# 8.2 引用变量

C++新增了一种复合类型——引用变量。引用是已定义的变量的别名（另一个名称）。例如，如果将twain作为clement量的引用，则可以交替使用twain和clement来表示该变量。那么，这种别名有何作用呢？是否能帮助那些不知道如何选择变量名的人呢？有可能，但引用变量的主要用途是用作函数的形参。通过将引用变量用作参数，函数将使用原始数据，而不是其副本。这样除指针之外，引用也为函数处理大型结构提供了一种非常方便的途径，同时对于设计类来说，引用也是必不可少的。然而，介绍如何将引用用于函数之前，先介绍一下定义和使用引用的基本知识。下述讨论旨在说明引用是如何工作的，而不是其典型用法。

## 8.2.1 创建引用变量

前面讲过，C和C++使用&符号来指示变量的地址。C++给&符号赋赋予了另一个含义，将其用来声明引用。例如，要将rodents作为rats变量的别名，可以这样做：

int rats;

int &rodents = rats; // 将rodents作为rats的别名

其中，&不是地址运算符，而是类型标识符的一部分。就像声明中的char\*指的是指向char的指针一样，int &指的是指向int的引用。上述引用声明允许将rats和rodents互换——它们指向相同的值和内存单元，程序清单8.2表明了这一点。

**程序清单8.2 firstref.cpp**

// firstref.cpp -- 定义并使用引用

#include <iostream>

int main()

{

 using namespace std;

 int rats = 101;

 int & rodents = rats; // rodents是引用变量

 cout << "rats = " << rats;

 cout << ", rodents = " << rodents << endl;

 rodents++;

 cout << "rats = " << rats;

 cout << ", rodents = " << rodents << endl;

// some implementations require type casting the following

// addresses to type unsigned

 cout << "rats address = " << &rats;

 cout << ", rodents address = " << &rodents << endl;

 cin.get();

 return 0;

}

请注意，下述语句中的&运算符不是地址运算符，而是将rodents的类型声明为int &，即指向int变量的引用：

int & rodents = rats;

但下述语句中的&运算符是地址运算符，其中&rodents表示radents引用的变量的地址：

cout<<”,rodents address:” << &rodents << endl;

下面是程序清单8.2中程序的输出：

rats = 101,rodents = 101

rats = 102,rodents = 102

rats address = 0x0065fd48 , rodents address = 0x0065fd48

从中可知，rats和rodents的值和地址都相同（具体的地址和显示格式随系统而异）。将rodents加1将影响这两个变量。更准确地说，rodent++操作将一个有两个名称的变量加1。

对于C语言用户而言，首次接触到引用时可能也会有些困惑，因为这些用户很自然地会想到指针，但它们之间还是有区别的。例如，可以创建指向rats的引用和指针：

int rats = 101；

int & rodents = rats; // rodents是引用

int \* prats = &rats; // prats是指针

这样，表达式rodents和\*prats都可以同rats互换：而表达式&rodents和prats都可以同&rats互换。这点来说，引用看上去很像伪装表示的指针（其中，\*解除引用运算符被隐式理解）。实际上，引用还是不同于指针的。除了表示法不同外，还有其他的差别。例如，差别之一是，必须在声明引用时将其初始化，而不能像指针那样，先声明，再赋值：

int rat;

int & rodent;

rodent = rat; // 不可以这样做

**注意**：必须在声明引用变量时进行初始化。

引用更接近const指针，必须在创建时进行初始化，一旦与某个变量关联起来，就将一直效忠于它。也就是说：

int & rodents = rats;

实际上是下述代码的伪装表示：

int \* const pr = &rats;

其中，引用rodents扮演的角色与表达式\*pr相同。

程序清单8.3演示了试图将rats变量的引用改为bunnies变量的引用时，将发生的情况。

**程序清单8.3 sceref.cpp**

// secref.cpp -- 定义并使用引用

#include <iostream>

int main()

{

 using namespace std;

 int rats = 101;

 int & rodents = rats; // rodents是一个引用

 cout << "rats = " << rats;

 cout << ", rodents = " << rodents << endl;

 cout << "rats address = " << &rats;

 cout << ", rodents address = " << &rodents << endl;

 int bunnies = 50;

 rodents = bunnies; // 我们能改变引用吗？

 cout << "bunnies = " << bunnies;

 cout << ", rats = " << rats;

 cout << ", rodents = " << rodents << endl;

 cout << "bunnies address = " << &bunnies;

 cout << ", rodents address = " << &rodents << endl;

 cin.get();

 return 0;

}

下面是程序清单8.3中程序的输出：

rats = 101, rodenta = 101

rats addresfa = 0x0065fd44,rodents address = 0x0065fd44

bunnies = 50,rats = 50,rodents = 50

bunnies address = 0x0065fd48,rodents address = 0x0065fd44

最初，rodents引用的是rats，但随后程序试图将rodents作为bunnies的引用：

rodents = bunnies;

咋一看，这种意图暂时是成功的，因为rodents的值从101变为了50。但仔细研究将发现，rats也变成了50，同时rats和rodents的地址相同，而该地址与bunnies的地址不同。由于rodents是rats的别名，因此上述赋值语句与下面的语句等效：

rats = bunnies;

也就是说，这意味着“将bunnies变量的值赋给rat变量”。简而言之，可以通过初始化声明来设置引用，但不能通过赋值来设置。

假设程序员试图这样做：

int rats = 101;

int \* pt = &rats;

int & rodents = \*pt;

int bunnies = 50;

pt = &bunnies;

将rodents初始化为\*pt使得rodents指向rats。接下来将pt改为指向bunnies，并不能改变这样的事实，即rodcnts引用的是rats。

## 8.2.2 将引用用作函数参数

引用经常被用作函数参数，使得函数中的变量名成为调用程序中的变量的别名。这种传递参数的方法称为**按引用传递**。按引用传递允许被调用的函数能够访问调用函数中的变量。C++新增的这项特性是对C语言的超越，C语言只能按值传递。按值传递导致被调用函数使用调用程序的值的拷贝。当然，C语言也允许避开按值传通的限制，采用按指针传递的方式。

现在我们通过一个常见的的计算机问题——交换两个变量的值，对使用引用和使用指针做一下比较。交换函数必须能够修改调用程序中的变量的值。这意味着按值传递变量将不管用，因为函数将交换原始变量副本的内容，而不是变量本身的内容。但传递引用时，函数将可以使用原始数据。另一种办法是，传递指针来访问原始数据。程序清单8.4演示了这三种方法，其中包括一种不可行的方法，以便您能对这些方法进行比较。

**程序清单8.4 swaps.cpp**

// swaps.cpp -- 使用引用和指针交换数据

#include <iostream>

void swapr(int & a, int & b); // a, b是int变量的别名

void swapp(int \* p, int \* q); // p, q是int变量的地址

void swapv(int a, int b); // a, b是新的变量

int main()

{

 using namespace std;

 int wallet1 = 300;

 int wallet2 = 350;

 cout << "wallet1 = $" << wallet1;

 cout << " wallet2 = $" << wallet2 << endl;

 cout << "Using references to swap contents:\n";

 swapr(wallet1, wallet2); // 传递变量

 cout << "wallet1 = $" << wallet1;

 cout << " wallet2 = $" << wallet2 << endl;

 cout << "Using pointers to swap contents again:\n";

 swapp(&wallet1, &wallet2); // 传递变量的地址

 cout << "wallet1 = $" << wallet1;

 cout << " wallet2 = $" << wallet2 << endl;

 cout << "Trying to use passing by value:\n";

 swapv(wallet1, wallet2); // 传递变量的值

 cout << "wallet1 = $" << wallet1;

 cout << " wallet2 = $" << wallet2 << endl;

 cin.get();

 return 0;

}

void swapr(int & a, int & b) // 使用引用

{

 int temp;

 temp = a; // 使用a, b作为变量的值

 a = b;

 b = temp;

}

void swapp(int \* p, int \* q) // 使用指针

{

 int temp;

 temp = \*p; // 使用\*p, \*q作为变量的值

 \*p = \*q;

 \*q = temp;

}

void swapv(int a, int b) // 尝试使用变量的值

{

 int temp;

 temp = a; // 使用a, b作为变量的值

 a = b;

 b = temp;

}

下面是程序清单8.4中程序的输出：

wallet1 = $300 wallet2 = $350

Using references to swap contents

wallet1 = $350 wallet2 = $300

Using pointers to swap contents again:

walletl = $300 waller2 = $350

Trying to use passing by value;

walletl = $300 wallet2 = $350

正如您预想的，引用和指针方法都成功地交换了两个钱夹（wallet）中的内容，而按值传递的办法没能完成这项任务。

### 程序说明

首先来看程序清单8.4中每个函数是如何被调用的：

swapr(wallet1, wallet2); // 传递变量

swapp(&wallet1, &wallet2); // 传递变量的地址

swapv(wallet1, wallet2); // 传递变量的值

按引用传递（swapr(wallet1,wallet2)）和按值传递（swapv(wallet1,waller2)）看起来相同。只能通过原型或函数定义才能知道swapr()是按引用传递的。然而，地址运算符（&）使得按地址传递（swapp(&wallet1,&wallet2)）一目了然（类型声明int \* p表明，p是一个int指针，因此与p对应的参数应为地址，如&wallet1）。

接下来，比较函数swapr()（按引用传递）和swapv()（按值传递）的代码，唯一的外在区别是声明函数参数的方式不同：

void swapr(int & a, int & b);

void swapv(int a, int b);

当然还有内在区别：在swapr()中，变量a和b是wallet1和wallet2的别名，所以交换a和b的值相当于交换wallet1和wallet2的值；但在swapv()中，变量a和b是复制了wallet1和wallet2的值的新变量，因此交换a和b的值并不会影响wallet1和wallet2的值。

最后，比较函数swapr()（传递引用）和swapp()（传递指针）。第一个区别是声明函数参数的方式不同：

void swapr(int & a, int & b);

void swapp(int \* p, int \* q);

另一个区别是指针版本需要在函数使用p和q的整个过程中使用解除引用运算符。

前面说过，应在定义引用变量时对其进行初始化。函数调用使用实参初始化形参，因此函数的引用参数被初始化为函数调用传递的实参。也就是说，下面的函数调用将形参a和b分别初始化为wallet1和wallet2：

Swapr(wallet1,wallet2);

## 8.2.3 引用的属性和特别之处

使用引用参数时，需要了解其一些特点。首先，请看程序清单8.5。它使用两个函数来计算参数的立方，其中一个函数接受double类型的参数，另一个接受double引用。为了说明这一点，我们有意将计算立方的代码编写得比较奇怪。

**程序清单8.5 cubes.cpp**

// cubes.cpp -- 常规和引用参量

#include <iostream>

double cube(double a);

double refcube(double &ra);

int main ()

{

 using namespace std;

 double x = 3.0;

 cout << cube(x);

 cout << " = cube of " << x << endl;

 cout << refcube(x);

 cout << " = cube of " << x << endl;

 cin.get();

 return 0;

}

double cube(double a)

{

 a \*= a \* a;

 return a;

}

double refcube(double &ra)

{

 ra \*= ra \* ra;

 return ra;

}

下面是该程序的输出：

27 = cube of 3

27 = cube of 27

refcube()函数修改main()中的x值，而cube()没有，这提醒我们为何通常按值传递。变量a位于cube()中，它被初始化为x的值，但修改a并不会影响x。但由于refcube()使用了引用参数，因此修改ra实际上就是修改x。如果程序员的意图是让函数使用传递给它的信息，而不对这些信息进行修改，同时又想使用引用，则应使用常量引用。例如，在这个例子中，应在函数原型和函数头中使用const：

double refcube(const double &ra);

如果这样做，当编译器发现代码修改了ra的值时，将生成错误消息。

顺便说一句，如果要编写类似上述示例的函数（即使用基本数值类型），应采用按值传递的方式，而不要采用按引用传递的方式。当数据比较大（如结构和类）时，引用参数将很有用，您稍后便会明白这一点。

按值传递的函数，如程序清单8.5中的函数cube()，可使用多种类型的实参。例如，下面的调用都是合法的：

double z = cube(x + 2.0); // 求x+2.0的值，按值传递

z = cube(8.0); // 按值传递8.0

int k = 10;

z = cube(k); // 将k的值转换为double类型，然后按值传递

double yo[3] = { 2.2,3.3,4.4 };

z = cube(yo[2]); // 按值传递4.4

如果将与上面类似的参数传递给接受引用参数的函数，将会发现，传递引用的限制更严格。毕竟，如果ra是一个变量的别名，则实参应是该变量。下面的代码不合理，因为表达式x+3.0并不是变量：

double z = refcube(x+3.0）; // 不能进行编译

例如，不能将值赋给该表达式：

x+3.0 = 5.0; // 无意义

如果试图使用像refcub(x+3.0)这样的函数调用，在现代C++中，这是错误的，大多数编译器都将指出这一点。

### 临时变量、引用参数和const

如果实参与引用参数不匹配，C++将生成临时变量。当前，仅当参数为const引用时，C++才允许这样做，但以前不是这样。下面来看看何种情况下，C++将生成临时变量，以及为何对const引用的限制是合理的。

首先，什么时候将创建临时变量呢？如果引用参数是const，则编译器将在下面两种情况下生成临时变量：

* 实参的类型正确，但不是左值；
* 实参的类型不正确，但可以转换为正确的类型。

左值是什么呢？左值参数是可被引用的数据对象，例如，变量、数组元素、结构成员、引用和解除引用的指针都是左值。非左值包括字面常量（用引号括起的字符串除外，它们由其地址表示）和包含多项的表达式。在C语言中，左值最初指的是可出现在赋值语句左边的实体，但这是引入关键字const之前的情况。现在，常规变量和const变量都可视为左值，因为可通过地址访问它们。但常规变量属于可修改的左值，而const变量属于不可修改的左值。

回到前面的示例。假设重新定义了refcube()。使其接受一个常量引用参数：

double refcube(const double &ra)

{

 return ra \* ra \* ra;

}

现在考虑下面的代码：

double side = 3.0;

double \* pd = &side;

double &rd = side;

long edge = 5L;

double lens[4] = { 2.0,5.0,10.0,12.0}:

double c1 = refcube(side); // ra is side

double c2 = refcube(lens[2]); // ra is lens[2]

double c3 = refcube(rd); // ra is rd is side

double c4 = refcube(\*pd); // ra is \*pd is side

double c5 = refcube{edge}; // ra is temporary variable

double c6 = refcube(7.0); // ra is temporary variable

doubre c7 = refcube(side + 10.0); // ra is tamporary variable

参数side、lens[2]、rd和\*pd都是有名称的、double类型的数据对象，因此可以为其创建引用，而不需要临时变量（还记得吗：数组元素的行为与同类型的变量类似）。然而，edge虽然是变量，类型却不正确，double引用不能指向long。另一方面，参数7.0和side+10.0的类型都正确，但没有名称，在这些情况下，编泽器都将生成一个临时匿名变量，并让ra指向它。这些临时变量只在函数调用期间存在，此后编译器便可以随意将其删除。

那么为什么对于常量引用，这种行为是可行的，其他情况下，却不行的呢？对于程序清单8.4中的函数swapr()：

void swapr(int & a, int & b) // 使用引用

{

 int temp;

 temp = a; // 使用a, b作为变量的值

 a = b;

 b = temp;

}

如果在早期C++较宽松的规则下，执行下面的操作将发生什么情况呢？

long a=3,b=5;

swapr(a,b);

这里的类型不匹配，因此编译器将创建两个临时int变量，将它们初始化为3和5，然后交换临时变量的内容，而a和b保持不变。

简而言之，如果接受引用参数的函数的意图是修改作为参数传递的变量，则创建临时变量将阻止这种意图的实现。解决方法是，禁止创建临时变量，现存的C++标准正是这样做的（然而，在默认情况下，有些编译器仍将发出警告，而不是错误消息，因此如果看到了有关临时变量的警告，请不要忽略）。

现在来看refcube()函数。该函数的目的只是使用传递的值，而不是修改它们，因此临时变量不会造成任何不利的影响，反而会使两数在可处理的参数种类方面更通用。因此，如果声明将引用指定为const，C++将在必要时生成临时变量。实际上，对于形参为const引用的C++函数，如果实参不匹配，则其行为类似于按值传递，为确保原始数据不被修改，将使用临时变量来存储值。

**注意**：如果函数调用的参数不是左值或与相应的const引用参数的类型不匹配，则C++将创建类型正确的匿名变量，将函数调用的参数的值传递给该匿名变量，并让参数来引用该变量。

### 应尽可能使用const

将引用参数声明为常量数据的引用的理由有三个：

* 使用const可以避免无意中修改数据的编程错误；
* 使用const使函数能够处理const和非const实参，否则将只能接受非const数据；
* 使用const引用使函数能够正确生成并使用临时变量。

因此，应尽可能将引用形参声明为const。

C++11新增了另一种引用——右值引用（rvaluc reference）。这种引用可指向右值，是使用&&声明的：

double && rref = std::sqrt(36.00); // not allowed for double &

double j = 15.0;

double && jref = 2.0\*j + 18.5; // not allowed for double &

std::cout << rref << ‘\n’; // display 6.0

std::cout << jref << ‘\n’; // display 48.5;

新增右值引用的主要目的是，让库设计人员能够提供有些操作的更有效实现。第18章将讨论如何使用右值引用来实现移动语义（move semantics）。以前的引用（使用&声明的引用）现在称为左值引用。

## 8.2.4 将引用用于结构

引用非常适合用于结构和类（C++的用户定义类型）。确实，引入引用主要是为了用于这些类型的，而不是基本的内置类型。

使用结构引用参数的方式与使用基本变量引用相同，只需在声明结构参数时使用引用运算符&即可。

例如，假设有如下结构定义：

struct free\_throws

{

 std::string name;

 int made;

 inc attempts;

 float percent;

};

则可以这样编写函数原型，在函数中将指向该结构的引用作为参数：

void set\_pc(free\_throws & ft); // 使用指向结构的引用

如果不希望函数修改传入的结构，可使用const：

void display(const free\_throws & ft); // 不允许修改结构

程序清单8.6中的程序正是这样做的。它还通过让函数返回指向结构的引用添加了一个有趣的特点，这与返回结构有所不同。对此，有一些需要注意的地方，稍后将进行介绍。

**程序清单8.6 strtref.cpp**

//strc\_ref.cpp -- 使用结构的引用

#include <iostream>

#include <string>

struct free\_throws

{

 std::string name;

 int made;

 int attempts;

 float percent;

};

void display(const free\_throws & ft);

void set\_pc(free\_throws & ft);

free\_throws & accumulate(free\_throws &target, const free\_throws &source);

int main()

{

 free\_throws one = {"Ifelsa Branch", 13, 14};

 free\_throws two = {"Andor Knott", 10, 16};

 free\_throws three = {"Minnie Max", 7, 9};

 free\_throws four = {"Whily Looper", 5, 9};

 free\_throws five = {"Long Long", 6, 14};

 free\_throws team = {"Throwgoods", 0, 0};

 free\_throws dup;

 set\_pc(one);

 display(one);

 accumulate(team, one);

 display(team);

 // 将返回值作为参数

 display(accumulate(team, two));

 accumulate(accumulate(team, three), four);

 display(team);

 // use return value in assignment

 dup = accumulate(team,five);

 std::cout << "Displaying team:\n";

 display(team);

 std::cout << "Displaying dup after assignment:\n";

 display(dup);

 set\_pc(four);

 // ill-advised assignment

 accumulate(dup,five) = four;

 std::cout << "Displaying dup after ill-advised assignment:\n";

 display(dup);

 std::cin.get();

 return 0;

}

void display(const free\_throws & ft)

{

 using std::cout;

 cout << "Name: " << ft.name << '\n';

 cout << " Made: " << ft.made << '\t';

 cout << "Attempts: " << ft.attempts << '\t';

 cout << "Percent: " << ft.percent << '\n';

}

void set\_pc(free\_throws & ft)

{

 if (ft.attempts != 0)

 ft.percent = 100.0f \*float(ft.made)/float(ft.attempts);

 else

 ft.percent = 0;

}

free\_throws & accumulate(free\_throws & target, const free\_throws & source)

{

 target.attempts += source.attempts;

 target.made += source.made;

 set\_pc(target);

 return target;

}

下面是该程序的输出：

Name: Ifelsa Branch

 Made ： 13 Attempts : 14 Percent: 92.8571

Name: Throwgoods

 Made : 13 Attempts : 14 Percent : 92.8571

Name: Throwgoods

 Made : 23 Attempts : 30 Percent : 76.6667

Name; Throwgoods

 Made : 35 Attempts; 48 Percent : 72.9167

Displaying team:

Name: Throwgoods

 Made : 41 Attempts: 62 Percent : 66.129

Displaying dup after assignment:

Name: Throwgoods

 Made : 41 Attempts: 62 Percent : 66.129

Displaying dup after ill-advised assignment:

Name: Whily Looper

 Made : 5 Attempts: 9 Percent : 55.5556

### 1．程序说明

该程序首先初始化了多个结构对象。前面讲过，如果指定的初始值比成员少，余下的成员（这里只有percent）将被设置为零。第一个函数调用如下：

set\_pc(one);

由于函数set\_pc()形参ft为引用，因此ft指向one，函数set\_pc()的代码设置成员one.percent。就这里而言，按值传递不可行，因为这将导致设置的是one的临对拷贝的成员percent。根据前一章介绍的知识，另一种方法是使用指针参数并传递地址，但要复杂些：

set\_pcp(&one); // 使用指针参数 - &one替代one

…

void set\_pcp(free\_throw \*pt)

{

 if(pt -> attempts !=0)

 pt -> percent = 100.0f \* float(pt-> made)/float(pt -> attempts);

 else

 pt -> percent = 0;

}

下一个函数调用如下：

display(one);

由于display()显示结构的内容，而不修改它，因此这个函数使用了一个const引用参数。就这个函数而言，也可按值传递结构，但与复制原始结构的拷贝相比，使用引用可节省时间和内存。

再下一个函数调用如下：

accumulate(team,one);

函数accumulate()接收两个结构参数，并将第二个结构的成员attempts和made的数据添加至第一个结构的相应成员中。只修改了第一个结构，因此第一个参数为引用，而第二个参数为const引用：

free\_throws & accumulate(free\_throws & target,const free\_throws & source};

返回值呢？当前讨论的函数调用没有使用它；就目前而言，原本可以将返回值声明为void，但请看下述函数调用：

display (accumulace (team,two));

上述代码是什么意思呢？首先，将结构对象team作为第一个参数传递给了accumulate()。这意味着在函数accumulae()中，target指向的是team。函数accumulate()修改team，再返回指向它的引用，注意到返回语句如下：

return target;

光看这条语句并不能知道返回的是引用，但函数头和原型指出了这一点：

free\_throws & accumulate(free\_throws & target , const free\_throws & source);

如果返回类型被声明为free\_throws而不是free\_throws &，上述返回语句将返到target（也就是team）的拷贝。但返回类型为引用，这意味着返回的是最初传递给accumulate()的team对象。

接下来，将accumulate()的返回值作为参数传递给了display()，这意味着将team传递给了display()。

display()的参数为引用，这意味着函数display()中的ft指向的是team，因此将显示team的内容。所以，下述代码：

display(accumulate(team,two));

与下面的代码等效：

accumulate(team,two);

display(team);

上述逻辑也适用于如下语句：

accumulate (accumulate (team, three),four);

因此，该语句与下面的语句等效：

accumulate (team , three);

accumulate (team , four);

接下来，程序使用了一条赋值语句：

dup = accumulate(iteam,five);

如您预期的，这条语句将team中的值复制dup中。

最后，程序以独特的方式使用了accumulate()：

accumulate (dup,five) = four;

这条语句将值赋给函数调用，这是可行的，因为函数的返回值是一个引用。如果函数accumulate()按值返回，这条语句将不能通过编译。由于返回的是指向dup的引用，因此上述代码与下面的代码等效：

accumulate(dup , five); // 将five的数据累加到dup

dup = four; // 用four中的内容覆盖dup中的内容

其中第二条语句消除了第一条语句所做的工作，因此在原始赋值语句使用accumulate()的方式并不好。

### 2．为何要返回引用

下面更深入地讨论返回引用与传统返回机制的不同之处。传统返回机制与按值传递函数参数类似：计算关键字return后面的表达式，并将结果返回给调用函数。从概念上说，这个值被复制到一个临时位置，而调用程序将使用这个值。请看下面的代码：

double m = sqrt(16.0);

cout << sqrt(25.0);

在第一条语句中，值4.0被复制到一个临时位置，然后被复制给m。在第二条语句中，值5.0被复制到一个临时位置，然后被传递给cout（这里是理论上的描述，实际上，编译器可能合并某些步骤）。

现在来看下面的语句：

dup = accumulate(team,five);

如果accumulate()返回一个结构，而不是指向结构的引用，将把整个结构复制到一个临时位置，再将这个拷贝复制给dup。但在返回值为引用时，将直接把team复制到dup，其效率更高。

**注意**：返回引用的函数实际上是被引用的变量的别名。

### 3．返回引用时需要注意的问题

返回引用时最重要的一点是，应避免返回函数终止时不再存在的内存单元的引用。您应避免编写下面这样的代码：

const free\_throws & clone2(free\_throws & ft)

{

 free\_throws newguy; // first step to big error

 newguy = ft; // copy info

 rerurn newguy; // return reference to copy

}

该函数返回一个指向临时变量（newguy）的引用，函数运行完毕后它将不再存在。第9章将讨论各种变量的持续性。同样，也应避免返回指向临时变量的指针。

为避免这种问题，最简单的方法是，返回一个作为参数传递给函数的引用。作为参数的引用将指向调用函数使用的数据，因此返回的引用也将指向这些数据。程序清单8.6中的accumulate()正是这样做的。

另一种方法是用new来分配新的存储空间。前面见过这样的函数，它使用new为字符串分配内存空间，并返回指向该内存空间的指针。下面是使用引用来完成类似工作的方法：

const free\_throws & clone(free\_throws & ft)

{

 free\_throws \* pt;

 \*pt = ft; // 复制信息

 return \*pt; // 返回指向副本的引用

}

第一条语句创建一个无名的free\_throws结构，并让指针pt指向该结构，因此\*pt就是该结构。上述代码似乎会返回该结构，但函数声明表明，该函数实际上将返回这个结构的引用。这样，便可以这样使用该函数：

free\_throw & jolly = clone(three);

这使得jolly成为新结构的引用。这种方法存在一个问题：在不再需要new分配的内存时，应使用delete来释放它们。调用clone()隐藏了对new的调用，这使得以后很容易忘记使用delete来释放内存。第16章讨论的auto\_ptr模板以及C++11新增的的unique\_ptr可帮助程序员自动完成释放工作。

### 4．为何将const用于引用返回类型

程序清单8.6包含如下语句：

accumulate(dup,five) = four;

其效果如下：首先将five的数据添加到dup中，再使用four的内容覆盖dup的内容。这条语句为何能够通过编译呢？在赋值语句中，左边必须是可修改的左值。也就是说，在赋值表达式中，左边的子表达式必须标识一个可修改的内存块。在这里，函数返回指向dup的引用，它确实标识的是一个这样的内存块，因此这条语句是合法的。

另一方面，常规（非引用）返回类型是右值——不能通过地址访问的值。这种表达式可出现在赋值语句的右边，但不能出现在左边。其他右值包括字面值（如10.0）和表达式（如x+y）。显然。获取字面值（如10.0）的地址没有意义，但为何常规函数返回值是右值呢？这是因为这种返回值位于临时内存单元中，运行到下一条语句时，它们可能不再存在。

假设您要使用引用返回值，但又不允许执行像给accumulate()赋值这样的操作，只需将返回类型声明为const引用：

const free\_throw &

 accumulate(free\_throws & target,const free\_throws & source);

现在返回类型为const，是不可修改的左值，因此下面的赋值语句不合法：

accumulate(dup,five) = four; // not allowed for const reference return

该程序中的其他函数调用又如何呢？返回类型为const引用后，下面的语句仍合法：

display(accumulate(team,two);

这是因为display()的形参也是const free\_throws &类型。但下面的的语句不合法，因为accumulate()的第一个形参不是const：

accumulate(accumulate(team,three),four);

这影响大吗？就这里而言不大，因为您仍可以这样做：

accumulate(team,three);

accumulate(team，four);

另外，您仍可以在赋值语句右边使用accumulate(}。

通过省略const，可以编写更简短代码，但其含义也更模糊。

通常，应避免在设计中添加模糊的特性，因为模糊特性增加了犯错的机会。将返回类型声明为const引用，可避免您犯糊涂。然而，有时候省略const确实有道理，第11章将讨论的重载运算符<<就是一个这样的例子。

## 8.2.5 将引用用于类对象

将类对象传递给函数时，C++通常的做法是使用引用。例如，可以通过使用引用，让函数将类string、ostrearh、istream、ofstream和ifstream等类的对象作为参数。

下面来看一个例子，它使用了string类，并演示了一些不同的设计方案，其中的一些是糟糕的。这个例子的基本思想是，创建一个函数，它将指定的字符串加入到另一个字符串的前面和后面。程序清单8.7提供了三个这样的函数，然而其中的一个存在非常大的缺陷，可能导致程序崩溃甚至通不过编译。

**程序清单8.7 strquote.cpp**

// strquote.cpp -- 不同的设计方案

#include <iostream>

#include <string>

using namespace std;

string version1(const string & s1, const string & s2);

const string & version2(string & s1, const string & s2); // has side effect

const string & version3(string & s1, const string & s2); // 糟糕的设计

int main()

{

 string input;

 string copy;

 string result;

 cout << "Enter a string: ";

 getline(cin, input);

 copy = input;

 cout << "Your string as entered: " << input << endl;

 result = version1(input, "\*\*\*");

 cout << "Your string enhanced: " << result << endl;

 cout << "Your original string: " << input << endl;

 result = version2(input, "###");

 cout << "Your string enhanced: " << result << endl;

 cout << "Your original string: " << input << endl;

 cout << "Resetting original string.\n";

 input = copy;

 result = version3(input, "@@@");

 cout << "Your string enhanced: " << result << endl;

 cout << "Your original string: " << input << endl;

 // cin.get();

 // cin.get();

 return 0;

}

string version1(const string & s1, const string & s2)

{

 string temp;

 temp = s2 + s1 + s2;

 return temp;

}

const string & version2(string & s1, const string & s2) // 有副作用

{

 s1 = s2 + s1 + s2;

 // safe to return reference passed to function

 return s1;

}

const string & version3(string & s1, const string & s2) // 糟糕的设计

{

 string temp;

 temp = s2 + s1 + s2;

 // unsafe to return reference to local variable

 return temp;

}

下面是该程序的运行情况：

Enter a string: It’s not my fault.

Your string as entered: It’s not my fault.

Your string enhanced: \*\*\*It's not my fault.\*\*\*

Your original string: It’s not my fault.

Your string enhanced: ###It's not my fault.###

Your original string: ###It‘s not my fault.###

Resetting original string.

此时，该程序已经崩溃。

### 程序说明

在程序清单8.7的三个函数中，version1最简单：

strirg version1 (const string & s1, const string & s2)

{

 string temp;

 temp = s2 + s1 + s2;

 return temp;

}

它接受两个string参数，并使用string类的相加功能来创建一个满足要求的新字符串。这两个函数参数都是const引用。如果使用string对象作为参数，最终结果将不变：

string version4 (string s1, string s2) // 最终结果不变

在这种情况下，s1和s2将为string对象。使用引用的效率更高，因为函数不需要创建新的string对象，并将原来对象中的数据复制到新对象中。限定符const指出，该函数将使用原来的string对象，但不会修改它。

temp是一个新的string对象，只在函数version1()中有效，该函数执行完毕后，它将不再存在。因此，返回指向temp的引用不可行，因此该函数的返回类型为string，这意味者temp的内容将被复制到一个临时存储单元中，然后在main()中，该存储单元的内容被复制到一个名为result的string中：

result = version1(input,”\*\*\*”);

### 将C-风格字符串用作string对象引用参数

对于函数version1()，您可能注意到了很有趣的一点：该函数的两个形参（s1和s2）的类型都是const string&，但实参（input和“\*\*\*”）的类型分别是string和const char\*。由于input的类型为string，因此让s1指向它没有任何问题。然而，程序怎么能够接受将char指针赋给string引用呢？

这里有两点需要说明。首先，string类定义了一种char \*到string的转换功能，这使得可以使用C-风格字符串来初始化string对象。其次是本章前面讨论过的类型为const引用的形参的一个属性。假设实参的类型与引用参数类型不匹配，但可被转换为引用类型，程序将创建一个正确类型的临时变量，使用转换后的实参值来初始化它，然后传递一个指向该临时变量的引用。例如，在本章前面，将int实参传递给const double &形参时，就是以这种方式进行处理的。同样，也可以将实参char \*或const char \*传递给形参const string &。这种属性的结果是，如果形参类型为const string &，在调用函数时，使用的实参可以是string对象或C-风格字符串，如用引号括起的字符串字面量、以空字符结尾的char数组或指向char的指针变量。因此，下面的代码是可行的：

result = version1(input , “\*\*\*”);

函数version2()不创建临时string对象，而是直接修改原来的string对象：

const string & version2(string & s1, const string & s2) // 有副作用

{

 s1 = s2 + s1 + s2;

 // safe to return reference passed to function

 return s1;

}

该函数可以修改s1，因为不同于s2，s1没有被声明为const。

由于s1是指向main()中一个对象（input）的引用，因此将s1作为引用返回是安全的。由于s1是指向input的引用，因此，下面一行代码：

result = version2(input,”###”);

与下面的代码等价：

version2(input ,”###”) // input被version2()函数直接修改

result = input; // 指向s1的引用就是指向input的引用

然而，由于s1是指向input的引用，调用该函数将带来修改input的副作用：

Your original string: It's not my fault.

Your string enhanced : ###It's not my fault.###

Your original string: ###It's not my fault.###

因此，如果要保留原来的字符串不变，这将是一种错误设计。

程序清单8.7中的第三个函数版本指出了什么不能做：

const string & version3(string & s1, const string & s2) // 糟糕的设计

{

 string temp;

 temp = s2 + s1 + s2;

 // unsafe to return reference to local variable

 return temp;

}

它存在一个致命的缺陷：返回一个指向version3()中声明的变量的引用。这个函数能够通过编译（但编译器会发出警告），但当程序试图执行该函数时将崩溃。具体地说，问题是由下面的赋值语句引发的：

result = version3(input,”@@@”）;

程序试图引用已经释放的内存。

## 8.2.6 对象、继承和引用

ostream和ofstream类凸现了引用的一个有趣属性。正如第6章介绍的，ofstream对象可以使用ostream类的方法，这使得文件输入，输出的格式与控制台输入/输出相同。使得能够将特性从一个类传递给另一个类的语言特性被称为继承，这将在第13章详细讨论。简单地说，ostream是基类（因为ofstream是建立在它的基础之上的），而ofstream是派生类（因为它是从ostream派生而来的）。派生类继承了基类的方法，这意味着ofstream对象可以使用基类的特性，如格式化方法precision()和setf()。

继承的另一个特征是，基类引用可以指向派生类对象，而无需进行强制类型转换。这种特征的一个实际结果是，可以定义一个接受基类引用作为参数的函数，调用该函数时，可以将基类对象作为参数，也可以将派生类对象作为参数。例如，参数类型为ostream &的函数可以接受ostream对象（如cout）或您声明的ofstream对象作为参数。

程序清单8.8通过调用同一个函数（只有函数调用参数不同）将数据写入文件和显示到屏幕上来说明了这一点。该程序要求用户输入望远镜物镜和一些目镜的焦距，然后计算并显示每个目镜的放大倍数。放大倍数等于物镜的焦距除以目镜的焦距，因此计算起来很简单。该程序还使用了一些格式化方法，这些方法用于cout和ofstream对象（在这个例子中为fout）时作用相同。

**程序清单8.8 filefunc.cpp**

//filefunc.cpp -- 以ostream &为参数的函数

#include <iostream>

#include <fstream>

#include <cstdlib>

using namespace std;

void file\_it(ostream & os, double fo, const double fe[],int n);

const int LIMIT = 5;

int main()

{

 ofstream fout;

 const char \* fn = "ep-data.txt";

 fout.open(fn);

 if (!fout.is\_open())

 {

 cout << "Can't open " << fn << ". Bye.\n";

 exit(EXIT\_FAILURE);

 }

 double objective;

 cout << "Enter the focal length of your "

 "telescope objective in mm: ";

 cin >> objective;

 double eps[LIMIT];

 cout << "Enter the focal lengths, in mm, of " << LIMIT

 << " eyepieces:\n";

 for (int i = 0; i < LIMIT; i++)

 {

 cout << "Eyepiece #" << i + 1 << ": ";

 cin >> eps[i];

 }

 file\_it(fout, objective, eps, LIMIT);

 file\_it(cout, objective, eps, LIMIT);

 cout << "Done\n";

 cin.get();

 cin.get();

 return 0;

}

void file\_it(ostream & os, double fo, const double fe[],int n)

{

 // 保存初始格式状态

 ios\_base::fmtflags initial;

 initial = os.setf(ios\_base::fixed, ios\_base::floatfield);

 std::streamsize sz = os.precision(0);

 os << "Focal length of objective: " << fo << " mm\n";

 os.precision(1);

 os.width(12);

 os << "f.l. eyepiece";

 os.width(15);

 os << "magnification" << endl;

 for (int i = 0; i < n; i++)

 {

 os.width(12);

 os << fe[i];

 os.width(15);

 os << int (fo/fe[i] + 0.5) << endl;

 }

 // 恢复初始格式状态

 os.setf(initial, ios\_base::floatfield);

 os.precision(sz);

}

下面是该程序的运行情况：

Enter the focal length of your telescope objective in mm:1800

Enter the focal lengths, in mm, of 5 eyepieces:

Eyepiece #1:30

Eyepiece #2:19

Eyepiece#3：14

Eyepiece #4:8.8

Eyepiece #5：7.5

Focal length of objective:1800 mm

f.1. eyepiece magnification

30.0 60

19.0 95

14.0 129

8.0 205

7.5 240

Done

下述代码行将目镜数据写入到文件ep-data.txt中：

file\_it(fout, objective, eps, LIMIT);

而下述代码行将同样的信息以同样的格式显示到屏幕上：

file\_it(cout, objective, eps, LIMIT);

### 程序说明

对于该程序，最重要的一点是，参数os（其类型为ostream &）可以指向ostream对象（如cout），也可以指向ofstream对象（如fout）。该程序还演示了如何使用ostream类中的格式化方法。下面复习（介绍）其中的一些，更详细的讨论请参阅第17章。

方法setf()让您能够设置各种格式化状态。例如，方法调用setf(ios\_base::fixed)将对象置于使用定点表示法的模式：setf(ios\_base::showpoint)将对象置于显示小数点的模式，即使小数部分为零。方法precision()指定显示多少位小数（假定对象处于定点模式下）。所有这些设置都将一直保持不变，直到再次调用相应的方法重新设置它们。方法width()设置下一次输出操作使用的字段宽度，这种设置只在显示下一个值时有效，然后将恢复到默认设置。默认的字段宽度为零，这意味着刚好能容纳下要显示的内容。

函数file\_it()使用了两个有趣的方法调用：

ios\_base:: fmtflags initial;

initial = os.setf(ios\_base::fixed); // 保存初始格式状态

…

os.setf(initial); // 恢复回复初始格式状态

方法setf()返回调用它之前有效的所有格式化设置。ios\_base::fmtflags是存储这种信息所需的数据类型名称。因此，将返回值赋给initial将存储调用file\_it()之前的格式化设置，然后便可以使用变量initial作为参数来调用setf()，将所有的格式化设置恢复到原来的值。因此，该函数将对象回到传递给file\_it()之前的状态。

了解更多有关类的知识将有助于更好地理解这些方法的工作原理，以及为何在代码中使用ios\_base。然而，您不同等到第17章才使用这些方法。

需要说明的最后一点是。每个对象都存储了自己的格式化设置。因此，当程序将cout传递给file\_it()时，cout的设置将被修改，然后被恢复；当程序将fout传递给file\_it()时，fout的设置将被修改，然后被恢复。

## 8.2.7 何时使用引用参数

使用引用参数的主要原因有两个。

* 程序员能够修改调用函数中的数据对象。
* 通过传递引用而不是整个数据对象，可以提高程序的运行速度。

当数据对象较大时（如结构和类对象），第二个原因最重要。这些也是使用指针参数的原因。这是有道理的，因为引用参数实际上是基于指针的代码的另一个接口。那么，什么时候应使用引用、什么时候应使用指针呢？什么时候应按值传递呢？下面是一些指导原则：

对于使用传递的值而不作修改的函数。

* 如果数据对象很小，如内置数据类型或小型结构，则按值传递。
* 如果数据对象是数组，则使用指针，因为这是唯一的选择，并将指针声明为指向const的指针。
* 如果数据对象是较大的结构，则使用const指针或const引用，以提高程序的效率。这祥可以节省复制结构所需的时间和空间。
* 如果数据对象是类对象，则使用const引用。类设计的语义常常要求使用引用，这是C++新增这项特性的主要原因。因此，传递类对象参数的标准方式是按引用传递。

对于修改调用函数中数据的函数：

* 如果数据对象是内置数据类型，则使用指针。如果看到诸如fixit (&x)这样的代码（其中x是int），则很明显，该函数将修改x。
* 如果数据对象是数组，则只能使用指针。
* 如果数据对象是结构，则使用引用或指针。
* 如果数据对象是类对象，则使用引用。

当然，这只是一些指导原则，很可能有充分的理由做出其他的选择。例如，对于基本类型，cin使用引用，因此可以使用cin>>n，而不是cin>>&n。