# 8.4 函数重载

函数多态是C++在C语言的基础上新增的功能。默认参数让您能够使用不同数目的参数调用同一个函数，而函数多态（函数重载）让您能够使用多个同名的函数。术语“多态”指的是有多种形式，因此函数多态允许函数可以有多种形式。类似地，术语“函数重载”指的是可以有多个同名的函数，因此对名称进行了重载。这两个术语指的是同一回事，但我们通常使用函数重载。可以通过函数重载来设计一系列函数——它们完成相同的工作，但使用不同的参数列表。

重载函数就像是有多种含义的动词。例如，Piggy小姐可以在棒球场为家乡球队助威（root），也可以在地里种植（root）菌类作物。根据上下文可以知道在每一种情况下，root的含义是什么。同样，C++使用上下文来确定要使用的的重载函数版本。

函数重载的关键是函数的参数列表——也称为函数特征标（function signature）。如果两个函数的参数数目和类型相同，同时参数的排列顺序也相同，则它们的特征标相同，而变量名是无关紧要的。C++允许定义名称相同的函数，条件是它们的特征标不同。如果参数数目和/或参数类型不同，则特征标也不同。例如，可以定义一组原型如下的print()函数：

void print(const char \* str, int width); // #1

void print(double d, int width); // #2

void print(long l, int width); // #3

void print(int i, int width); // #4

void print(const char \*str); // #5

使用print()函数时，编译器将根据所采取的用法使用有相应特征标的原型：

print("Pancakes",15); // 使用 #1

print(“Syrup”); // 使用 #5

print(1999.0,10); // 使用 #2

print(1999,12); // 使用 #4

print(1999L,15); // 使用 #3

例如，print(“Pancakes”，15)使用一个字符串和一个整数作为参数，这与#1原型匹配。

使用被重载的函数时，需要在函数调用中使用正确的参数类型。例如，对于下面的语句：

unsigned int year = 3210;

print (year, 6); // 不明确的调用

print()调用与哪个原型匹配呢？它不与任何原型匹配！没有匹配的原型并不会自动停止使用其中的某个函数，因为C++将尝试使用标准类型转换强制进行匹配。如果#2原型是print()唯一的原型，则函数调用print(year,6)将把year转换为double类型。但在上面的代码中，有3个将数字作为第一个参数的原型，因此有3种转换year的方式。在这种情况下，C++将拒绝这种函数调用，并将其视为错误。

一些看起来彼此不同的特征标是不能共存的。例如，请看下面的两个原型：

double cube(double x);

double cube(double & x);

您可能认为可以在此处使用函数重载，因为它们的特征标看起来不同。然而，请从编译器的角度来考虑这个问题。假设有下面这样的代码：

cout << cube(x);

参数x与double x原型和double &x原型都匹配，因此编译器无法确定究竟应使用哪个原型。为避免这种混乱，编译器在检查函数特征标时，将把类型引用和类型本身视为同一个特征标。

匹配函数时，并不区分const和非const变量。请看下面的原型：

void dribble (char \* bits); // overloaded

void dribble (const char \* cbits); // overloaded

void dabble(char \* bits); // not overloaded

void drivel(const char \* bits); // not overloaded

下面列出了各种函数调用对应的原型：

const char p1[20] ="How's the weather?”;

char p2[20] = "How's business?“;

dribble(pl); // dribble(const char \*);

dribble(p2); // dribble(char \*);

dabble(pl); // no match

dabble(p2); // dabble(char \*);

drivel(pl); // drivel(const char \*);

drivel(p2); // drivel(const char \*);

dribble()函数有两个原型，一个用于const指针，另一个用于常规指针，编译器将根据实参是否为const来决定使用哪个原型。dribble()函数只与带非const参数的调用匹配，而drivel()函数可以与带const或非const参数的调用匹配。drivel()和dabble()之所以在行为上有这种差别，主要是由于将非const值赋给const变量是合法的，但反之则是非法的。

请记住，是特征标，而不是函数类型使得可以对函数进行重载。例如，下面的两个声明是互斥的：

long gronk(int n,float m); // 相同的特征标,

double gronk(int n,float m); // 因此不允许重载

因此，C++不允许以这种方式重载gronk()。返回类型可以不同，但特征标也必须不同：

long gronk (int n,float m); // 不同的特征标,

double gronke(float n, float m); // 可以重载

在本章稍后讨论过模板后，将进一步讨论函数匹配的问题。

**重载引用参数**

类设计和STL经常使用引用参数，因此知道不同引用类型的重载很有用。请看下面三个原型：

void sink( double & r1);

void sank(const double & r2);

void sunk(double && r3);

左值引用参数r1与可修改的左值参数（如double变量）匹配；const左值引用参数r2与可修改的左值参数、const左值参数和右值参数（如两个double值的和）匹配；最后，左值引用参数r3与左值匹配。注意到与r1或r3匹配的参数都与r2匹配。这就带来了一个问题：如果重载使用这三种参数的函数，结果将如何？答素是将调用最匹配的版本：

void staff (double & rs); // 匹配可修改的左值

void staff (const double & rcs) // 匹配右值，常量左值

void stove (double & r1); // 匹配可修改的左值

void stove (const double & r2); // 匹配常量左值

void stove (double && r3); // 匹配右值

这让您能够根据参数是左值、const还是右值来定制函数的行为：

double x = 55.5;

const double y = 32.0;

stove(x); // 调用stove(double &)

stove(y); // 调用stove(const double &)

stove(x+y); // 调用stove(double &&)

如果没有定义函数stove(double &&)，stove(x+y)将调用函数stove(const double &)。

## 8.4.1 重载示例

本章前面创建了一个left()函数，它返回一个指针，指向字符串的前n个字符。下面添加另一个left()函数，它返回整数的前n位。例如，可以使用该函数来查看被存储为整数的、美国邮政编码的前3位——如果要根据城区分拣邮件，则这种操作很有用。

该函数的整数版本编写起来比字符串版本更困难些，因为并不是整数的每一位被存储在相应的数组元素中。一种办法是，先计算数字包含多少位。将数字除以10便可以去掉一位，因此可以使用除法来计算数位。更准确地说，可以用下面的循环完成这种工作：

unsigned digits = 1；

while(n /= 10)

 digits++;

上述循环计算每次删除n中的一位时，需要多少次才能删除所有的位。前面讲过，n/=10是n = n/10的缩写。例如，如果n为8，则该测试条件将8/10的值（0，由于这是整数除法）赋给n。这将结束循环，digits的值仍然为1。但如果n为238，第一轮循环测试将n设置为238/10，即23。这个值不为零，因此循环将digits增加到2。下一轮循环将n设置为23/10，即2。这个值还是不为零，因此digits将增加到3。下一轮循环将n设置为2/10，即0，从而结束循环，而digits被设置为正确的值——3。

现在假设知道数字共有5位，并要返回前3位，则将这个数除以10后再除以10，便可以得到所需的值。每除以10次就删除数字的最后一位。要知道需要删除多少位，只需将总位数减去要获得的位数即可。例如，要获得9位数的前4位，需要删除后面的5位。可以这样编写代码：

ct = digits - ct;

while(ct--)

 num /= 10;

return num;

程序清单8.10将上述代码放到了一个新的left()函数中。该函数还包含一些用于处理特殊情况的代码，如用户要求显示0位或要求显示的位数多于总位数。由于新left()的特征标不同于旧的left()，因此可以在同一个程序中使用这两个函数。

**程序清单8.10 leftover.cpp**

// leftover.cpp -- 重载left()方法

#include <iostream>

unsigned long left(unsigned long num, unsigned ct);

char \* left(const char \* str, int n = 1);

int main()

{

 using namespace std;

 char \* trip = "Hawaii!!"; // 测试值

 unsigned long n = 12345678; // 测试值

 int i;

 char \* temp;

 for (i = 1; i < 10; i++)

 {

 cout << left(n, i) << endl;

 temp = left(trip,i);

 cout << temp << endl;

 delete [] temp; // point to temporary storage

 }

 cin.get();

 return 0;

}

// 这个方法返回数字num的前ct个数字

unsigned long left(unsigned long num, unsigned ct)

{

 unsigned digits = 1;

 unsigned long n = num;

 if (ct == 0 || num == 0)

 return 0; // return 0 if no digits

 while (n /= 10)

 digits++;

 if (digits > ct)

 {

 ct = digits - ct;

 while (ct--)

 num /= 10;

 return num; // 返回前ct位

 }

 else // 如果ct >= number的位数

 return num; // 则返回整个数字

}

// 这个函数返回一个指向新字符串的指针，

// 这个字符串由字符串str的前n个字符组成

char \* left(const char \* str, int n)

{

 if(n < 0)

 n = 0;

 char \* p = new char[n+1];

 int i;

 for (i = 0; i < n && str[i]; i++)

 p[i] = str[i]; // 复制字符

 while (i <= n)

 p[i++] = '\0'; // 将字符串的剩余部分设置为'\0'

 return p;

}

下面是该程序的输出：

1

H

12

Ha

123

Haw

1234

Hawa

12345

Hawai

123456

Hawaii

1234567

Hawaii!

12345678

Hawaii!!

12345678

Hawaii!!

## 8.4.2 何时使用函数重载

虽然函数重载很吸引人，但也不要滥用。仅当函数基本上执行相同的任务，但使用不同形式的数据时，才应采用函数重载。另外，您可能还想知道，是否可以通过使用默认参数来实现同样的目的。例如，可以用两个重载函数来代替面向字符串的left()函数：

char \* left (const char \* str,unsigned n); // 两个参数

char \* left (const char \* str); // 一个参数

使用一个带默认参数的函数要简单些。只需编写一个函数（而不是两个函数），程序也只需为一个函数（而不是两个）请求内存；需要修改函数时，只需修改一个。然而，如果需要使用不同类型的参数，则默认参数便不管用了，在这种情况下，应该使用函数重载。

**什么是名称修饰**

C++如何跟踪每一个重载函数呢？它给这些函数指定了秘密身份。使用C++开发工具中的编辑器编写和编译程序时，C++编译器将执行一些神奇的操作——名称修饰（name decoration）或名称矫正（name mangling），它根据函数原型中指定的形参类型对每个函数名进行加密。请看下述未经修饰的函数原型：

long MyFunctionFoo(int,float);

这种格式对于人类来说很适合：我们知道函数接受两个参数（一个为int类型，另一个为float类型），并返回一个long值。而编译器将名称转换为不太好看的内部表示，来描述该接口，如下所示：

?MyFunctionFoo@@YAXH

对原始名称进行的表面看来无意义的修饰（或矫正，因人而异）将对参数数目和类型进行编码。添加的一组符号随函数特征标而异，而修饰时使用的约定随编译器而异。