion sequence）被用来把实参转换成可行函数参数的类型更为合适。因为是一个转换序列而不是单个转换被应用到实参上将其转换成相应参数的类型，所以函数重载解析的第三步是将转换序列划分等级。

转换序列的等级是构成该序列最坏转换的等级。正如在 9.2 节中描述的，类型转换的等级划分如下：精确匹配好于提升，提升好于标准转换、在前面的例子中，序列中的两个转换都具有精确匹配的等级。

一个转换序列潜在地由下列转换以下列顺序构成：

左值转换——>

提升或者标准转换——>

限定修饰转换

左值转换（lvalue transformation）是指 9.2 节里讲的精确匹配类别中描述的前三个转换：从左值到右值的转换、从数组到指针的转换和从函数到指针的转换。转换序列的构成是这样的：首先是 0 个或一个左值转换，接着是 0 个或一个提升、或者 0 个或一个标准转换，再后面是 0 个或一个限定修饰转换。至多，每种转换会有一个被应用上，以将实参转换成相应的参数。

这种转换序列被称为**标准**（standard）转换序列。还有另外一种转换序列被称为**用户定义的**（user-defined）转换序列。用户定义的转换序列包含类成员转换函数，类成员转换函数和用户定义的转换序列将在 15 章讲述。

下面的例子中实参上的转换序列是什么？

```
namespace libs_R_us {
     int max( int, int );
     double max( double, double );
}
// using 声明
using libs_R_us::max;
void func()
{
     char c1, c2;
     max( c1, c2 );    // 调用 libs_R_us::max( int, int )
}
```

在 max()的调用中的实参是 char 型的。调用函数 libs_R_us::max(int, int)的实参上的转换序列如下：

1a．因为该实参按值传递，所以首先是从左值到右值转换，它从实参 c1 和 c2 中抽取值。

2a．提升转换将实参从 char 转换到 int。

调用函数 libs_R_us::max(double,double)的实参上的转换序列如下：

1b．先应用一个从左值到右值的转换，它从实参 c1 和 c2 抽取值。

2b．浮点——整值标准转换将实参从 char 转换成 double。

第一个转换序列的等级是提升（序列中最差的转换），而第二个转换序列的等级是标准转换。因为提升比标准转换好，所以函数 libs_R_us::max(int,int)被选为该调用的最佳可行函数，或最佳匹配函数。

如果通过对实参上的转换序列划分等级，仍然不能够判别出一个可行函数比其他函数更匹配实参的类型，则该调用就是二义的。在下面的例子中，calc()的两个实例都要求下列转换序列：

1．首先是一个从左值到右值的转换，它从实参 i 和 j 中抽取值。

2．通过一个标准转换把实参转换成相应的参数。

因为每个转换序列都和另一个一样好，所以该调用是二义的：

```
int i, j;
extern long calc( long, long );
extern double calc( double, double );

void jj() {
     // 错误：二义，没有最佳匹配
     calc( i, j );
}
```

限定转换（把 const 或 volatile 修饰符加到指针指向的类型上的转换）具有精确匹配的等级。但是，如果两个转换序列前面都相同，只是一个在序列尾部有一个额外的限定转换，则另一个没有额外限定转换的序列比较好。例如：

```
void reset( int * );
void reset( const int * );

int* pi;

int main() {
     reset( pi ); //没有限定转换的比较好；选择 reset( int * )
     return 0;
}
```

应用在调用第一个候选函数 reset(int*)的实参上的标准转换序列是个精确匹配：它只要求一个从左值到右值的转换来抽取实参的值。对第二个候选函数 reset(const int*)，也应用了一个从左值到右值的转换，接着是限定修饰转换把结果值从 int 指针转换成 const int 指针。这两个序列都是精确匹配，但是上面的函数调用不是二义的，因为这两个转换序列的第一个转换相同，但是第二个转换序列末尾有额外的限定修饰转换，所以第一个没有限定修饰转换的序列被认为是较好的匹配。因此，可行函数 reset(int*)是最佳可行函数。

下面是另一个例子，其中限定修饰转换影响了被选择的转换序列：

```
int extract( void * );
int extract( const void * );

int* pi;

int main() {
     extract( pi ); // 选择 extract( void * )
     return 0;
}
```

该调用有两个可行函数：extract(void*)和 extract(const void*)。在调用第一个可行函数 extract(void*)上应用的转换序列中，有一个从左值到右值的转换来抽取实参的值，接着是一个标准指针转换，它将该值从一个 int 指针转换成一个 void 指针。应用在调用第二个函数 extract(const void*)上的转换序列也是如此，只不过还应用了一个额外的限定修饰转换，它将结果从 void 指针转换成 const void 指针。因为这两个转换序列，除了第二个转换序列在尾部

是一个额外的限定修饰转换外，其余都是相同的，所以第一个转换序列被选择为更好的转换序列。函数 extract(void*)被选为该实参的最佳可行函数。

const 或 volatile 修饰符也能影响引用参数的初始化的等级。如同转换序列的情形一样如果两个引用初始化是相同的，只不过其中一个增加了一个额外的 const 或 volatile 修饰符，那么对于函数重载解析来说，没有额外修饰符的引用初始化是比较好的引用初始化，例如：

```
#include <vector>
void manip( vector<int> & );
void manip( const vector<int> & );
vector<int> f();
extern vector<int> vec;

int main() {
     manip( vec );  // 选择 manip( vector<int> & ) is selected
     manip( f() );  // 选择 manip( const vector<int> & ) is selected
     return 0;
}
```

在第一个调用中，两个调用的引用初始化都是精确匹配。但是该调用不是二义的．因为两个引用初始化都相同，除了第二个加上了 const 修饰符，所以没有额外限定修饰的初始化被认为是较好的初始化。因此，可行函数 manip(vector<int>&)是第一个调用的最佳可行函数。

在第二个调用中，该调用只有一个可行函数：manip(const vector<int>&)。因为实参是存放函数 f()返回值的临时单元，所以该实参是一个右值，它不能被用来初始化 manip(vector<int>&)的非 const 引用参数。因此，对于第二个调用的最佳可行函数，编译器只考虑一个可行函数：manip(const vector<int>&)。

当然，函数调用可以有一个以上的实参，选择最佳可行函数时必须考虑转换全部实参所需的转换序列的等级。我们来看一个例子：

```
extern int ff( char*, int );
extern int ff( int, int );

int main() {
     ff( 0, 'a' );// ff( int, int )
     return 0;
}
```

由于下述原因，有两个 int 型的参数的函数 ff()被选为最佳可行函数。

1．它的第一个实参较好。0 是 int 型的参数的精确匹配，而对于第一个 ff(char*,int)函数来说，它需要一个指针标准转换序列来匹配 char*型的参数。

2．它们的第二个实参一样好。实参‘a’的类型是 char，两个函数的第二个参数的匹配都要求一个提升等级的转换序列。

这里还有另外一个例子：

```
int compute( const int&, short );
int compute( int&, double );
extern int iobj;

int main() {
```

```
        compute( iobj, 'c' );   // compute( int&, double )
        return 0;
    }
```

这两个函数 compute(const int&,short)和 compute(int&,double)都是可行函数。由于下述原因第二个函数被选为最佳可行函数。

1．它的第一个实参较好。第一个可行函数的引用初始化比较差，因为它加入了一个 const 限定修饰符，而第二个可行函数的初始化没有加入 const 限定修饰符。

2．它们的第二个实参一样好。实参‘c’的类型是 char，要匹配两个函数的第二个参数都要求一个标准转换等级的转换序列。

9.4.4 缺省实参

缺省实参可以使多个函数进入到可行函数集合中。可行函数是指可以用调用中指定的实参进行调用的函数。可行函数可以有比函数调用实参表中的实参个数更多的参数，只要每个多出来的参数都有相应的缺省实参即可：

```
    extern void ff( int );
    extern void ff( long, int = 0 );
    int main() {
        ff( 2L );     // 匹配 ff( long, 0 );
        ff( 0, 0 );   // 匹配 ff( long, int );
        ff( 0 );      // 匹配 ff( int );
        ff( 3.14 );   // 错误：二义
    }
```

对于第一个和第三个调用，即使该实参表中只有一个实参，第二个函数 ff()仍然是两个调用的可行函数，原因如下：

1．函数的第二个参数有相应的缺省实参。

2．函数的第一个参数是 long 型，与第一个调用的实参类型精确匹配。通过标准转换等级的转换序列，与第三个调用的实参类型也匹配。

最后一个调用是二义的，这是因为通过在第一个实参上应用标准转换，两个实例都可以匹配。这里不能选择 ff(int)作为更好的函数，因为它只有一个实参。

练习 9.9

解释在 main()中对 compute()的调用的函数重载解析过程中发生的事情。哪些函数是候选函数？哪些函数是可行函数？应用在实参上使其与每个可行函数的参数匹配的类型转换序列是什么？哪个函数（如果存在的话）是最佳可行函数？

```
    namespace primerLib {
        void compute( );
        void compute( const void * );
    }

    using primerLib::compute;
    void compute( int );
    void compute( double, double = 3.4 );
```

```
void compute( char*, char* = 0 );
int main() {
     compute( 0 );
     return 0;
}
```

如果将 using 声明放在 main()中，但是在 compute()调用之前，会怎么样？回答与上面同样的问题。