# 第四章 自由函数的实现

**前章回顾**

C++中有许多可供所有程序员重用的自由函数，我们可以通过访问http://www.cplusplus.com/reference来获取这些函数的文档。在编程过程中，将相关的代码组合成一个定义明确的函数，以供日后测试并调用，已经被认为是一种非常好的实践经验，这有助于我们构建出可读性更强的程序。但C++本身不可能提供我们每个人在所有应用场景中所需要用到的函数。

**本章提要**

本章将重点介绍如何编写属于自己的函数。我们希望在完成本章的学习之后，你将掌握：

* 如何实现自由函数。
* 如何传递相关的值给你的函数，以作为它的输入。
* 如何从你的函数中返回相对值，以作为它的输出。
* 如何测试你的新函数。
* 初步理解对象与函数的所在域。

## 4.1 实现属于自己的函数

从上一章提到的那些函数（譬如min、max、abs、round和sqrt）可以看出，函数的定义应该是一个函数头信息加上一个语句块。

**通用格式 4.1**： *自由函数*

*function-heading* *block*

其中，*block*是一个以{开头，并以}结尾的语法单元，其内容主要由变量声明和执行语句等部分组成。

**通用格式 4.2**: *语句块*

{ *object-initializations* *statements* }

函数的输入就是调用该函数的实参。在获取输入之后，函数会利用这些输入值计算出某个结果，然后将其返回给它的调用者。我们之前已经介绍过如何通过实参与形参关联配对的方式将值输入到函数中，下面我们来介绍如何通过返回语句将值传回给调用该函数的代码。

**通用格式 4.3**： *返回语句*

return *expression* ;

我们可以具体示范一下将值返回给调用方代码的做法：

int minOf3(int a, int b, int c) {  
 // post: Return the smallest value amongst the 3 arguments  
 return min(a, min(b, c));  
}

当程序执行到返回语句时，return后面的表达式将会实际替换掉调用方代码调用该函数的位置，以作为程序控制权返回的地方。接下来，我们来实现一个名为f的函数。需要提醒的是，这个函数的编码必须写在调用它的函数之前，也就是说，函数f必须位于调用它的main函数之前。

#include <iostream> // For cout  
#include <cmath> // For pow  
using namespace std;  
  
double f(double x) { // post: Return 2 \* x \* x - 1  
 double result;  
 result = 2 \\* pow(x, 2) - 1.0;  
 return result;  
}  
  
int main() {  
 double x, y;  
 cout << "Input x: ";  
 cin >> x;  
 // Call function f:  
 y = f(x);  
 cout << "f(" << x << ") = " << y << endl;  
 return 0;  
}

**程序会话**

Input x: ***1.01***  
f(1.01) = 1.0402

**自检题**

4-1 请根据之前示例中实现的，写出下列函数调用的返回值，如果调用出错，请说明出错原因。

* 1. f(0.0)
  2. f(-2.0)
  3. f(3)
  4. f(1, 2)
  5. f()
  6. f(5.8)

在接下来的这个示例中，函数serviceCharge所声明的返回值类型是double，因此调用serviceCharge函数的地方将会被替换成某个double值，而该值则取决于其调用实参的值。

// Call serviceCharge to determine a bank debit  
#include <iostream>  
using namespace std;  
  
const double MONTHLY\_FEE = 5.00;  
  
double serviceCharge(int checks, int ATMs) {  
 // pre: checks >= 0 and ATM >= 0  
 // post: Return a banking fee based on local rules  
 double result;  
  
 result = 0.25 \* checks + 0.10 \* ATMs + MONTHLY\_FEE;  
 return result;  
}  
  
int main() {  
 // 0. Initialize objects  
 int checks;  
 int ATMs;  
 double fee; // Stores the function return result  
  
 // 1. Input  
 cout << "Checks this month? ";  
 cin >> checks;  
 cout << "ATMs this month? ";  
 cin >> ATMs;  
  
 // 2. Process  
 fee = serviceCharge(checks, ATMs); //Call to serviceCharge  
  
 // 3. Output  
 cout << "Fee: " << fee << endl;  
  
 return 0;  
}

**程序会话**

Checks this month? ***17***  
ATMs this month? ***9***  
Fee: 10.15

如你所见。上述程序在运行过程中发生了以下事情：

1. 用户被要求输入自己使用支票和ATM进行交易的次数。
2. 调用实参的值（即17和9）被传递给了serviceCharge函数的形参（即checks = 17和ATMs = 9），该函数将根据这些被指定的值来返回当前月需支付的银行手续费。
3. serviceCharge函数中的语句都会被执行。
4. serviceCharge函数最终会遇到return关键字。
5. main函数中的函数调用serviceCharge(checks, ATMs)将会被替换成该函数返回的值10.15。
6. 函数的返回值将会被赋值给fee。
7. fee会被输出显示。

### 4.1.1 测试驱动程序

当我们需要用实参调用函数时，在两个不同的地方声明相同变量名声明的情况并不罕见。以之前的程序为例，main函数中声明的checks和ATMs同时也是serviceCharge函数的形参。main函数中声明的对象是用来获取用户输入的，而serviceCharge函数声明的形参则用来获取来自main函数的输入。尽管它们使用了相同的变量名，但它们是不同的变量。

当然，有时候我们也不需要在形参上复制main函数中的同名对象。在下面的程序中，我们将会看到这回没有用户输入操作了，自然也 就没有对象需要复制。取而代之的是，这会我们用来测试函数的实参是一些常量，而且这次也不再需要将返回值赋值给另一个对象了，程序会直接显示返回结果。这段程序唯一的功能就是测试函数——验证其返回值是否符合预期。在将该函数应用到更大的程序中之前，做一些这样的测试是一件好事。事实上，许多编程问题都需要我们进行这种形式的测试。

// The main function makes several calls to test a new function  
  
#include <iostream>  
using namespace std;  
  
const double MONTHLY\_FEE = 5.00;  
  
double serviceCharge(int checks, int ATMs) {  
 // pre: checks >= 0 and ATM >= 0  
 // post: Return a banking fee based on local rules  
 double result;  
 result = 0.25 \* checks + 0.10 \* ATMs + MONTHLY\_FEE;  
 return result;  
}  
  
int main() {  
 // Test drive serviceCharge // Sample problems:  
 cout << serviceCharge(0, 0) << endl; // 5.0  
 cout << serviceCharge(1, 0) << endl; // 5.25  
 cout << serviceCharge(0, 1) << endl; // 5.1  
 cout << serviceCharge(1, 1) << endl; // 5.35  
 return 0;  
}

**程序输出**

5  
5.25  
5.1  
5.35

这个版本的main函数叫做*测试驱动程序（test driver）*，这是一种专门用于测试新函数的程序。譬如，像serviceCharge、sqrt和pow这些函数通常都只是较大型程序中的一小部分，因此在它们被重用之前全都应该经过全面性的测试。另外，对于上面所呈现的四个样本问题。我们都在对应的注释中注明了自己的预期值。如你所见，对于serviceCharge函数来说这是一个成功的测试程序。

### 4.1.2 只有一条返回语句的函数

有时候我们会看到一些极其简单的函数，它们可能只有一条返回语句。譬如：

double serviceCharge(int checks, int ATMs) {  
 // pre: checks >= 0 and ATM >= 0  
 // post: Return a banking fee based on local rules  
 return 0.25 \* checks + 0.10 \* ATMs + MONTHLY\_FEE;  
}

但除了上面这种极其简单（即只包含一条返回语句）的函数外，本书在定义函数时都将遵守以下约定：

1. 我们会声明一个名为result的局部变量，并且它的类型与函数定义的返回类型相同。
2. 然后将最终所需的值存储到变量result中。
3. 最后返回result。

虽然，这种约定在简单函数中会显得有些多余。但在第7章“选择操作”之后，这种模式将会非常有助于我们面对那些日益复杂的处理操作。

除此之外，看似多余的这两行代码有时还能避免一个非常常见的错误。这或许是因为其他编程语言在技术上支持这样做，或许是因为想当然，总之我们常常会试图将一个值赋值给函数名，这会导致编译时错误，因为只有变量才能被赋值。

double serviceCharge(int checks, int ATMs) {  
 // You cannot assign a value to a function name  
 serviceCharge = 0.25 \* checks + 0.10 \* ATMs + 5.00; // ERROR  
 return serviceCharge; // ERROR, attempt to return function  
}

当然，如果你真的犯了这个常见的错误，编译器会告知你的。我们只需要在表达式中加上正确的返回值类型double就可以修复这个错误，然后就可以返回这个函数了。

**自检题**

4-2 请根据下面给出的函数f1，写出f1(9.0)这个调用的返回值。

double f1(double x) {  
 // pre: x is zero or positive, but not 1.0  
 // post: Return f(x) = (square root of x) / ( x - 1.0 )  
 return sqrt(x)/(x - 1.0);  
}

4-3 f1(-1.5)这个函数调用是否满足上述函数的前置条件？使用负数做实参调用f1函数会发生什么情况？

4-4 请分别说明下列函数中错误的修复方法。

* a
* double f1(int j);{  
   return 2.5 \* j;  
  }
* b
* double f2(int) {  
   return 2.5 \* j;  
  }
* c
* double f3(int x) {  
   return 2.5 \* j;  
  }
* d
* double f4(double x) {  
   return 2.5 \* x;  
  }
* e
* double f5(double x) {  
   return double; ;  
  }
* f
* int f6(string s) {  
   return s;  
  }

4-5 请编写一个名为times3的函数，该函数的功能是返回一个三倍于其实参的值（譬如times3(2.0)应该返回6.0）。

## 4.2 分析、设计与实现

与编写程序相比，实现函数所需要考虑的问题只是一个大型程序中的一小部分。它可能只是某个算法中的一个步骤，只不过经常会被调用罢了。

*问题：* 计算出两点之间的距离。

### 4.2.1 分析

还记得吗？我们在程序开发的分析阶段会涉及到如何确定输入与输出，另外还有用计算机解决问题时常涉及的IPO算法模式。也就是说，开发者必须得先确定哪些是必须要发送给用户的输出，哪些是用户所需要提供的输入。在函数的设计中，我们只需要将之前的*用户*替换成*客户端*，IPO模式一样是可以适用的。除此之外，函数的输出现在变成了函数的返回语句，而输入则变成了函数实参与形参的关联配对。下面，我们来描述一个适用于函数而不是程序的通用IPO算法模式。

**适用于函数的IPO模式**

* 输入：经由实参与形参之间的关联配对动作将相关值输入给函数。
* 处理：计算出待返回的结果。
* 输出：返回结果。

问题样例是我们确定自己是否已经理解目标问题是一个好方法。而且在程序测试过程中，这些样例所提供的预期结果也可以被拿来与程序的输出进行比对。同样地，为新函数开发一组问题样例也是一个很好的主意，我们可以通过这些样例来确定函数所需的输入。也就是要在函数头信息中写明的形参类型及其数量。当然，问题样例也为我们的测试驱动程序提供了预期的输出值。

根据以下公式可以看出，想要计算出（, ）与（, ）这两点之间的距离，我们需要四个double类型的值：

下面是几组、、、的值以及预估的输出。

**问题样例**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | Distance |
| 1.0 | 1.0 | 2.0 | 2.0 | 1.414 |
| 0.0 | 0.0 | 3.0 | 4.0 | 5 |
| -5.7 | 2.5 | 3.3 | –4.7 | 11.5256 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

接下来，我们要将IPO模式应用到该函数中去：

* 输入：我们会以（，）和（，）这两点为参数来调用函数。
* 处理：按照这个运算公式来进行求值。
* 输出：返回计算结果。

### 4.2.2 设计

作为设计者，我们必须要确定函数需要多少参数以及这些参数的类型。在当前这个示例中，我们需要四个值来表示两个点（分别命名为x1、y1、x2、y2）。这些形参最合适的类型应该是double（即5.62、-9.864这样的值）。另外，函数返回值最合适的类型也是double，因为我们的计算将涉及到平方根函数，double类型有助于返回较为精确的答案。最后，该函数将被命名为distance，这个函数名很清晰地说明了这个函数的功能。总结一下，这个函数的头信息应该是：返回值类型为double、函数名为distance、四个具有描述性名称的double类型的形参。具体如下：

double distance(double x1, double y1, double x2, double y2)  
// post: Return distance between two points (x1, y1) and (x2, y2)

接下来，在函数体内（即之前提到的语句块），我们要将x1、y1、x2、y2这四个形参代入距离计算公式，计算出结果：

result = sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2));

### 4.2.3 实现

在下面的程序中，除了上述所有内容的实现，我们还专门为其编写了一个用作测试函数的main函数（即测试驱动程序）：

// Call distance four times  
#include <iostream> // For cout  
#include <cmath> // For sqrt and pow  
using namespace std;  
  
double distance(double x1, double y1, double x2, double y2) {  
 // post: Return the distance between any two points  
 double result;  
 result = sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2));  
  
 return result;  
}  
  
int main() {  
 // Test drive the distance function  
 cout << "(1.0, 1.0) (2.0, 2.0): "  
 << distance(1.0, 1.0, 2.0, 2.0) << endl;  
 cout << "(0.0, 0.0) (3.0, 4.0): "  
 << distance(0.0, 0.0, 3.0, 4.0) << endl;  
 cout << "(-5.7,2.5) (3.3,-4.7): "  
 << distance(-5.7,2.5, 3.3,-4.7) << endl;  
 cout << "(0.0, 0.0) (0.0, 0.0): "  
 << distance(0.0, 0.0, 0.0, 0.0) << endl;   
 return 0;  
}

**程序输出**

(1.0, 1.0) (2.0, 2.0): 1.41421  
(0.0, 0.0) (3.0, 4.0): 5  
(-5.7,2.5) (3.3,-4.7): 11.5256  
(0.0, 0.0) (0.0, 0.0): 0

实参与形参之间的关联配对操作与程序的输入操作非常类似。以上述代码中对distance函数的第二次调用为例，作为输入，以下四个值会这样被复制给distance函数：

double distance(x1, y1, x2, y2)  
 ↑ ↑ ↑ ↑   
 distance(0.0, 0.0, 3.0, 4.0)

接下来，控制权就被移交给了函数，它将会利用其形参计算出实参所表示的那两点之间的距离，以下是它的计算步骤：

sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2))  
sqrt(pow((0.0 - 3.0), 2) + pow((0.0 - 4.0), 2))  
sqrt(pow(( -3.0 ), 2) + pow(( -4.0 ), 2))  
sqrt( 9.0 + 16.0 )  
sqrt( 25.0 )  
 5.0 )

这四个实参可以成为函数的输入，是因为系统会负责将四个实参的值复制给其各自所关联的形参。这种实参与形参之间的关联操作模式我们称之为*值传递（pass by value）*，因为传递到函数中的都是变量的值。当一个函数需要以double、int这样的小型对象输入时，函数头信息中的形参必须要写成以下这种形式：

*class-name identier*

### 4.2.4 测试

对函数进行分别测试是一个不错的想法。我们在上述程序中就是这样做的。该程序没有做别的，就是使用了几组不同的实参调用了目标函数，并输出显示了它的返回结果。请注意一下这四个调用与问题样例之间的比对。实参是函数的输入，返回结果应该与其预期结果相匹配。

建议你们也要用一个测试驱动程序来测试自己的新函数。

### 4.2.5 标识符的域

标识符的*域（scope）*指的是程序中可以引用到该标识符的那个部分。通常情况下，一个标识符的域会从它被声明之处开始，一直延续到该声明所在的语句块结束为止。还记得吗？语句块是以一对左右大括号{ }区隔开的语法单元。例如在下面的程序中，local的域应该就是one函数。这个local变量是在one函数中被声明的，其语句块以外的地方是无法引用该标识符的，包括main函数。

// Illustrate the scope of an object  
#include <iostream>  
using namespace std;  
  
const int maxValue = 9999;  
  
void one() {  
 int local = -1;  
 // The scope of local is this function  
 cout << local << endl;  
 // maxValue is known after its declaration including here:  
 cout << maxValue << endl;  
}  
  
int main() {  
 // The scope of local is limited to one() so this is an error:  
 local = 5;  
 // Function one() is known everywhere after its declaration  
 one();  
 // maxValue is known everywhere after its declaration  
 cout << maxValue << endl;  
 return 0;  
}

当变量被声明在语句块之外时（譬如上面的maxValue），该变量的域就会从其被声明之处开始，一直延续到该文件的末尾。声明在语句块内的标识符只能在该语句块内被引用，这些标识符被称之为*局部*标识符。而声明在语句块之外的标识符（像maxValue这样的）则被称之为*全局*标识符，全局标识符可以被其在文件中被声明的位置之后的任意部分引用，除非另有语句块也声明了这个标识符（也就是该标识符被重新声明了）。在后一种情况中，先声明的标识符黑被后来的重新声明隐藏。由于同一个程序中会存在多个语句块，所以确定一个对象的域有时候会是件复杂的事。例如在下面的程序中，我们声明了三个不同的int变量identier，你可以试着猜测一下它会输出什么:

// This program is a tedious test of your ability to  
// determine which of the three int variables named  
// identier are being referenced at any given point.  
#include <iostream>  
using namespace std;  
  
const int identier = 1; // Global variable  
  
void one() {  
 // This is a reference to the global identier  
 cout << "identier in one(): " << identier << endl;  
}  
  
void two() {  
 int identier = 2; // local to two()  
 cout << "identier in two(): " << identier << endl;  
}  
  
int main() {  
 int identier = 3; // local to main()  
 one();  
 two();  
 cout << "identier in main(): " << identier << endl;  
 return 0;  
}

**程序输出**

identier in one(): 1  
identier in two(): 2  
identier in main(): 3

如你所见，当函数one被调用时，它引用的是全局声明global const int identier = 1。这个全局的identier可以在所有没有另行声明identier这个标识符名的函数中被引用。因此，one函数尽管自身内部并没有声明identier。但它可以引用第一个被声明的，初始值为1的identier。 但在函数two引用identier这个标识符时，由于全局的identier就被局部的identier隐藏掉了，当程序出现执行到此处时，函数one的输出是1，函数two的输出是2。同样地，main函数最后一条语句所引用的identier也应该是main声明的局部identier，其初始值为3。

通常情况下，函数都会在其语句块开头声明一个或多个变量。这些变量都属于局部变量，因为它们只能在该函数内部被引用。同样的限制也被引用到了函数的形参上。函数的形参是一种声明在( )之间的局部变量，它并不位于函数的语句块内。这种限制为局部变量提供了一种安全保护，能确保其不会意外地被程序的其它部分修改。

void f1(double x) {  
 int local = 0;  
 str = "A"; // Error attempting to reference main's local str  
}  
  
int main() {  
 string str; // str is local to main  
 x = 5.0; // Error attempting to reference f1's parameter x  
 local = 1;  
 // Error attempting to reference f1's local return 0;  
}

**自检题**

4-6 请根据下面给出的部分程序，判断这些函数是否可引用以下标识符。其中，cin和cout的初始化在iostream文件中已经完成，因此在#include <iostream>之后，也都属于已知标识符了。

// cout b cin MAX c f1 a d f2 main e  
#include <iostream>  
using namespace std;  
  
const int MAX = 999;  
  
void f1(int a) {  
 int b;  
}  
  
void f2(double c) {  
 double d;  
}  
  
int main() {  
 int e;  
 return 0;  
}

4-7 请命名一下函数中可以被声明为局部变量的两种东西。

4-8 如果一个变量被声明在了一个函数之外，在哪些地方可以引用它？

### 4.2.6 函数名的域

那么，函数名的情况又如何呢？它们毕竟也是标识符啊，它们的域是什么呢？和cin和cout对象一样，在被包含文件（譬如cmath文件）中的这些函数名的域是从该文件开始，一直到执行了#include <cmath>指令的文件末尾。因此sqrt、pow、ceil和fabs这些函数，都可以在所有的语句块中被调用，除非这些函数名被重新声明成了其他东西。

### 4.2.7 全局标识符

到目前为止，我们遇到的问题都不算复杂，它们的规模都不算大，相信大家在自己的工作中实际上或多或少都已经接触过了。但是，当问题的规模达到一个团队的级别时，我们对于域的使用就必须要谨慎了。

由于全局标识符在被声明之后，对于程序中所有的地方都是可见的，这意味着它们是完全开放的，我们可以在一个大型程序中的任何一个地方修改它们。这样一来，就很难保障团队中没有人会在错误的时间意外地修改了某个对象。所以我们认为，尽量使用局部变量是大家需要养成的一个好习惯。也就是说，我们应该尽量使用( )之间定义的形参和{ }之间定义的对象来做事。例如下面main函数中的localX和localY就属于局部变量：

int main() {  
 double localX, localY;  
 // . . .  
}

如果我们需要在函数之间搬运数据，就需要使用实参来传递，这样的话我们就必须要在函数中声明相应的形参，这比使用全局变量x来得好：

double f(double x) { // x is local to f  
 double result; // result is local to f  
 // Do something with x . . .  
}

如果我们确实需要一个贯穿整个程序，在许多地方都会用到的值，建议将其声明成const对象：

#include <iostream>  
using namespace std;  
  
const int MAXIMUM\_ENTRIES = 100; // ... a large program with many functions may follow

然而从另一方面来说，我们在C++中也确实经常会用到全局标识符。譬如在引入这个头文件之后，cout就是一个对于任何地方都可见的对象。当然，前提是我们做了using namespace std;这条声明，免去了在引用cout时要为其加上std::前缀的麻烦，否则，我们每次都要在cout的前面（也就是它的左边）加上std::这个限定词。

#include <iostream>  
using namespace std;  
  
void f() {  
 cout << "In f\\n";  
}  
  
void g() {  
 cout << "In g\\n";  
}  
  
int main() {  
 f();  
 g();  
 cout << "In main\\n";  
 return 0;  
}

#include <iostream>  
// Equivalent code with std::  
void f() {  
 std::cout << "In f\\n";  
}  
  
void g() {  
 std::cout << "In g\\n";  
}  
  
int main() {  
 f();  
 g();  
 std::cout << "In main\\n";  
 return 0;  
}

实质上，就是using namespace std;这个声明将cout变成了一个全局标识符。这样做是好的吗？好吧，确实有大量的计算机科学家是这样认为的。因为控制台环境通常都是唯一的，cout的任何输出都只能被传递给同一个控制台环境，这种情况下我们无需关心具体是哪个函数发出的输出。

## 4.3 void函数与引用型形参

当我们将函数的返回值类型声明成关键字void时，就表示该函数不返回任何东西。与返回某些值给客户端的函数不同的是，void函数通常会改变传递给它的对象的状态。在本节，我们可以来演示一下如何实现一个叫做swap的void函数，它对自己的两个实参做了修改。首先，一个函数要想在其调用过程中修改其实参对象的状态，它就必须要使用引用型形参（reference parameter）——即在声明时加上&。其通用格式如下：

**通用格式 4.4**：*引用型形参*

*class-name* & *identier*

**示范在函数头信息中声明引用型形参**

void swap(double & parameterOne, double & parameterTwo)  
void changeFormat(ostream & cout)

（使用＆声明的）引用型形参上的变化会修改其所关联的实参，其参数名代表的是一个内存位置，实际上是对该形参所关联实参的引用。

虽然函数的形参通常得从它的调用者获取输入，但形参与实参之间有时候会建立起更强的联系。在下面第一个关于引用型形参的用法示范中，由于swap函数必须修改两个对象，而通过return语句只能返回一个值，因此该函数需要返回给调用方的值是比return语句要多的。当我们在函数头信息中给各形参名前面加上＆这个特殊符号时，该函数节后的就是实参内存位置的引用，而不是实参的复制品了。

当引用型形参上发生变更操作时，它将会引起被引用实参对象的同样变更。这是因为该函数的形参和实参指向的是内存中的同一对象。例如在下面的程序中，当swap函数修改parmOne和parmTwo这两个形参时，实参argOne和argTwo也指向了同一个修改过的对象：

// Notice the reference symbol & is in front of parmOne  
// and parmTwo. Now a change to parmOne or parmTwo alters  
// the associated object that is the argument's value.  
  
#include <iostream>  
using namespace std;  
  
// Swap the values of any two int arguments.  
// The & lets any change to the parameter alter it argument  
void swap(int & parmOne, int & parmTwo) {  
 int temp = parmOne;  
 parmOne = parmTwo; // Change argument argOne in main  
 parmTwo = temp; // Change argument argTwo in main  
}  
  
int main() {  
 int argOne = 89; // argOne argTwo  
 int argTwo = 76;  
  
 cout << argOne << " " << argTwo << endl; // 89 76  
 swap(argOne, argTwo);  
 cout << argOne << " " << argTwo << endl; // 76 89  
  
 return 0;  
}

**程序输出**

89 76  
 76 89

如果我们将上述程序中的＆符号拿掉，main中的实参就不会发生任何变化。因为在这种情况下，argOne和argTwo执行的是按值传递，而不是引用传递。也就是说，在没有参考符号＆的情况下，parmOne和parmTwo中值的变化是局部性的。只在swap函数中有效，而其在main中所关联的实参则不受影响，因为它们是不同的对象。

下面，我们用一张表来说明一下引用兴趣形参与值类型形参之间的差异。

**引用型形参**：*实参与形参引用的是同一个对象*

parmOne = address of argOne and parmTwo = address of argTwo  
  
void swap(int & parmOne, int & parmTwo) {  
 parmOne  
 parmTwo 89- 76 由于parmOne与argOne指向的是相同的内存位置，所以  
 76- 89 在swap函数中将89改成76也会同步影响到argOne所指的对象。  
  
}  
  
int main() {  
 argOne  
 argTwo 由于parmTwo与argTwo指向的是相同的内存位置，所以  
 在swap函数中将76改成89也会同步影响到argOne所指的对象。  
  
}

**值类型形参**：*形参上的变化不会改变其关联实参的值*

parmOne = 89 (value of argOne) and parmTwo = 76(value of argTwo)  
  
void swap(int parmOne, int parmTwo) {  
 parmOne 89- 76 由于parmTwo中的值是以按值传递的方式传入swap函数的，  
 76- 89 所以swap中的局部操作不会影响到其他不同的函数。  
}  
  
int main() {  
 argOne 89  
 argTwo 76  
 由于是这里的参数是“按值传递”的，  
 所以main中的值不受任何影响。  
}

由于引用型形参上的变化会同步引起实参的变化，所以实参必须是一个变量，如果在这里使用字面常量表达式来调用函数，会导致编译时错误。

swap(89, 76); // Error: Argument must be a variable

**自检题**

4-9 请写出下面各段代码在执行到return 0;时arg1和arg2的值。

* 1. ```C++ #include using namespace std; void changeOr(int a, int b) { a = a \* 2 + 1;
     1. = 123; }
* int main() { int arg1 = 5; int arg2 = 5; changeOr(arg1, arg2); // arg1 \_\_\_\_ arg2 \_\_\_\_\_ return 0; } ```
  1. ```C++ #include using namespace std; void changeOr(int & a, int & b) { a = a \* 2 + 1; b = 123; }
* int main() { int arg1 = 5; int arg2 = 5; changeOr(arg1, arg2); // arg1 \_\_\_\_ arg2 \_\_\_\_ return 0; } ```

## 4.4 const的引用型形参

到目前为止，在C++的以下三种形参传递模式中，我们已经介绍了其中的两种：

1. 值传递形参 —— 这类形参主要用于传递int这类小型对象的值。
2. 引用型形参 —— 这类形参允许在被调用函数中同步修改一个或多个实参的状态。
3. const＆（引用）型形参 —— 这类形参主要考虑的是参数传递的安全和效率。

const的引用型形参通常主要用来传递一些不允许被调用函数修改的“大型”对象。大型对象通常指的是会占用大量内存的对象，譬如非常大的字符串等。至于为什么程序员要通过const的引用来传递大型对象，我们首先要理解当实参被传递给函数时，具体会发生哪些事。

如果执行的是按值传递，整个目标对象都会被复制到函数中另一个相同大小的变量中，这块数据所需要的内存就等于翻了一倍。而如果是引用型形参的话，我们就只需将目标对象的地址复制给函数即可。在这种情况下，该函数就只需要四个字节的内存。因为这是实参和形参引用的是相同的对象。而如果是const &形参的话，目标对象的地址一样会被复制给函数，该函数同样只需要增加四个字节的内存，但由于形参被声明成了const，函数中任何尝试修改形参值的动作都会被编译器标记为错误。也就是说，我们可以用const来防止被调用函数对其实参的意外更改。程序员们可以通过添加const限定词来避免相关对象因不同域中不可预知的修改而产生的bug。

|  |  |  |
| --- | --- | --- |
| 按值传递  int f1(int j) | 引用传递  int f2(string & b) | const引用传递  void f3(const int & n) |
| 这种方式需要获得足够的内存来存储整个目标对象，后者所有的字节都会被复制到函数中，而该函数对形参做的任何修改都不会影响到其实参。 | 这种方式会用四个字节的内存来存储目标对象的地址，后者的地址将会被复制给函数，当我们需要对函数的实参进行同步修改时可以使用这种方式，这也是非常有效率的方式。 | const表示的是函数实参的内容是包括修改的，任何企图修改n的操作都会导致编译时错误，这是一种安全且高效的传递方式。 |
| f1无法修改实参的状态。 | f2可以修改实参的状态。 | f3无法修改实参的状态，但这种传递方式的销效率很好。 |

我们选择使用const的引用型形参主要有两个原因。首先是执行效率——即这种方式可以让程序执行得更快。另一个因素是更好的内存利用率——即这种方式可以让函数以更小的内存存储大型对象。例如，按值传递int这类小心对象，需要在函数中分配四个字节的内存，然后将其整个复制过去。但如果大型对象也按值传递的话，可能就需要数千个字节了，该程序有可能会耗尽计算机上的可用内存。

除此之外，在按值传递的情况下，为了确保实参的每一个字节都能被复制到函数中。计算机必须要为此做大量不必要的工作。这可能会导致程序的运行速度明显变慢。而以下两种传递方式都可以让程序更有效率地使用空间（节省内存）和时间（运行速度更快）：

1. 采用引用传递的方式传递大型对象——这种方式效率很好，但也有一定的风险。
2. 采用传递const引用的方式传递大型对象——这种方式既高效又安全。

我们会推荐尽量使用第二种传递方式。这种方式可以让程序减少很多操作。在采用按值传递时，程序必须等到每个字节都从调用者复制到函数中之后才能继续。而如果我们采用const &引用的传递方式的话，就只需要传递四个字节，而且在安全性上它与值传递形参的情况是一样的（即不会改变实参的状态）。当然，如果我们需要在函数中修改实参的状态的话，那就得选择带&符号的引用传递了。

任何企图修改const引用所传递对象的操作都会导致编译时错误。const的使用一种纠错技术，它可以让编译器告知我们程序中所有针对const形参的意外修改。在这种情况下，大多数不修改对象的函数仍会继续完成调用动作——譬如string对象的length函数，我们会在下一章中讨论它们。但是，编译器也会标记出string对象的insert这一类函数的调用。而顾名思义，insert函数显然是要在string对象中添加内容的，它必然要改变该对象的状态，这时候就不能将新值赋值给一个const的引用型形参了。

void addSomeStuff(const string & str) {  
 cout << str.length() << endl; // Okay  
 str.insert(5, "xtra"); // ERROR: can not modify a const parameter  
 str = "new string"; // ERROR: Can not assign to a const parameter  
}

但是，如果我们这里采用的是值传递的形参，就不会收到这样的报错消息了，因为其实参对象根本不会被改变。

|  |  |
| --- | --- |
| 在f函数中修改x，并不会改变main函数中y的值。 | 这段代码会产生”无法修改const对象“这样的编译时错误。 |
| #include <iostream> using namespace std; double f(double x) {  double result;  // This does not modify y  x = x – 1.5;  result = 2 \* x;  return result; } int main() { // Output:  // 8  double y = 5.0; // y: 5  cout << f(y) << endl;  cout << "y: " << y << endl;  return 0; } | #include <iostream> using namespace std; double f(const double & x) {  double result;  // An error. Good!  x = x – 1.5;  result = 2 \* x;  return result; }  int main() {  double y = 5.0;  cout << f(y) << endl;  cout << "y: " << y << endl;  return 0; } |

当然，我们以后应该会注意到通常只有少量的对象可以采用const引用的传递方式，在接下来的章节中，读者们只会偶尔看到使用const 引用的方式来传递大型对象。而且，值传递形参也会比引用型形参更常见。

## 本章小结

* 函数所执行的往往是一些已经被定义好的服务，它主要借由实参与形参的关联以及return语句来实现双向通信。也就是说，其客户代码会以实参的形式将要输入的值传递给函数，后者则会通过return语句来返回结果。
* 本章还涉及了几个在函数实现中会遇到的新问题，譬如在标识符的域方面：
* 所有的标识符都必须要先声明再引用。
* 对象的域通常都只局限于声明它的语句块内。
* 对于那些未在任何语句块中声明的标识符，我们通常称之为全局标识符。譬如我们在#include <cmath>之后使用的函数sqrt，以及在#include <iostream>之后使用的对象std::cout，都属于全局标识符。
* 函数形参的域只局限于其所在的函数内。
* 函数的域应该是从函数头信息的声明之处开始，一直持续到其声明所在文件的末尾，或#include了该文件的文件末尾。
* 在进行实参与形参之间的关联配对时，我们需要记住很多细节：
* 在调用函数时所用的实参数量必须要与被调用函数的头信息中声明的形参数量相匹配。
* 在函数名之前冠以void返回值类型，代表该没有返回值。void函数是不能返回任何对东西的。
* 当我们需要让函数要返回某个值时，就必须要在函数头信息的开头将其定义成非void的返回值类型，其函数语句块中也必须包含相应的return语句，并且return语句中的表达式类型也必须与返回值类型相同。
* 在调用函数时，我们有时会需要它能接收输入——这就是形参的用途。有时会需要函数能返回某些东西——这就是return语句的用途。有时会需要在函数中直接修改客户端代码中的对象——这就是引用型形参的用途。
* 在调用函数时所用的实参通常应该与其所关联的形参属于同一类型，当然，情况也有例外，譬如int类型的实参是可以通过类型转换机制直接赋值给double类型的形参的。
* 只用于接受实参值副本的形参（输入型形参）在声明时是不带＆的值传递形参。
* 如果我们需要利用形参来修改其所关联的实参，就必须使用引用型形参（即使用＆符号来声明）——也就是说，引用型形参上发生的变化会同步给其关联实参，值传递形参则没有这样的功能。
* const的引用型形参主要用于传递大型对象，这可以避免掉一些大量耗费内存字节的复制操作——这一点声明成＆就可以做到了，而加上const则是为了增加形参的安全性。

## 练习题

1. 在{ }所括起来的语句块之间可以写多少条语句？
2. 在C++程序执行时，它首先要调用的是哪一个函数？
3. 同一个函数可以被调用一次以上吗？
4. 请写出下面程序会输出的内容：

* #include <iostream>  
  using namespace std;  
  double f2(double x, double y) {  
   return 2 \* x - y;  
  }  
    
  int main() {  
   cout << f2(1, 2.5) << endl;  
   cout << f2(-4.5, -3) << endl;  
   cout << f2(5, -2) << endl;  
   return 0;  
  }

1. 请写出下面函数会输出的内容：

* #include <iostream>  
  #include <cmath>  
  using namespace std;  
    
  double mystery(double p) {  
   return pow(p, 3) - 1;  
  }  
    
  int main() {  
   double a = 3.0;  
   cout << mystery( a) << endl;  
   cout << mystery(4.0) << endl;  
   cout << mystery( -2) << endl;  
   return 0;  
  }

1. 请编写一个返回值类型为double的sumOf3函数，它的功能是返回三个double值的和值，譬如sumOf3(1.5, 2.2, 3.7)返回的结果应该是7.4。
2. 请编写一个返回值类型为int的maxOf4，它的功能是返回四个整型实参中的最大值，譬如maxOf4(99, 2, 99, -4)返回的结果应该是99。
3. 在下面的代码中，其所引用的这些标识符的域各是什么？
   * 1. std
     2. cin
     3. MAX
     4. aaa
     5. string
     6. f
     7. result
     8. s
     9. cout

* #include <iostream>  
  #include <cmath>  
  using namespace std;  
    
  const double MAX = 2.0;  
    
  double f(double aaa) {  
   double result;  
   result = pow(3.0, aaa);  
   return result;  
  }  
    
  int main() {  
   string s = "a string";  
   cout << f(MAX);  
   return 0;  
  }

1. 值传递形参上发生的变化会导致其关联的实参被修改吗？
2. 引用型形参上发生的变化会导致其关联的实参被修改吗？
3. 请写出下面程序会输出的内容：

* #include <iostream>  
  using namespace std;  
    
  void changeArgs(double & x, double & y) {  
   x = x - 1.1;  
   y = y + 2.2;  
  }  
    
  int main() {  
   double a = 3.3;  
   double b = 4.4;  
    
   cout << a << " " << b << endl;  
   changeArgs(a, b);  
   cout << a << " " << b << endl;  
   changeArgs(a, b);  
   cout << a << " " << b << endl;  
   return 0;  
  }

## 编程小技巧

1. 我们在写函数时经常会犯以下这些错误：

* 在函数的头信息后面放一个分号：
* string move(int n) ; // ERROR  
  { // many errors agged here. Remove ; from line above  
  }
  + 给函数名赋值：
  + double f(double x) {  
    f = 2 \* x; // ERROR: Can not assign value to function  
    return f; // ERROR: Can not return a function name  
    }
  + 这个问题的解决方法是：为其声明一个局部对象，将相关的值赋值给这对象，然后返回它。在函数较为简单的情况下，也可以选择直接返回表达式：
  + double f(double x) {  
    return 2 \* x;  
    }  
    // or do this when there is more going on inside the function:  
    double f(double x) {  
    double result;  
    result = 2 \* x;  
    return result;  
    }
  + 在非void函数中忘了返回值操作：
  + double f2(double x) {  
    double result;  
    result = 2 \* x;  
    // ERROR: f2 must return a number  
    }
  + 在void函数中执行了返回值操作：
  + void foo(double x) {  
    return 2 \* x; // ERROR  
    }

1. 函数之间相互通信的方式有以下几种：
   * 调用方以按值传递的方式将值和对象发送给函数。
   * 当函数的设计需要修改实参时，调用方会以引用传递的方式将实参对象发送给函数。
   * 当函数的设计不需要修改实参时，为了节省时间和内存，调用方会const引用的形式将对象传递给函数
   * 调用方会经由函数的return语句获取到函数所返回的值。
   * 调用方会经由函数中对引用型形参的修改从其关联的实参中获得被修改的值。
2. 当我们希望函数返回两个以上的值时，就应该考虑使用引用型形参。因为return语句只能返回单一对象，如果我们希望函数返回一个以上的对象，也可以考虑在return语句之外，再使用一个以上的引用型形参。

## 编程项目

### 4A 三个数之和

请编写一个命名为sumThree的函数，它的功能是返回三个double类型实参的和值。

// Test drive sumThree  
int main() {  
 cout << sumThree(1.1, 2.2, 3.3) << endl; // 6.6  
 cout << sumThree(-1, -2, 3) << endl; // 0  
 return 0;  
}

### 4B 四舍五入至n位小数

请编写一个名为round的函数，它的功能是将其double类型的实参值四舍五入至第二个实参所指定的小数位。

// Test drive round  
int main() {  
 // Arguments: number to round (-2.9), decimal places (0)  
 cout << round(-2.9, 0) << endl; // -3  
 cout << round(-2.59, 1) << endl; // -2.6  
 cout << round(0.0059, 2) << endl; // 0.01  
 cout << round(1.23467, 3) << endl; // 1.235  
 cout << round(9.999999, 4) << endl; // 10  
 return 0;  
}

### 4C 还贷问题

贷款的还款问题是一个关于贷款利率、还款次数（即还款的周期）和贷款总额的函数。请以这三个变量为实参编写一个名为payment的函数，功能是根据传入函数的三个实参值返回相应的需还款数。我们在下面为你定义好了函数头信息和相应的测试驱动程序。另外，请将最终的答案值四舍五入至两位小数（参见第3.2节），然后验证一些你得到的结果是否与在线抵押贷款计算器匹配。

#include <iostream> // For the cout object  
#include <cmath> // For pow, which you denitely need here  
using namespace std;  
  
double payment(double amtBorrowed, double interestRate, int numPeriods) {  
 // TODO: Complete this function  
}  
  
int main() { // Test drive payment  
  
 // 6.0 needs to be divided by 100 and then by 12 to become a monthly  
 // interest rate, The number of years (30) also needs to multiplied  
  
 // by 12. The following test cases represent a monthly payment.  
 cout << payment(185000.00, 6.0/100.0/12, 30\*12) << endl;  
 cout << payment(185000.00, 5.0/100.0/12, 30\*12) << endl;  
 cout << payment(185000.00, 4.0/100.0/12, 30\*12) << endl;  
  
 return 0;  
}

给定贷款总额、单次还款利率以及还款次数的还贷计算公式如下：

### 4D 人口增长预测

根据 **http://www.census.gov/popclock/** 的统计数据。直到笔者编写本书的当下，我们对美国人口增长的预估是这样的：

* 每*8秒*出生一人。
* 每*13秒*死亡一人。
* 每*40秒*迁入一位国际移民。

请根据上述信息编写一个用于人口增长预测的函数，它要能返回在任意给定天数之后，在给定当前人口基础上新增的人口数。下面是我们位该函数提供的测试驱动程序，注释中标出了其中各个调用应该产生的输出：

int main() {  
 cout << populationPrediction (320000000, 0) << endl; // 320000000  
  
 // One and two day growth:  
 cout << populationPrediction(320000000, 1) << endl; // 320006314  
 cout << populationPrediction(320000000, 2) << endl; // 320012628  
  
 // One and two year growth  
 cout << populationPrediction(320000000, 365) << endl; // 322304554  
 cout << populationPrediction(320000000, 2\*365) << endl; // 324609108  
  
 return 0;  
}

在函数通过了测试之后，请再为其编写一个可由用户在终端输入当前人口数和待预测天数的完整程序，其整个程序的回话过程应该如下：

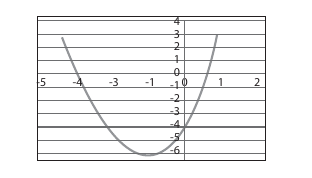
Predict population growth given the current population and days into the future  
  
Current population? 320000000  
Day into the future? \*\*\*365\*\*\*  
  
In 365 days, the population should grow by 2304554 to become 322304554

### 4E 求根公式

在下面的求根公式中，我们要使用a、b、c三个变量计算出这个二次方程式的平方根。

例如，的两个实根应该是1和−4，其函数执行时的会话过程应该如下：

Enter a b and c coef cients of a quadratic equation: \*\*\*1 3 -4\*\*\*  
  
roots: 1 and -4  
  
1x^2 + 3x + -4 when x is 1 should be 0  
This should be 0 or very close? 0  
  
1x^2 + 3x + -4 when x is -4 should be 0  
This should be 0 or very close? 0



4-1

现在，我们的要求是根据以下注释中的说明完成下面三个函数，并测试它们：

// Given the 3 coef cients, compute the two roots that  
// are made accessible as reference parameters. Assignment to  
// root1 and root2 also change the associated arguments.  
void ndBothRoots(double a, double b, double c,  
 double & root1, double & root2)  
  
// Evaluate any quadratic equation given the 3 coef cients  
// and the root in question. This function should return 0.0,  
// but something close to 0.0 like -6.66134e-16 is okay.  
// This function could return nan if b^2 – 4ac < 0 or a is 0.  
double evaluate(double a, double b, double c, double root)  
  
// Generate the requested dialog using the two functions above.  
int main()

为了避免函数在计算负数的平方根时得到非数字的结果（nan），这里可以使用负数。 但如果计算结果不是实根，就允许返回nan作为结果（或者是Visual Studio中的-1.#IND）。譬如这个二次方程式是没有实根的，所以当a、b、c依次被输入的值为3、4、2时，该程序就会显示它没有实根：

Enter a b and c coef cients of a quadratic equation: \*\*\*3 4 2\*\*\*  
  
roots: nan and nan  
  
3x^2 + 4x + 2 when x is nan should be 0  
This should be 0 or very close? nan  
  
3x^2 + 4x + 2 when x is nan should be 0  
This should be 0 or very close? nan