第14章　函数重载、拷贝构造函数和默认变元

本章讨论函数重载、拷贝构造函数和默认变元。函数重载是定义C++编程语言的一个方面，它不仅提供了对编译时多态性的支持，也增加了灵活性和方便性。某些最常见的重载函数是构造函数。或许重载构造函数中最重要的形式是拷贝构造函数。与函数重载密切相关的是默认变元。有时，默认变元可以替换函数重载。

14.1　函数重载

函数重载是两个或更多的函数使用同一名字的过程。重载的关键是函数的每个重定义必须使用不同类型的参数或不同数目的参数。只需通过这些简单的区别，编译器就知道在任何给定的情况下应该调用哪个函数，例如，下面的程序通过使用不同类型的参数重载函数myfunc()。

```
#include <iostream>
using namespace std;

int myfunc(int i); // these differ in types of parameters
double myfunc(double i);

int main( )
{
  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(5.4); // calls myfunc(double i)

  return 0;
}

double myfunc(double i)
{
  return i;
}

int myfunc(int i)
{
  return i;
}
```

下面的程序使用不同数目的参数重载myfunc()：

```
#include <iostream>
using namespace std;

int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);

int main( )
{
```

```
  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(4, 5); // calls myfunc(int i, int j)

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

如前所述，函数重载的关键是，函数必须具有不同类型或不同数目的参数。仅仅返回值类型不同的两个函数不能够重载。例如，下面是重载myfunc( )的无效尝试：

```
int myfunc(int i);      // Error: differing return types are
float myfunc(int i);    // insufficient when overloading.
```

有时两个函数声明的形式好像有所不同，但实际上是相同的。例如，考虑下面的声明：

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

记住，对于编译器，*p与p[ ]是相同的。因此，虽然两个原型在参数类型上不同，但事实上它们没有区别。

14.2 重载构造函数

可以重载构造函数，事实上，重载的构造函数是非常常见的。为什么想要重载构造函数，主要有三个理由：为了获得灵活性，为了创建初始化和未初始化对象，以及为了定义拷贝构造函数。在本节中，先讨论前两项，下一节讨论拷贝构造函数。

14.2.1 重载构造函数以获得灵活性

很多时候，我们都会想创建一个类，对这个类来说，有两种或更多的方式来构造一个对象。这时，你会想要为每种方式提供一个重载的构造函数。这是一个自我实行的原则，因为如果尝试创建一个对象，而该对象没有匹配的构造函数，则就会出现编译时错误。

通过为每种方式（你的类的用户可能想依据每种方式构建一个对象）提供一个构造函数，可以增加类的灵活性。在给定环境下，用户可以自由选择构建对象的最好方式。考虑下面的程序，它创建了一个称为date的类，以容纳公历日期。注意，构造函数是用两种方式重载的：

```
#include <iostream>
#include <cstdio>
using namespace std;

class date {
  int day, month, year;
public:
```

```
  date(char *d);
  date(int m, int d, int y);
  void show_date( );
};

// Initialize using string.
date::date(char *d)
{
  sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
}

// Initialize using integers.
date::date(int m, int d, int y)
{
  day = d;
  month = m;
  year = y;
}

void date::show_date( )
{
  cout << month << "/" << day;
  cout << "/" << year << "\n";
}

int main( )
{
  date ob1(12, 4, 2003), ob2("10/22/2003");

  ob1.show_date( );
  ob2.show_date( );

  return 0;
}
```

在这个程序中，可以初始化类型为date的对象，其方法是通过用三个整数代表月、日和年来指定日期，或者使用一个字符串来指定日期，该字符串包含按下面的一般形式出现的日期：

mm/dd/yyyy

因为这两种都是表示一个日期的常见的方式，当构造一个对象时，date允许这两种形式是有意义的。

正如date类所演示的，重载构造函数的最通常的理由是，允许对每种特殊的情况使用最恰当、最自然的方式创建对象。例如，在下面的main( )中，提示用户把日期输入到数组s中。以后可以直接使用这个字符串来创建d，不需要转换成任何其他形式。但是，如果date( )不被重载以接受字符串格式，将不得不用手工方式把它转换成三个整数。

```
int main( )
{
  char s[80];
  cout << "Enter new date: ";
  cin >> s;

  date d(s);
```

```
  d.show_date( );

  return 0;
}
```

在另一种情况下，使用三个整数初始化类型为date的对象更为方便。例如，如果日期是通过某种计算方法产生的，那么date(int,int,int)就是创建类型为date的对象的最自然和最恰当的构造函数。这里的关键是通过重载date的构造函数，使之用起来更灵活、更简单。这种增加的灵活性和易用性，对创建供其他程序员使用的类库尤为重要。

14.2.2 允许创建初始化和未初始化对象

重载构造函数的另一个常见的理由是允许创建初始化和未初始化的对象（或者，更准确地讲，默认的初始化对象）。如果想要能够创建某些类的动态对象数组，这是特别重要的，因为不能初始化一个动态分配的数组。要允许未初始化的对象数组和初始化的对象，必须包括一个支持初始化的构造函数和一个不支持初始化的构造函数。

例如，下面的程序声明了两个powers类型的数组，一个被初始化了，而另一个没有被初始化。它也动态地分配一个数组。

```
#include <iostream>
#include <new>
using namespace std;

class powers {
  int x;
public:
  // overload constructor two ways
  powers( ) { x = 0; } // no initializer
  powers(int n) { x = n; } // initializer

  int getx( ) { return x; }
  void setx(int i) { x = i; }
};

int main( )
{
  powers ofTwo[ ] = {1, 2, 4, 8, 16}; // initialized
  powers ofThree[5]; // uninitialized
  powers *p;
  int i;

  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << ofTwo[i].getx( ) << " ";
}
  cout << "\n\n";

  // set powers of three
  ofThree[0].setx(1);
  ofThree[1].setx(3);
  ofThree[2].setx(9);
  ofThree[3].setx(27);
```

```
  ofThree[ 4] .setx(81);

  // show powers of three
  cout << "Powers of three: ";
  for(i=0; i<5; i++) {
    cout << ofThree[ i] .getx( ) << " ";
  }
  cout << "\n\n";

  // dynamically allocate an array
  try {
    p = new powers[ 5] ; // no initialization
  } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      return 1;
  }

  // initialize dynamic array with powers of two
  for(i=0; i<5; i++) {
      p[ i] .setx(ofTwo[ i] .getx( ));
  }

  // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
    cout << p[ i] .getx( ) << " ";
  }
  cout << "\n\n";

  delete [] p;
  return 0;
}
```

在这个例子中，两个构造函数都是必要的。默认的构造函数用子构建未初始化的ofThree数组和动态分配的数组。参数化的构造函数被调用来创建ofTwo数组的对象。

14.3　拷贝构造函数

重载构造函数中一个较重要的形式是拷贝构造函数。定义一个拷贝构造函数可以帮助防止在用一个对象初始化另一个对象时可能出现的问题。

让我们通过重述设计拷贝构造函数要解决的问题开始。默认时，当用一个对象初始化另一个对象时，C++按位进行拷贝，即在目标对象里创建了初始化对象的一个完全相同的拷贝。尽管对于很多情况这样做已经足够了，但在有些情况下用户还是希望不用按位拷贝。最常见的情形之一是当对象分配内存时（当其被创建时）应避免按位拷贝。例如，假设有一个类MyClass，在其创建时它为每个对象分配内存，同时假定那个类的对象A已经存在。这意味着已经为A分配了内存。又假设用A初始化B，如下所示：

```
MyClass B = A;
```

如果执行的是按位拷贝，那么B即为A的完全拷贝。这意味着B将使用A正在使用的内存而不必另外为B分配内存。显然，这不是我们想要的结果。例如，如果MyClass包含一个释放内存

的析构函数，则当A和B被销毁时，同样的内存将会被释放两次！

同样的问题可能以另外两种形式出现：第一种是当对象作为一个变元传递给函数时对象的拷贝生成；第二种是当一个临时对象作为函数返回值被创建时。记住：临时对象是被自动创建用于容纳函数返回值的，它们也可以在其他环境下创建。

为了解决刚才所说的问题，C++允许创建拷贝构造函数，供编译器在用一个对象初始化另一个对象时使用。拷贝构造函数一旦存在，按位拷贝就不起作用了。拷贝构造函数的一般形式为：

```
classname (const classname &o) {
  // body of constructor
}
```

其中，o是初始化时左边对象的引用。只要定义了默认变元，就允许拷贝构造函数有另外的参数。但不管在什么情况下，第一个参数必须是对完成初始化工作的对象的引用。

重要的是要理解，C++定义了两个独特的情况，用于将一个对象的值传给另一个对象。第一种是赋值。第二种是初始化，初始化方法有三种：

- 当一个对象初始化另一个对象时，例如在声明中。
- 当所创建的一个对象的拷贝传给一个函数时。
- 当生成临时对象时（最常见的是作为返回值）。

拷贝构造函数只应用于初始化。例如，假定有一个类mycalss，且y是mycalss的一个对象，则下面的每条语句都做初始化操作：

```
myclass x = y;      // y explicitly initializing x
func(y);            // y passed as a parameter
y = func( );        // y receiving a temporary, return object
```

下面是一个需要显式的拷贝构造函数的例子。程序创建了一个能防止溢出数组边界的、有限的“可靠”整型数组类型（第15章展示了一个使用重载运算符创建可靠数组的更好的方法）。每个数组的存储空间用new分配，在每个数组对象的边界内维护指向内存的指针。

```
/* This program creates a "safe" array class. Since space
   for the array is allocated using new, a copy constructor
   is provided to allocate memory when one array object is
   used to initialize another.
*/
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class array {
   int *p;
   int size;
public:
  array(int sz) {
    try {
      p = new int[ sz] ;
```

```
    } catch (bad_alloc xa) {
      cout << "Allocation Failure\n";
      exit(EXIT_FAILURE);
    }
    size = sz;
  }
  ~array( ) { delete [] p; }

  // copy constructor
  array(const array &a);

  void put(int i, int j) {
    if(i>=0 && i<size) p[i] = j;
  }
  int get(int i) {
    return p[i];
  }
};

// Copy Constructor
array::array(const array &a) {
  int i;

  try {
    p = new int[a.size];
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    exit(EXIT_FAILURE);
  }
  for(i=0; i<a.size; i++) p[i] = a.p[i];
}

int main( )
{
  array num(10);
  int i;

  for(i=0; i<10; i++) num.put(i, i);
  for(i=9; i>=0; i--) cout << num.get(i);
  cout << "\n";

  // create another array and initialize with num
  array x(num); // invokes copy constructor
  for(i=0; i<10; i++) cout << x.get(i);

  return 0;
}
```

让我们仔细看看当在下面的语句中使用num初始化x时发生了什么。

```
array x(num); // invokes copy constructor
```

调用拷贝构造函数，为新数组分配内存并存储在x.p中，同时将num的内容拷贝到x的数组中。这样，x和num就有了包含相同值的数组，但是每个数组又是独立且不同的（即num.p和x.p不指向同一块内存）。如果不创建拷贝构造函数，则默认的按位初始化就使x和num分享

同一块内存（即num.p和x.p真正指向同一地址）。

记住，调用拷贝构造函数只是为了初始化。例如，下面的程序不调用在上面程序中定义的拷贝构造函数。

```
array a(10);
// ...
array b(10);

b = a; // does not call copy constructor
```

在此情况下，b=a执行赋值运算。如果不重载=（因为它在别的地方），就会导致按位拷贝。所以，在某些情况下，仍需要重载运算符=并创建拷贝构造函数，以避免产生问题（参见第15章）。

14.4 查找重载函数的地址

正如在第5章中解释的，可以获得函数的地址。这样做的一个理由是：把函数的地址赋给一个指针，然后用指针来调用函数。如果函数没有重载，这个过程很简单直接。然而，对于重载函数来说，处理起来就稍复杂一些。为理解其原因，先考虑下面的语句，它把函数myfunc()的地址赋给指针p：

```
p = myfunc;
```

如果myfunc()没有重载，那么有且仅有一个称为myfunc()的函数，编译器不难将其地址赋给P。但如果重载了myfunc()，那么编译器怎么知道把哪个函数地址赋给p呢？答案取决于p是如何被声明的。例如，考虑下面的程序：

```
#include <iostream>
using namespace std;

int myfunc(int a);
int myfunc(int a, int b);

int main( )
{
  int (*fp)(int a); // pointer to int f(int)

  fp = myfunc; // points to myfunc(int)

  cout << fp(5);

  return 0;
}

int myfunc(int a)
{
  return a;
}

int myfunc(int a, int b)
{
  return a*b;
}
```

其中，有两个版本的 myfunc( )。虽然二者都返回 int，但是一个带有一个整型变元，另一个带有两个整型变元。在程序中，fp 被声明为一个指向函数的指针，该函数返回一个整数并带一个整型变元。当 fp 被赋予 myfunc( )的地址时，C++ 使用这个信息选择 myfunc( )的 myfunc(int a) 形式。如果 fp 是像下面这样声明的：

```
int (*fp)(int a, int b);
```

那么，fp 将被赋予 myfunc( )的 myfunc(int a,int b)的地址。

通常，当把一个重载函数的地址赋给一个函数指针时，正是指针声明决定了获得哪个函数的地址。此外，函数指针声明必须匹配一个且仅仅一个重载函数的声明。

14.5　重载的过去与现在

当 C++ 创立时，要用关键字 overload 来创建重载函数。现在它已经过时且不再使用。事实上，在标准 C++ 中，它甚至不是保留字。然而，因为我们可能遇到老版本的程序，也为了了解历史，知道 overload 是如何使用的仍不失为一个好主意。下面是重载的一般形式：

overload *func-name*;

其中，func-name 是要重载的函数名。这个语句必须置于重载声明之前。例如，下面告诉老式的编译器用户将重载一个称为 test( )的函数：

```
overload test;
```

14.6　默认的函数变元

C++ 在调用函数时，如果没有指定和参数相对应的变元，则允许函数赋给参数一个默认值。默认值是按语法上和变量初始化类似的方法指定的。例如，下面声明了一个函数 myfunc( )，它带有一个默认值为 0.0 的 double 变元。

```
void myfunc(double d = 0.0)
{
    // ...
}
```

下面的例子说明调用函数 myfunc( )的两种方法是：

```
myfunc(198.234);   // pass an explicit value
myfunc( );         // let function use default
```

第一个调用把值 198.234 传给 d，第二个调用自动把默认值 0 赋给 d。

默认变元包含在 C++ 里的原因是，默认变元给程序员提供了管理更大复杂性的另一种方法。为了处理更广泛的情况，往往是一个函数包含比通常情况下所需的更多的参数。所以，当默认值适用时，只需指定对特定情况而不是最一般情况有意义的变元。例如，许多 C++ I/O 函数就是因为这个原因而使用默认变元的。

下面程序中的函数 clrscr( )简要地说明了一个默认函数变元的用途。函数 clrscr( )通过输出一系列换行符来清除屏幕（它不是最好的方法，但对本例已足够了）。由于最常用的显示方式是 25 行文本，所以默认值为 25。某些终端能够显示多于 25 行或少于 25 行文本（通常按使用的

显示方式而定），这时可以通过显式地指定变元值来取代默认值。

```
#include <iostream>
using namespace std;

void clrscr(int size=25);

int main( )
{
  register int i;

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get( );
  clrscr( ); // clears 25 lines

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get( );
  clrscr(10); // clears 10 lines

  return 0;
}

void clrscr(int size)
{
  for(; size; size--) cout << endl;
}
```

这个程序说明，在调用clrscr( )时，如果默认值合适，就不必指定变元了。然而，在需要时也可以赋给size一个不同的值来取代默认值。

默认变元也可以用做标志，以通知函数重用以前的变元。为了说明这种默认变元的用法，这里编写了一个简单的函数iputs( )，它自动地按指定的数缩进一个串。下面是不使用默认变元的函数版本：

```
void iputs(char *str, int indent)
{
  if(indent < 0) indent = 0;

  for( ; indent; indent--) cout << " ";

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

调用这种形式的iputs( )要带两个变元：输出的串作为第一个变元，缩进的数量作为第二个变元。尽管这样写没有任何错误，但我们还可以改进它，即给参数indent提供一个默认值，以通知iputs( )缩进到和上一行对齐。常见的是按每行一样的量缩进显示一个文本块。在这种情况下，不要一次又一次地赋给变元indent相同的值，而是赋给indent一个默认值，以通知iputs( )按以前指定的数量缩进。下面的程序演示了这种方法：

```
#include <iostream>
using namespace std;

/* Default indent to -1. This value tells the function
   to reuse the previous value. */
void iputs(char *str, int indent = -1);
```

```
int main( )
{
  iputs("Hello there", 10);
  iputs("This will be indented 10 spaces by default");
  iputs("This will be indented 5 spaces", 5);
  iputs("This is not indented", 0);

  return 0;
}

void iputs(char *str, int indent)
{
  static i = 0; // holds previous indent value

  if(indent >= 0)
    i = indent;
  else // reuse old indent value
    indent = i;

  for( ; indent; indent--) cout << " ";

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

这个程序显示如下的输出：

```
          Hello there
          This will be indented 10 spaces by default
     This will be indented 5 spaces
This is not indented
```

在创建带默认变元的函数时，重要的是要记住只能指定一次默认值，而且必须是在文件中第一次声明函数时。在前面的程序中，默认变元在iputs( )的原型中指定。如果试图在iputs( )的定义中指定新的默认值(甚至是同样的值)，编译器就会提示一个错误并停止编译。尽管在同一程序中不能重定义默认值，但可以给重载函数的不同形式指定不同的默认值。

带默认值的所有参数必须出现在不带默认值的参数的右边。例如，下面这样定义iputs( )是错误的：

```
// wrong!
void iputs(int indent = -1, char *str);
```

一旦开始定义带默认值的参数，就不能指定不带默认值的参数了。所以，下面的声明是错误的，不能编译：

```
int myfunc(float f, char *str, int i=10, int j);
```

由于i被赋予了默认值，所以j也必须被赋予默认值。

也可以在对象的构造函数中使用默认参数。例如，下面的类cube管理一个立方体的整数维。如果没提供任何别的变元，它的构造函数使所有维默认为0：

```
#include <iostream>
using namespace std;

class cube {
```

```
  int x, y, z;
public:
  cube(int i=0, int j=0, int k=0) {
    x=i;
    y=j;
    z=k;
  }

  int volume( ) {
   return x*y*z;
  }
};

int main( )
{
  cube a(2,3,4), b;

  cout << a.volume( ) << endl;
  cout << b.volume( );

  return 0;
}
```

在适当的时候，把默认变元包含在类的构造函数里有两个好处。第一，使用户不必提供不带参数的构造函数。例如，假设不给cube( )的参数赋默认值，那么就需要用如下所示的另一个构造函数处理b的声明（没有指定变元）。

```
cube( ) {x=0; y=0; z=0}
```

第二，使用默认的常见初值比在每次声明对象时指定这些值要方便得多。

14.6.1 默认变元与重载

在某些情况下，默认变元可以用做函数重载的一种缩写形式。刚才所示的cube类的构造函数就是一个例子。让我们看另一个例子。假设要建立标准strcat( )函数的两种定制版本。第一种版本正如strcat( )一样，将两个串首尾相接；第二种版本带有第三个变元，该变元说明要连接的字符的个数。也就是说，第二种版本仅将一个串中指定数目的字符接入另一个串的尾部。因此，假设调用定制函数mystrcat( )，其原型如下：

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

第一种版本将s2中len个字符复制到s1的尾部；第二种版本将把s2指向的整个串复制到s1所指的串的尾部，操作方法如同strcat( )。

虽然上述两种版本没有错误，但有更容易的方法。使用默认变元，可以创建一种mystrcat( )版本来完成两个函数的功能。程序如下：

```
// A customized version of strcat( ).
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);
```

```
int main( )
{
  char str1[ 80] = "This is a test";
  char str2[ 80] = "0123456789";
  mystrcat(str1, str2, 5); // concatenate 5 chars
  cout << str1 << '\n';

  strcpy(str1, "This is a test"); // reset str1

  mystrcat(str1, str2); // concatenate entire string
  cout << str1 << '\n';

  return 0;
}

// A custom version of strcat( ).
void mystrcat(char *s1, char *s2, int len)
{
  // find end of s1
  while(*s1) s1++;

  if(len == -1) len = strlen(s2);

  while(*s2 && len) {
    *s1 = *s2; // copy chars
    s1++;
    s2++;
    len--;
  }

  *s1 = '\0'; // null terminate s1
}
```

其中，mystrcat( )将 s2 中长度为 len 的字符连入 s1 的尾部。但是。如果 len 为 -1，正如允许有默认值一样，mystrcat( )将 s2 的整个串连入 s1（因此，当 len 为 -1 时，函数操作同标准的 strcat( )函数一样）。通过使用 len 的默认变元，可以将两个操作组合进一个函数中，这样，默认变元就提供了函数重载的缩写形式。

14.6.2　正确使用默认变元

若使用正确的话，默认变元是非常强大的工具，但也有造成滥用的可能。使用默认变元的关键是使函数有效、方便、灵活地完成其任务，所以，所有的默认变元都应该反映函数最常使用的情况。当一个值在与参数相联时，如果没有意义，就没有理由声明一个默认变元。事实上，如果没有足够的理由就声明默认变元，常常会使程序代码结构混乱，让人读起来相当费解或使人误入歧途。

在使用默认变元时，应遵守的另一个重要原则是：不应有任何默认变元会造成伤害和毁坏性的行为，也就是说，计划外偶尔使用默认变元不应该导致灾难。

14.7　函数重载和二义性

有可能出现这样一种情况，即编译器不能在两个或两个以上重载函数中做出正确选择。当这发生时，我们称之为具有二义性。具有二义性的语句是错误的，具有二义性的程序无法编译。

造成二义性的主要原因是C++的自动类型转换。我们知道，C++自动试图把函数调用时所用的变元转换成函数所期望的变元类型。例如，考虑下面的程序段：

```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

正如注释指出的，由于C++自动将字符c转换为等价的double值，所以上面的语句没有错误。在C++中，不允许进行这种转换的类型很少。虽然自动类型转换很方便，它们也是引起二义性的主要原因。例如，考虑下面的程序：

```
#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main( )
{
  cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
  cout << myfunc(10); // ambiguous

  return 0;
}

float myfunc(float i)
{
  return i;
}

double myfunc(double i)
{
  return -i;
}
```

其中，myfunc( )被重载，因此它能带有浮点或双精度类型的变元。在无二义性的行里，要调用myfunc(double)，这是因为，在C++中，除非明确地指定为float，所有浮点常数都自动为double型，所以该调用没有二义性。但是，当使用整数10调用myfunc( )时，就出现二义性了，这是因为编译器无法知道它应该转换成float还是double，从而就会提示错误信息并停止编译程序。

前面的例子说明，不是关于double和float的myfunc( )的重载引起了二义性，而是用不定的变元对myfunc( )的特定调用引起了二义性。换句话说，错误不是通过myfunc( )的重载引起的，而是通过特殊的调用引起的。

下面是由C++的自动类型转换引起的二义性的又一个例子：

```
#include <iostream>
using namespace std;

char myfunc(unsigned char ch);
char myfunc(char ch);

int main( )
{
```

```
  cout << myfunc('c'); // this calls myfunc(char)
  cout << myfunc(88) << " "; // ambiguous

  return 0;
}

char myfunc(unsigned char ch)
{
  return ch-1;
}

char myfunc(char ch)
{
  return ch+1;
}
```

在C++中，unsigned char和char并非天生具有二义性，但当用整型数88调用myfunc( )时，编译器无法知道要调用哪一个函数，即88应该转换成char还是unsigned char。

另一个导致二义性的原因是在重载函数中使用默认变元，看看下面的程序，就会明白：

```
#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main( )
{
  cout << myfunc(4, 5) << " "; // unambiguous
  cout << myfunc(10); // ambiguous

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

其中，第一次调用myfunc( )时，指定了两个变元，所以不会引起二义性，并且调用的是myfunc(int i,int j)。然而，当第二次调用myfunc( )时，因为编译器不知道是调用带一个变元的myfunc( )，还是默认情况的带两个变元的myfunc( )，所以引起了二义性。

有些类型的重载函数即使在开始时好像没有二义性，但实际上天生就有二义性。例如，考虑下面的程序：

```
// This program contains an error.
#include <iostream>
using namespace std;

void f(int x);
```

```
void f(int &x); // error

int main( )
{
  int a=10;

  f(a); // error, which f( )?

  return 0;
}

void f(int x)
{
  cout << "In f(int)\n";
}

void f(int &x)
{
  cout << "In f(int &)\n";
}
```

正如程序注释所述，当惟一的区别是一个函数带引用参数而另一个函数带通常的按值调用参数时，这两个函数不能重载。这时，编译器无法知道到底要调用哪一个函数。记住，当一个变元作为引用参数或值参数被接受时，其被指定的方式没有语义上的区别。