# 第 9 章　使用引用

前一章介绍了如何使用指针来操纵自由存储区中的对象以及如何间接地访问这些对象。本章将介绍的引用几乎提供了指针的所有功能，但语法更简单。

本章介绍以下内容：

- 什么是引用？
- 引用和指针的区别何在？
- 如何创建和使用引用？
- 引用的局限性是什么？
- 如何按引用将值和对象传入和传出函数？

## 9.1　什么是引用

引用是别名：创建引用时，你将其初始化为另一个对象（即目标）的名称。然后，引用将成为目标的另一个名称，对引用执行任何操作实际上都是针对目标的。

创建引用的方法是，首先给出目标对象的类型，然后加上引用运算符（&）、引用的名称、等号和目标对象的名称。

引用名可以是任何合法的变量名，但很多程序员喜欢在引用名中加上前缀 r。如果有一个名为 someInt 的 int 变量，可以编写如下代码来创建一个指向该变量的引用：

```
int &rSomeRef = someInt;
```

这条语句的含义是，rSomeRef 是一个指向 int 变量的引用，并被初始化为指向 someInt。引用和其他变量的区别在于，声明引用的同时必须对其进行初始化。如果创建引用时不给它赋值，将出现编译错误。程序清单 9.1 演示了如何创建和使用引用。

**注意：**引用运算符（&）和地址运算符的符号相同，但它们并不是同一个运算符，虽然它们之间显然是相关的。

引用运算符之前的空格必不可少，而引用运算符和引用变量名之间的空格是可选的：

```
int &rSomeRef = someInt;  //ok
int & rSomeRef = someInt;  //ok
```

**程序清单 9.1　创建和使用引用**

```
1:  //Listing 9.1 - Demonstrating the use of references
2:
3:  #include <iostream>
4:
5:  int main()
```

```
 6: {
 7:    using namespace std;
 8:    int  intOne;
 9:    int &rSomeRef = intOne;
10:
11:    intOne = 5;
12:    cout << "intOne: " << intOne << endl;
13:    cout << "rSomeRef: " << rSomeRef << endl;
14:
15:    rSomeRef = 7;
16:    cout << "intOne: " << intOne << endl;
17:    cout << "rSomeRef: " << rSomeRef << endl;
18:
19:    return 0;
20: }
```

**输出：**

```
intOne: 5
rSomeRef: 5
intOne: 7
rSomeRef: 7
```

**分析：**

第 8 行声明了一个局部 int 变量 intOne；第 9 行声明了一个 int 引用 rSomeRef，并将其初始化指向 intOne。正如前面指出的，如果声明了一个引用而没有对其进行初始化，将导致编译错误。必须对引用进行初始化。

第 11 行将 5 赋给 intOne。第 12 和 13 行打印 intOne 和 rSomeRef 的值，当然它们相同。

第 15 行，将 7 赋给 rSomeRef。由于它是引用，是 intOne 的别名，因此 7 实际上被赋给了变量 intOne，如第 16 和 17 行的打印输出所示。

## 9.2 将地址运算符用于引用

读者知道，符号&可用于获得变量的地址以及声明引用。如果将地址运算符用于引用变量将如何呢？这将返回其指向的目标的地址。这就是引用的特征，引用是目标的别名。程序清单 9.2 演示了如何获取一个名为 rSomeRef 的引用变量的地址。

**程序清单 9.2 获取引用的地址**

```
 1: //Listing 9.2 - Demonstrating the use of references
 2:
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:    using namespace std;
 8:    int  intOne;
 9:    int &rSomeRef = intOne;
10:
11:    intOne = 5;
12:    cout << "intOne: " << intOne << endl;
13:    cout << "rSomeRef: " << rSomeRef << endl;
```

```
14:
15:     cout << "&intOne: " << &intOne << endl;
16:     cout << "&rSomeRef: " << &rSomeRef << endl;
17:
18:     return 0;
19: }
```

**输出：**

```
intOne: 5
rSomeRef: 5
&intOne:  0x3500
&rSomeRef: 0x3500
```

**警告：**最后两行为内存地址，这些内容随计算机上和每次运行而异，因此读者的输出可能不同。

**分析：**

同样，rSomeRef 也被初始化为指向 intOne 的引用。但第 15 和 16 行打印了这两个变量的地址，它们相同。

在 C++中，没有提供获取引用本身的地址的方法，因为与指针或其他变量不同，获取引用本身的地址毫无意义。引用在创建时被初始化，此后它一直是目标的别名，即使将地址运算符用于它时。

例如，如果有一个 President 类，你可能使用下面的语句来声明这个类的一个实例：

```
President George_Washington;
```

然后可以声明一个 President 引用，并将其初始化为前面创建的对象：

```
President &Father_Of_Our_Country = George_Washington;
```

在这种情况下，只有一个 President 对象；这两个标识符指的都是同一个类的同一个对象。对 FatherOfOurCauntry 执行的任何操作都将针对 George_Washington。

请注意区分程序清单 9.2 中第 9 行的符号&和第 15 行及 16 行的符号&，前者声明一个名为 rSomeRef 的 int 引用，而后者返回 int 变量 intOne 和引用 rSomeRef 的地址。编译器知道如何根据上下文区分这两种用法。

**注意：**通常，使用引用时不使用地址运算符，引用就相当于目标变量。

### 不能给引用重新赋值

引用不能被重新赋值。即使是经验丰富的 C++程序员也搞不清给引用重新赋值将带来什么后果。引用变量总是其目标的别名。给引用重新赋值相当于给目标重新赋值，程序清单 9.3 说明了这一点。

**程序清单 9.3　给引用赋值**

```
1:  //Listing 9.3 - //Reassigning a reference
2:
3:  #include <iostream>
4:
5:  int main()
6:  {
7:     using namespace std;
8:     int  intOne;
9:     int &rSomeRef = intOne;
10:
11:    intOne = 5;
12:    cout << "intOne:    " << intOne << endl;
13:    cout << "rSomeRef:  " << rSomeRef << endl;
```

```
14:    cout << "&intOne:   " << &intOne << endl;
15:    cout << "&rSomeRef: " << &rSomeRef << endl;
16:
17:    int intTwo = 8;
18:    rSomeRef = intTwo;  // not what you think!
19:    cout << "\nintOne:    " << intOne << endl;
20:    cout << "intTwo:    " << intTwo << endl;
21:    cout << "rSomeRef:  " << rSomeRef << endl;
22:    cout << "&intOne:   " << &intOne << endl;
23:    cout << "&intTwo:   " << &intTwo << endl;
24:    cout << "&rSomeRef: " << &rSomeRef << endl;
25:    return 0;
26: }
```

**输出：**

```
intOne:    5
rSomeRef:  5
&intOne:   0012FEDC
&rSomeRef: 0012FEDC

intOne:    8
intTwo:    8
rSomeRef:  8
&intOne:   0012FEDC
&intTwo:   0012FEE0
&rSomeRef: 0012FEDC
```

**分析：**

第 8 和 9 行声明了一个 int 变量和一个 int 引用。第 11 行将 5 赋给 int 变量，第 12～15 行打它们的值和地址。

第 17 行创建了一个新变量 intTwo，并将它初始化为 8。在第 18 行，程序员试图给 rSomeRef 重新赋值，使其成为变量 intTwo 的别名，但结果并非如此。实际情况是，rSomeRef 仍是变量 intOne 的别名，这种赋值与下面的语句等价：

```
intOne = intTwo;
```

确实如此，第 19～21 行打印 intOne 和 rSomeRef 的值时，结果与 intTwo 相同。实际上，从第 22～24 行打印的地址可知，rSomeRef 仍指向 intOne 而不是 intTwo。

**应该：**

务必使用引用来创建对象的别名。

务必初始化所有的引用。

**不应该：**

不要试图给引用重新赋值。

不要混淆地址运算符和引用运算符。

## 9.3 引用对象

任何对象都可以被引用，包括用户定义的对象。注意，你创建指向对象（而不是类）的引用。例如，下面的语句不能通过编译：

```
int & rIntRef = int;    // wrong
```

必须将引用 rIntRef 初始化为指向某个 int 变量，如：

```
int howBig = 200;
int & rIntRef = howBig;
```

同样，不能将引用初始化为指向 Cat 类：

```
Cat & rCatRef = Cat;  // wrong
```

而必须将 rCatRef 初始化为指向特定的 Cat 类对象：

```
Cat Frisky;
Cat & rCatRef = Frisky;
```

使用指向对象的引用就像使用对象本身一样。要通过对象的引用来访问成员数据和方法，可使用类成员访问运算符（.）；和内置类型的引用一样，对象引用也是对象的别名，程序清单 9.4 说明了这一点。

**程序清单 9.4　指向对象的引用**

```
// Listing 9.4 - References to class objects

#include <iostream>

class SimpleCat
{
  public:
    SimpleCat (int age, int weight);
    ~SimpleCat() {}
    int GetAge() { return itsAge; }
    int GetWeight() { return itsWeight; }
  private:
    int itsAge;
    int itsWeight;
};

SimpleCat::SimpleCat(int age, int weight)
{
   itsAge = age;
   itsWeight = weight;
}

int main()
{
   SimpleCat Frisky(5,8);
   SimpleCat & rCat = Frisky;

   std::cout << "Frisky is: ";
   std::cout << Frisky.GetAge() << " years old." << std::endl;
   std::cout << "And Frisky weighs: ";
   std::cout << rCat.GetWeight() << " pounds." << std::endl;
   return 0;
}
```

**输出：**

```
Frisky is: 5 years old.
```

```
And Frisky weighs 8 pounds.
```

**分析：**

第 25 行将 Frisky 被声明为一个 SimpleCat 对象；第 26 行声明了一个 SimpleCat 引用，并将它初始化为指向 Frisky。第 29 和 31 行分别使用 SimpleCat 对象和 SimpleCat 引用来访问 SimpleCat 类的存取器方法。注意，访问方式是相同的。同样，引用是实际对象的别名。

**引用**

引用是另一个变量的别名。声明引用的方法如下：首先指出类型，然后加上引用运算符（&）和引用名。必须在创建引用的同时对其进行初始化。

**范例 1：**

```
int hisAge;
int &rAge = hisAge;
```

**范例 2：**

```
Cat Boots;
Cat &rCatRef = Boots;
```

## 9.4 空指针和空引用

在指针没有被初始化或删除指针时，应将它们赋为空（0）。但引用并非如此，因为必须在创建的同时将引用始化为它指向的东西。

然而，为让 C++可用于编写能够直接访问硬件的设备驱动程序、嵌入式系统和实时系统，引用特定地址的能力很有用，也必不可少。因此，大多数编译器允许将引用初始化为空或数值，仅当你试图在引用非法时使用对象时，才会导致错误。

然而，在常规编程中利用这种支持仍不是好主意。将使用了空引用的程序移植到其他计算机或编译器时，可能出现难以查找的错误。

## 9.5 按引用传递函数参数

第 5 章介绍了函数的两个缺点：参数按值传递；返回语句只能返回一个值。

按引用将参数传递给函数可克服这两个缺点。在 C++中，按引用传递参数有两种方式：使用指针和使用引用。注意它们的不同：使用指针按引用传递或使用引用按引用传递。

使用指针的语法和使用引用的语法不同，但效果是相同的：不是创建一个作用域为整个函数的拷贝，而是相当于让函数能够访问原始对象。

按引用传递对象让函数能够修改被指向的对象。第 5 章介绍过，传递给函数的参数存储在堆栈中。（使用指针或引用）按引用将参数传递给函数时，原始对象的地址而不是对象本身被存储到堆栈中。实际上，在有些计算机上，地址存储在寄存器中的，在堆栈中不存储任何东西。无论地址是存储在堆栈还是寄存器中，编译器都知道如何获得原始对象，修改是在原始对象而不是其拷贝中进行的。

第 5 章的程序清单 5.5 表明，调用 swap( )函数并不会影响调用函数中的值。程序清单 9.5 再次列出了程序清单 5.5 中的代码，以方便读者参考。

**程序清单 9.5 按值传递**

```
1:  //Listing 9.5 - Demonstrates passing by value
2:  #include <iostream>
```

```
 3:
 4: using namespace std;
 5: void swap(int x, int y);
 6:
 7: int main()
 8: {
 9:    int x = 5, y = 10;
10:
11:    cout << "Main. Before swap, x: " << x << " y: " << y << endl;
12:    swap(x,y);
13:    cout << "Main. After swap, x: " << x << " y: " << y << endl;
14:    return 0;
15: }
16:
17: void swap (int x, int y)
18: {
19:    int temp;
20:
21:    cout << "Swap. Before swap, x: " << x << " y: " << y << endl;
22:
23:    temp = x;
24:    x = y;
25:    y = temp;
26:
27:    cout << "Swap. After swap, x: " << x << " y: " << y << endl;
28: }
```

**输出：**

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

**分析：**

该程序在 main( )中初始化了两个变量，然后将它们传递给函数 swap( )，后者看似是交换这两个变量的值。在 main( )中再次检查这两个变量时，发现它们的值并没有变！

这里的问题是，参数 $x$ 和 $y$ 是按值传递给函数 swap( )的。也就是说，将在函数中创建局部拷贝。函数对局部拷贝进行修改，函数返回时，局部拷贝被丢弃，分配给它们的存储空间被释放。一种更好的方法是按引用传递 $x$ 和 $y$，这样将修改原始变量而不是局部拷贝的值。

在 C++中，解决这个问题的方法有两种：将 swap( )的参数声明为指针，并让它指向原始值；传递指向原始值的引用。

### 9.5.1　使用指针让 swap( )管用

传递指针时，实际上传递的是对象的地址，这样函数便能够操纵存储在该地址处的值。要通过使用指针让函数 swap( )能够交换 $x$ 和 $y$ 的值，应将函数 swap( )声明为接受两个 int 指针参数。然后通过对指针解除引用，访问 $x$ 和 $y$ 的值，并进行交换。程序清单 9.6 演示了这种想法。

**程序清单 9.6　通过使用指针来按引用传递**

```
1: //Listing 9.6 Demonstrates passing by reference
2: #include <iostream>
```

```
 3:
 1: using namespace std;
 5: void swap(int *x, int *y);
 6:
 7: int main()
 8: {
 9:    int x = 5, y = 10;
10:
11:    cout << "Main. Before swap, x: " << x << " y: " << y << endl;
12:    swap(&x,&y);
13:    cout << "Main. After swap, x: " << x << " y: " << y << endl;
14:    return 0;
15: }
16:
17: void swap (int *px, int *py)
18: {
19:    int temp;
20:
21:    cout << "Swap. Before swap, *px: " << *px <<
22:       " *py: " << *py << endl;
23:
24:    temp = *px;
25:    *px = *py;
26:    *py = temp;
27:
28:    cout << "Swap. After swap, *px: " << *px <<
29:       " *py: " << *py << endl;
30:
31: }
```

**输出：**

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, *px: 5 *py: 10
Swap. After swap, *px: 10 *py: 5
Main. After swap, x: 10 y: 5
```

**分析：**

成功了！在第 5 行，函数 swap( )的原型被修改为接受两个 int 指针而不是 int 变量作为参数。第 12 行调用函数 swap( )时，将 x 和 y 的地址作为参数传递给它。之所以知道传递的是地址，是因为这里使用了地址运算符（&）。

第 19 行在函数 swap( )中声明了一个局部变量 temp。temp 不必是指针；它只是用于在函数的生命周期内存储*px 的值（调用函数中变量 x 的值）。函数返回后，不再需要 temp。

第 24 行将 temp 赋给*px。第 25 行*py 的值赋给*px。第 26 行将 temp 的值（即*px 的原始值）赋给*py。这样做的结果是，调用函数中两个变量（它们的地址被传递给函数 swap( )）的值确实被交换了。

### 9.5.2 使用引用来实现 swap( )

上述程序虽然管用，但函数 swap( )的语法比较繁琐，这表现在两个方面：首先，在函数 swap( )中需要对指针进行解除引用，这很容易出错：如果没有对指针解除引用，编译器仍允许将一个整数赋给指针，但指针指向的地址将是错误的；这也不好理解。其次，需要在调用函数中传递变量的地址，这使得 swap( )函数的内部工作原理对用户来说过于明显。

诸如 C++等面向对象编程语言的目标之一是，让用户不必考虑函数的工作原理。使用指针传递参数将本应是被调用函数的责任转移到了调用函数。程序清单 9.7 使用引用重新编写了 swap( )函数。

**程序清单 9.7　使用引用重新编写后的 swap( )**

```
1:  //Listing 9.7 Demonstrates passing by reference
2:  // using references!
3:  #include <iostream>
4:
5:  using namespace std;
6:  void swap(int &x, int &y);
7:
8:  int main()
9:  {
10:   int x = 5, y = 10;
11:
12:   cout << "Main. Before swap, x: " << x << " y: "
13:      << y << endl;
14:
15:   swap(x,y);
16:
17:   cout << "Main. After swap, x: " << x << " y: "
18:      << y << endl;
19:
20:   return 0;
21: }
22:
23: void swap (int &rx, int &ry)
24: {
25:   int temp;
26:
27:   cout << "Swap. Before swap, rx: " << rx << " ry: "
28:      << ry << endl;
29:
30:   temp = rx;
31:   rx = ry;
32:   ry = temp;
33:
34:
35:   cout << "Swap. After swap, rx: " << rx << " ry: "
36:      << ry << endl;
37:
38: }
```

**输出：**

```
Main. Before swap, x:5 y: 10
Swap. Before swap, rx:5 ry:10
Swap. After swap, rx:10 ry:5
Main. After swap, x:10, y:5
```

**分析：**

和使用指针的范例一样，第 10 行声明了两个变量，并在第 12 行打印它们的值。第 15 行调用函数 swap( )，但传递的是 x 和 y，而不是它们的地址。调用函数只是传递这两个变量。

函数 swap( )被调用时，程序跳到第 23 行执行，在这里参数为引用。第 27 行打印变量的值，但不需要使用任何特殊的运算符。它们都是原变量的别名，可以像原变量一样使用。

第 30～32 行交换参数的值，然后在第 35 行进行打印。程序跳回到调用函数中的第 17 行执行：打印 main( )中两个变量的值。由于函数 swap( )的参数被声明为引用，因此 main( )中的变量是按引用传递的，所以在 main( )中看到的也是修改后的值。

从这个程序清单可知，引用在使用方面与常规变量一样方便和容易，同时具备指针的强大功能和按引用传递的能力。

## 9.6 理解函数头和原型

程序清单 9.6 演示了如何使用指针来编写 swap( )，而程序清单 9.7 演示了如何使用引用来编写该函数。将引用作为参数的函数使用起来更容易，代码也更容易理解，但调用函数如何知道是按引用还是按值传递参数呢？作为函数 swap( )的客户（用户），程序员必须确信函数 swap( )将交换参数的值。

这是函数原型的另一种用途。通过查看原型（通常，所有函数原型都被存储在一个头文件中）中声明的参数，程序员知道函数 swap( )按引用传递参数，将正确地交换参数的值。

如果函数 swap( )是类的成员函数，类声明（也位于头文件中）也将提供这些信息。

在 C++中，类和函数的客户可通过头文件获悉所需的信息，头文件中包含类和函数的接口；实际实现是对客户隐藏的。这让程序员可以将注意力集中在要解决的问题上，并在使用类和函数时不必关心它们的工作原理。

Colonel John Roebling 设计 Brookly 大桥时，需要详细考虑如何倾倒混凝土和制造桥的钢筋等问题。他亲自参与了制造材料的机械和化学流程。然而在今天，工程师们对建筑材料有深入地了解，从而可以更有效地使用他们的时间，而不必关心制造商如何生产这些材料。

C++的目标是，让程序员能够依赖于其对类和函数的了解，而不必关心它们的内部工作原理。可以将这些“零部件”组装成一个程序，就像可以将钢筋、管道、支架及其他构件组装成大楼和桥梁一样。

正如工程师通过查看管道规格表，以了解它的承载能力、容量、适应尺寸等一样，C++程序员通过阅读函数或类的声明，了解它们能够提供什么服务、接受什么参数以及返回什么值。

## 9.7 返回多个值

正如本书前面讨论过的，函数只能返回一个值。如果需要从函数那里获得两个值，该如何办呢？解决这种问题的方法之一是，按引用将两个对象传递给函数。然后，函数便可以将正确的值赋给这两个对象。由于按引用传递让函数能够修改原始对象，这相当于让函数能够返回两组信息，这种方法不要求函数返回值，可以将返回值保留用于报告错误。

同样，这可以使用引用或指针来实现。程序清单 9.8 演示一个返回 3 个值的函数：其中两个为指针参数，一个为函数的返回值。

**程序清单 9.8 通过指针来返回值**

```
1: //Listing 9.8 - Returning multiple values from a function
2:
3: #include <iostream>
4:
5: using namespace std;
6: short Factor(int n, int* pSquared, int* pCubed);
7:
```

```
 8: int main()
 9: {
10:    int number, squared, cubed;
11:    short error;
12:
13:    cout << "Enter a number (0 - 20): ";
14:    cin >> number;
15:
16:    error = Factor(number, &squared, &cubed);
17:
18:    if (!error)
19:    {
20:       cout << "number: " << number << endl;
21:       cout << "square: " << squared << endl;
22:       cout << "cubed: "  << cubed  << endl;
23:    }
24:    else
25:       cout << "Error encountered!!" << endl;
26:    return 0;
27: }
28:
29: short Factor(int n, int *pSquared, int *pCubed)
30: {
31:    short Value = 0;
32:    if (n > 20)
33:       Value = 1;
34:    else
35:    {
36:       *pSquared = n*n;
37:       *pCubed = n*n*n;
38:       Value = 0;
39:    }
40:    return Value;
41: }
```

**输出：**

```
Enter a number (0-20): 3
number: 3
square: 9
cubed: 27
```

**分析：**

第 10 行将 number、squared 和 cubed 定义为 int 变量。第 14 行将用户输入的值赋给 number。第 16 行将 number 以及 squared 和 cubed 的地址传递给函数 Factor( )。

在第 32 行，函数 Factor( )检查第一个参数，这个参数是按值传递的。如果它大于 20（该函数能够处理的最大值），则将返回值 Value 设置为一个简单的错误值。函数 Factor( )的返回值用于指示错误信息，如果一切正常，将其设置为 0；在第 40 行，函数返回这个值。

实际需要的值（number 的平方和立方）并不是通过返回机制返回的，而是通过修改传递给函数的指针指向的值来返回的。

第 36 和 37 行设置指针指向的值，通过间接访问（通过将解除引用运算符*用于指针）将这些值赋给原来的变量。第 38 行将表示成功的值赋给变量 Value，第 40 行将其返回。

**提示：**由于按引用传递时不能控制对对象属性和方法的访问，因此应在让函数能完成其工作的情况下，将尽可能少的访问权限提供给函数。这有助于确保函数使用起来更安全，也更易于理解。

### 按引用返回值

虽然程序清单 9.8 管用，但如果使用引用而不是指针，程序将更容易理解和维护。程序清单 9.9 使用引用重新编写了该程序。

程序清单 9.9 还做了另一方面的改进：定义了一个枚举类型，使返回值更容易理解。该程序不是返回 0 或 1，而是通过使用枚举返回 SUCCESS 或 FAILURE。

**程序清单 9.9　使用引用重写程序**

```
1: //Listing 9.9
2: // Returning multiple values from a function
3: // using references
4: #include <iostream>
5:
6: using namespace std;
7:
8: enum ERR_CODE { SUCCESS, ERROR };
9:
10: ERR_CODE Factor(int, int&, int&);
11:
12: int main()
13: {
14:    int number, squared, cubed;
15:    ERR_CODE result;
16:
17:    cout << "Enter a number (0 - 20): ";
18:    cin >> number;
19:
20:    result = Factor(number, squared, cubed);
21:
22:    if (result == SUCCESS)
23:    {
24:       cout << "number: " << number << endl;
25:       cout << "square: " << squared << endl;
26:       cout << "cubed: "  << cubed  << endl;
27:    }
28:    else
29:       cout << "Error encountered!!" << endl;
30:    return 0;
31: }
32:
33: ERR_CODE Factor(int n, int &rSquared, int &rCubed)
34: {
35:    if (n > 20)
36:       return ERROR;   // simple error code
37:    else
38:    {
39:       rSquared = n*n;
40:       rCubed = n*n*n;
```

```
41:       return SUCCESS;
42:    }
43: }
```

**输出：**

```
Enter a number (0 - 20): 3
number: 3
square: 9
cubed: 27
```

**分析：**

程序清单 9.9 与程序清单 9.8 相比有两点不同：枚举 ERR_CODE 使得第 36 行和第 41 行的错误报告以及第 22 行的错误处理更容易理解。

然而，更大的变化是，函数 Factor( )现在被声明为接受引用而不是将指针作为参数。这使得对这些参数的操纵更简单，更容易理解。

## 9.8　按引用传递以提高效率

按值将对象传递给函数时，都将创建该对象的一个拷贝；而按值从函数返回一个对象时，将创建另一个拷贝。

第 5 章介绍过，这些对象被复制到堆栈中。这样做既费时又占用内存。对于小型对象，如内置的整数值，这样的开销是微不足道的。

然而，对于用户定义的大型对象，开销大得多。用户定义的对象在堆栈中占据的空间是其所有的成员变量所占空间的总和，而这些成员变量本身又可能是用户创建的对象，通过在堆栈中复制来传递如此庞大的结构，无论是在性能方面还是在内存占用方面都是非常昂贵的。

还有另一种开销。每当创建这些临时拷贝时，都要调用一个特殊的构造函数：复制构造函数。有关复制构造函数的工作原理以及如何创建自己的复制构造函数，将在下一章介绍；就现在而言，读者只需知道每次在堆栈中创建临时拷贝时都将调用复制构造函数即可。

函数返回时，临时对象将被销毁，这将调用对象的构析函数。如果函数按值返回对象，将需要创建和销毁该对象的一个拷贝。

对于大型的对象，调用构造函数和析构函数在速度和内存方面的开销都可能很大。为说明这一点，程序清单 9.10 创建了一个简单的用户定义的对象：SimpleCat。实际的对象可能更大，开销也可能更高，但使用它足以说明复制调用构造函数和析构函数的调用频率。

**程序清单 9.10　按引用传递对象**

```
1: //Listing 9.10 - Passing pointers to objects
2:
3: #include <iostream>
4:
5: using namespace std;
6: class SimpleCat
7: {
8:   public:
9:     SimpleCat ();           // constructor
10:    SimpleCat(SimpleCat&);   // copy constructor
11:    ~SimpleCat();           // destructor
12: };
```

```
13:
14: SimpleCat::SimpleCat()
15: {
16:    cout << "Simple Cat Constructor..." << endl;
17: }
18:
19: SimpleCat::SimpleCat(SimpleCat&)
20: {
21:    cout << "Simple Cat Copy Constructor..." << endl;
22: }
23:
24: SimpleCat::~SimpleCat()
25: {
26:    cout << "Simple Cat Destructor..." << endl;
27: }
28:
29: SimpleCat FunctionOne (SimpleCat theCat);
30: SimpleCat* FunctionTwo (SimpleCat *theCat);
31:
32: int main()
33: {
34:    cout << "Making a cat..." << endl;
35:    SimpleCat Frisky;
36:    cout << "Calling FunctionOne..." << endl;
37:    FunctionOne(Frisky);
38:    cout << "Calling FunctionTwo..." << endl;
39:    FunctionTwo(&Frisky);
40:    return 0;
41: }
42:
43: // FunctionOne, passes by value
44: SimpleCat FunctionOne(SimpleCat theCat)
45: {
46:    cout << "Function One. Returning... " << endl;
47:    return theCat;
48: }
49:
50: // functionTwo, passes by reference
51: SimpleCat* FunctionTwo (SimpleCat *theCat)
52: {
53:    cout << "Function Two. Returning... " << endl;
54:    return theCat;
55: }
```

**输出：**

```
Making a cat...
Simple Cat Constructor...
Calling FunctionOne...
Simple Cat Copy Constructor...
Function One. Returning...
Simple Cat Copy Constructor...
Simple Cat Destructor...
```

```
Simple Cat Destructor...
Calling FunctionTwo...
Function Two. Returning...
Simple Cat Destructor...
```

**分析：**

程序清单 9.10 创建了一个 SimpleCat 对象，然后调用两个函数。第一个函数按值接受一个 Cat 对象，然后按值返回它。第二个函数接受一个对象指针而不是对象本身作为参数，并返回一个指向对象的指针。

第 6~12 行声明了一个非常简单的 SimpleCat 类。构造函数、复制构造函数和析构函数打印一条消息，以便它们被调用时读者能够知道。

在第 34 行，main( )函数打印一条消息，即输出中的第一行。第 35 行实例化一个 SimpleCat 对象。这将导致构造函数被调用，输出中的第 2 行就是构造函数打印的消息。

在第 36 行，main( )函数指出将调用 FunctionOne，这是输出中的第 3 行。由于调用 FunctionOne( )时按值传递了 SimpleCat 对象，因此将在堆栈中创建该 SimpleCat 对象的一个拷贝，将其作为被调用函数的局部对象。这导致复制构造函数被调用，生成输出中的第 4 行。

然后，程序跳到被调用函数中的第 46 行执行：打印一条消息，这是输出中的第 5 行。然后函数返回，并按值返回 SimpleCat 对象，这将创建该对象的另一个拷贝，导致复制构造函数被调用，生成输出中的第 6 行。

函数 FunctionOne( )的返回值没有被赋给任何对象，因此为返回而创建的临时对象被丢弃，这导致析构函数被调用，生成输出中的第 7 行。由于 FunctionOne( )已经结束，其局部拷贝不再在作用域中，因此将被销毁，导致析构函数被调用，生成输出中的第 8 行。

程序返回到 main( )继续执行，并调用 FunctionTwo( )函数，但按引用传递参数。这不会创建对象拷贝，因此没有输出。FunctionTwo( )打印位于输出中第 10 行的消息，然后按引用返回 SimpleCat 对象，因此不会导致调用构造函数和析构函数。

最后，程序结束，Frisky 不再在作用域中，导致最后一次调用析构函数并打印输出中的最后一行。

调用 FunctionOne( )时，由于按值传递 Frisky，导致复制构造函数和析构函数被调用两次；调用 FunctionTwo( )时没有导致复制构造函数和析构函数被调用。

### 9.8.1　传递 const 指针

虽然给函数 FunctionTwo( )传递指针的效率更高，但这样做也是危险的。函数 FunctionTwo( )并不打算对传递给它的 SimpleCat 对象进行修改，但仍获得了该对象的地址。这就使原始对象暴露在被修改的危险之中，失去了按值传递提供的保护。

按值传递就像将作品的照片而不是实物交给博物馆；在这张照片上做任何改动都不会损害原物。按引用传递就像将家里的地址告诉博物馆，并邀请客人来家中观看实物。

解决这个问题的方法是，传递一个指向 cosnt SimpleCat 对象的指针。这样做可防止对 SimpleCat 对象调用任何非 const 方法，从而防止对象被改变。

传递 const 引用使客人能够看到原物，但不允许做任何修改。程序清单 9.11 说明了这种思想。

**程序清单 9.11　传递指向 const 对象的指针**

```
1:  //Listing 9.11 - Passing pointers to objects
2:
3:  #include <iostream>
4:
5:  using namespace std;
6:  class SimpleCat
7:  {
8:    public:
9:      SimpleCat();
```

```
10:     SimpleCat(SimpleCat&);
11:     ~SimpleCat();
12:
13:     int GetAge() const { return itsAge; }
14:     void SetAge(int age) { itsAge = age; }
15:
16:   private:
17:     int itsAge;
18: };
19:
20: SimpleCat::SimpleCat()
21: {
22:    cout << "Simple Cat Constructor..." << endl;
23:    itsAge = 1;
24: }
25:
26: SimpleCat::SimpleCat(SimpleCat&)
27: {
28:    cout << "Simple Cat Copy Constructor..." << endl;
29: }
30:
31: SimpleCat::~SimpleCat()
32: {
33:    cout << "Simple Cat Destructor..." << endl;
34: }
35:
36: const SimpleCat * const FunctionTwo
37:    (const SimpleCat * const theCat);
38:
39: int main()
40: {
41:    cout << "Making a cat..." << endl;
42:    SimpleCat Frisky;
43:    cout << "Frisky is " ;
44:    cout << Frisky.GetAge();
45:    cout << " years old" << endl;
46:    int age = 5;
47:    Frisky.SetAge(age);
48:    cout << "Frisky is " ;
49:    cout << Frisky.GetAge();
50:    cout << " years old" << endl;
51:    cout << "Calling FunctionTwo..." << endl;
52:    FunctionTwo(&Frisky);
53:    cout << "Frisky is " ;
54:    cout << Frisky.GetAge();
55:    cout << " years old" << endl;
56:    return 0;
57: }
58:
59: // functionTwo, passes a const pointer
60: const SimpleCat * const FunctionTwo
61:    (const SimpleCat * const theCat)
```

```
62: {
63:    cout << "Function Two. Returning..." << endl;
64:    cout << "Frisky is now " << theCat->GetAge();
65:    cout << " years old " << endl;
66:    // theCat->SetAge(8);  const!
67:    return theCat;
68: }
```

**输出：**

```
Making a cat...
Simple Cat constructor...
Frisky is 1 years old
Frisky is 5 years old
Calling FunctionTwo...
FunctionTwo. Returning...
Frisky is now 5 years old
Frisky is 5 years old
Simple Cat Destructor...
```

**分析：**

在 SimpleCat 类中添加了两个存取器函数：第 13 行的 GetAge( )和第 14 行的 SetAge( )，其中 GetAge( )是一个 const 方法，而 SetAge( )不是。还在第 17 行添加了成员变量 itsAge。

构造函数、复制构造函数和析构函数仍被定义为打印相应的消息。然而，复制构造函数永远不会被调用，因为对象是按引用传递的，不会创建对象拷贝。第 42 行创建一个对象，从第 43 行开始打印它的默认年龄。

第 47 行使用存取器函数 SetAge( )设置 itsAge 的值，第 48 行打印结果。在这个程序中，没有使用 FunctionOne( )，但调用了 FunctionTwo( )。FunctionTwo( )有细微的变化，第 36 行将其参数和返回值声明为指向 const 对象的 const 指针。

由于参数和返回值仍是按引用传递的，因此不需要创建对象拷贝，也就不会调用复制构造函数。然后，由于在函数 FunctionTwo( )中，被指向的对象为 const，因此不能调用非 const 方法 SetAge( )。如果不将调用 SetAge( )的第 66 行注释掉，程序将不能通过编译。

注意，在 main( )函数中创建的对象并不是 const 的，因此 Frisky 能够调用函数 SetAge( )。这个非 const 对象的地址被传递给函数 FunctionTwo( )，但由于该函数的原型将指针声明为指向 const 对象的 const 指针，因此该对象被视为 const 的。

### 9.8.2　用引用代替指针

程序清单 9.11 解决了需要创建拷贝的问题，从而减少了对复制构造函数和析构函数的调用。它使用指向 const 对象的 const 指针，因此也解决了在函数中可能修改对象的问题。然而，这有些繁琐，因为传递给函数的是指向对象的指针。

由于对象不可能为空，如果传递引用而不是指针，则在函数中处理起来将更方便。程序清单 9.12 说明了这一点。

**程序清单 9.12　传递指向对象的引用**

```
1: //Listing 9.12 - Passing references to objects
2:
3: #include <iostream>
4:
5: using namespace std;
6: class SimpleCat
```

```
 7: {
 8:   public:
 9:     SimpleCat();
10:     SimpleCat(SimpleCat&);
11:     ~SimpleCat();
12:
13:     int GetAge() const { return itsAge; }
14:     void SetAge(int age) { itsAge = age; }
15:
16:   private:
17:     int itsAge;
18: };
19:
20: SimpleCat::SimpleCat()
21: {
22:   cout << "Simple Cat Constructor..." << endl;
23:   itsAge = 1;
24: }
25:
26: SimpleCat::SimpleCat(SimpleCat&)
27: {
28:   cout << "Simple Cat Copy Constructor..." << endl;
29: }
30:
31: SimpleCat::~SimpleCat()
32: {
33:   cout << "Simple Cat Destructor..." << endl;
34: }
35:
36: const    SimpleCat & FunctionTwo (const SimpleCat & theCat);
37:
38: int main()
39: {
40:   cout << "Making a cat..." << endl;
41:   SimpleCat Frisky;
42:   cout << "Frisky is " << Frisky.GetAge() << " years old" << endl;
43:   int age = 5;
44:   Frisky.SetAge(age);
45:   cout << "Frisky is " << Frisky.GetAge() << " years old" << endl;
46:   cout << "Calling FunctionTwo..." << endl;
47:   FunctionTwo(Frisky);
48:   cout << "Frisky is " << Frisky.GetAge() << " years old" << endl;
49:   return 0;
50: }
51:
52: // functionTwo, passes a ref to a const object
53: const SimpleCat & FunctionTwo (const SimpleCat & theCat)
54: {
55:   cout << "Function Two. Returning..." << endl;
56:   cout << "Frisky is now " << theCat.GetAge();
57:   cout << " years old " << endl;
58:   // theCat.SetAge(8);  const!
```

```
59:    return theCat;
60: }
```

**输出：**

```
Making a cat...
Simple Cat constructor...
Frisky is 1 years old
Frisky is 5 years old
Calling FunctionTwo...
FunctionTwo. Returning...
Frisky is now 5 years old
Frisky is 5 years old
Simple Cat Destructor...
```

**分析：**

该程序的输出与程序清单 9.11 相同。惟一明显的变化是，现在函数 FunctionTwo( )接受一个指向 const 对象的引用作为参数，并返回一个这样的引用。同样，使用引用比使用指针更简单，但在内存节省和效率方面与指针相同，同时可以使用 const 来提供安全。

**const 引用**

C++程序员对“指向 SimpleCat 对象的 const 引用”和“指向 const SimpleCat 对象的引用”通常是不加以区分的。由于不能给引用重新赋值，使之指向另一个对象，因此它们总是 const 的。如果将关键字 const 用于引用，将使被指向的对象为 const 的。

## 9.9　何时使用引用和指针

相对于指针来说，经验丰富的 C++程序员更喜欢使用引用。引用不但更清晰，更容易使用，而且能够更好地隐藏信息，如前面的例子所示。

然而，引用不能被重新赋值。如果需要首先指向一个对象，然后指向另一个，则必须使用指针。引用不能为空，因此如果对象可能为空，则绝对不能使用引用，而必须使用指针。

第二种情况的一个例子是使用 new 运算符来创建对象。如果 new 不能在自由存储区中分配内存，将返回一个空指针。由于引用不能为空，因此除非已经确定内存不为空，否则不要将引用初始化为指向该内存。下面的例子演示了如何处理这种情况：

```
int *pInt = new int;
if (pInt != NULL)
int &rInt = *pInt;
```

在这个例子中，声明了一个 int 指针 pInt，并将其初始化为指向 new 运算符返回的内存。然后测试 pInt 的地址，如果不为空，则对 pInt 解除引用（对 int 指针解除引用的结果是 int 对象），并将 rInt 初始化为指向这个对象。这样，rInt 就成了运算符 new 返回的 int 对象的别名。

**应该：**

尽可能按引用传递参数。

尽可能使用 const 来保护引用和指针。

**不应该：**

在可以使用引用时不要使用指针。

不要试图给引用重新赋值，使之指向另一个变量，这是不可能的。

## 9.10 混合使用引用和指针

在同一个函数参数列表中同时声明指针和引用以及按值传递对象是完全合法的。下面是一个这样的例子：

```
Cat * SomeFunction (Person &theOwner, House *theHouse, int age);
```

该声明指出，函数 SomeFunction( )接受 3 个参数。第 1 个是指向 Person 对象的引用，第 2 个是指向 House 对象的指针，第 3 个是一个 int 变量。该函数返回了一个指向 Cat 对象的指针。

关于在声明这些变量时，将引用运算符（&）和间接访问运算符（*）放在什么地方存在较大的争论。以下声明引用的方式都是合法的：

```
1: Cat & rFrisky;
2: Cat & rFrisky;
3: Cat & rFrisky;
```

空白被完全忽略了，因此任何有空格的地方可以放置任意数目的空格、制表符和换行。

撇开自由表达的问题不谈，哪种方式最好呢？以下是认为各种方式最好的理由。

第一种方式最好的理由是，rFrisky 是一个变量，其名称是 rFrisky 的变量，类型为指向 Cat 对象的引用。因此这种观点认为，&应和类型放在一起。

反对者认为，类型应为 Cat。&是声明的一部分，声明包括变量名和&。更重要的是，将&和 Cat 放在一起导致下面这样的错误：

```
Cat& rFrisky, rBoots;
```

如果不注意，可能认为 rFrisky 和 rBoots 都是指向 Cat 对象的引用，但是你错了。这行代码的实际意思是，rFrisky 是一个 Cat 引用，而 rBoots 不是引用（虽然名称中包含前缀 r），而是一个 Cat 对象。这行代码应这样书写：

```
Cat &rFrisky, rBoots;
```

对于这种写法，反对者认为，就不应像上面这样将引用声明和变量声明混在一起。正确的方式如下：

```
Cat& rFrisky;
Cat boots;
```

最后，很多程序员选择置身于这种争论之外，将&放在类型和名称中间，如第二种方式所示。

当然，这里对引用运算符（&）的所有讨论也适用于间接运算符（*）。重要的在于，对于哪种方式是正确的存在不同看法。选择一种适合于自己的风格，并在同一个程序保持一致；代码清晰仍然是我们的目标。

**注意**：很多程序员喜欢使用下述声明引用和指针的方式：

- 将&和*放在中间，两边各加一个空格。
- 决不在同一行中同时声明引用、指针和变量。

## 9.11 返回指向不在作用域中的对象的引用

C++程序员学习按引用传递后，常常会过度迷恋，进而滥用它。别忘了，引用是其他对象的别名。将引用传入或传出函数时，务必这样自问：引用指向的对象是什么？每次使用它时它是否存在？

程序清单 9.13 说明了返回指向不再存在的对象的引用的危险性。

**程序清单 9.13 返回指向不存在的对象的引用**

```
1: // Listing 9.13
```

```
2: // Returning a reference to an object
3: // which no longer exists
4:
5: #include <iostream>
6:
7: class SimpleCat
8: {
9:   public:
10:    SimpleCat (int age, int weight);
11:    ~SimpleCat() {}
12:    int GetAge() { return itsAge; }
13:    int GetWeight() { return itsWeight; }
14:  private:
15:    int itsAge;
16:    int itsWeight;
17: };
18:
19: SimpleCat::SimpleCat(int age, int weight)
20: {
21:   itsAge = age;
22:   itsWeight = weight;
23: }
24:
25: SimpleCat &TheFunction();
26:
27: int main()
28: {
29:   SimpleCat &rCat = TheFunction();
30:   int age = rCat.GetAge();
31:   std::cout << "rCat is " << age << " years old!" << std::endl;
32:   return 0;
33: }
34:
35: SimpleCat &TheFunction()
36: {
37:   SimpleCat Frisky(5,9);
38:   return Frisky;
39: }
```

**输出：**

```
Compile error: Attempting to return a reference to a local object!
```

**警告：**这个程序在 Borland 编译器中不能通过编译，但在 Microsoft C++编译器中能够通过编译。然而，需要指出的是，这是一种糟糕的编码习惯。

**分析：**

第 7～17 行声明了 SimpleCat 类。第 29 行声明了一个 SimpleCat 引用，并将其初始化为指向函数 TheFuction()的返回值，在第 25 行，该函数被声明为返回一个 SimpleCat 引用。

第 35～39 行是 TheFuction()的函数体，其中声明了一个局部 SimpleCat 对象，并初始化了该对象的年龄和重量。然后，在第 38 行，该函数返回指向该局部对象的引用。有些编译器很聪明，能够发现这种错误，进而不允许用户运行该程序；其他编译器允许运行该程序，但结果是不可预料的。

函数 TheFunction( )返回时，局部对象 Frisky 将被销毁。函数返回的引用是一个并不存在的对象的别名，这很糟糕。

### 返回指向堆中对象的引用

你可能试图让函数 TheFuction( )在堆中创建对象 Frisky 来解决程序清单 9.13 中的问题。这样，当函数返回时 Frisky 仍存在。

这种解决方法存在的问题是：使用完 Frisky 后，如何释放分配给它的内存？程序清单 9.14 说明了这种问题。

**程序清单 9.14　内存泄漏**

```
// Listing 9.14 - Resolving memory leaks

#include <iostream>

class SimpleCat
{
  public:
    SimpleCat (int age, int weight);
    ~SimpleCat() {}
    int GetAge() { return itsAge; }
    int GetWeight() { return itsWeight; }

  private:
    int itsAge;
    int itsWeight;
};

SimpleCat::SimpleCat(int age, int weight)
{
   itsAge = age;
   itsWeight = weight;
}

SimpleCat & TheFunction();

int main()
{
   SimpleCat & rCat = TheFunction();
   int age = rCat.GetAge();
   std::cout << "rCat is " << age << " years old!" << std::endl;
   std::cout << "&rCat: " << &rCat << std::endl;
   // How do you get rid of that memory?
   SimpleCat * pCat = &rCat;
   delete pCat;
   // Uh oh, rCat now refers to ??
   return 0;
}

SimpleCat &TheFunction()
{
   SimpleCat * pFrisky = new SimpleCat(5,9);
```

```
42:    std::cout << "pFrisky: " << pFrisky << std::endl;
43:    return *pFrisky;
44: }
```

输出：

```
pFrisky: 0x00431C60
rCat is 5 years old!
&rCat: 0x0043_C6C0
```

警告：该程序能够通过编译和链接，且看起来运行也正常，但它是一个将要爆炸的定时炸弹。

分析：

在第 39～44 行，对函数 TheFuction( )进行了修改，使之它不再返回一个指向局部变量的引用。第 41 行从自由存储区中分配内存，并将其地址赋给一个指针，然后打印存储在该指针中的地址，并对指针解除引用，从而按引用返回该 SimpleCat 对象。

第 28 行将函数 TheFuction( )的返回值赋给一个 SimpleCat 引用，然后使用该引用来获得对象的年龄，并在第 30 行打印它。

为证明函数 main( )中声明的引用指向的是函数 TheFuction( )在自由存储区中创建的对象，将地址运算符用于 rCat。毫无疑问，这将获得其指向的对象的地址，且与自由存储区中对象的地址相同。

到目前为止，一切顺利。但如何释放被占用的内存呢？对于引用是不能使用 delete 运算符的。一种聪明的解决方案是，创建另一个指针，并将其初始化为从 Cat 获得的地址。这样确实可以释放内存，避免了内存泄漏；但存在一个小问题：在第 34 行之后，rCat 将指向什么呢？正如前面指出的，引用必须始终是一个实际对象的别名；如果指向一个空对象（就像现在这样），程序将是非法的。

注意：使用指向空对象的引用的程序也许能够通过编译，但它是非法的，结果是不可预料的，对于这一点如何强调都不过分。

对于这种问题，存在 3 种解决方案。第 1 种是在第 28 行声明一个 SimpleCat 对象，然后从函数 TheFuction( )按值返回该对象；第 2 种是仍在函数 TheFuction( )中声明一个位于自由存储区中的 SimpleCat 对象，但让函数 TheFuction( )返回一个指向该对象的指针。这样，使用完该对象后，调用函数可以将该指针删除。

第 3 种可行也是正确的解决方案是，在调用函数中声明对象，然后按引用将它传递给函数 TheFuction( )。

## 9.12　指针归谁所有

程序在自由存储区中分配内存时，将返回一个指针。必须让一个指针指向该内存，因为如果没有这样的指针，内存将无法被释放，导致内存泄漏。

在函数间传递该内存块时，某个函数拥有这个指针。通常，按引用来传递该内存块中的数据，分配内存的函数负责释放它。但这只是一般性规则，并非金科玉律。

然而，在一个函数中分配内存，而在另一个函数中释放它是非常危险的。如果对谁拥有指针不明确，可能导致两个问题之一：要么忘记删除指针，要么将指针删除两次。这两种问题都将给程序带来严重的后果。编写函数时，使之释放自己分配内存的将更安全。

如果需要在一个函数中分配内存，并将它传递给调用函数，应考虑修改接口：让调用函数分配内存，然后按引用将其传递给被调用的函数。这将把所有的内存管理工作移到程序外，由释放内存的函数负责分配。

应该：

在必须按值传递参数时务必这样做。

在必须按值返回时务必这样做。

不应该：

如果被引用的对象可能位于作用域外，不要按引用传递。

不要失去内存是何时在什么地方分配的线索，以确保内存得到释放。

## 9.13 小　　结

本章介绍引用并将其同指针进行了比较。必须对引用进行初始化，使之指向一个现有的对象；同时不能给引用重新赋值，使之指向其他对象。对引用的任何操作实际上针对的都是引用的目标对象；有关这一点的证据是，将地址运算符用于引用时，返回的是它指向的目标对象的地址。

与按值传递相比，按引用传递对象的效率更高。另外，按引用传递让被调用的函数能够修改调用函数传入的实参。

函数的参数和返回值可以按引用传递，这可以使用指针或引用来实现。

读者学习了如何使用指向 const 对象的指针和 const 引用在函数之间安全地传递值，同时获得按引用传递的高效率。

## 9.14 问 与 答

**问：**既然指针具备引用的所有功能，为什么还要使用引用？

**答：**引用更容易使用和理解。间接被隐藏，不需要重复地对变量解除引用。

**问：**既然引用更容易，为什么还要使用指针呢？

**答：**引用不能为空，也不能被重新赋值。指针提供了更大的灵活性，但使用起来更难些。

**问：**为什么要从函数按值返回？

**答：**如果被返回的对象是局部的，必须按值返回，否则将返回指向不存在的对象的引用。

**问：**鉴于按引用返回的危险性，为什么不总是按值返回呢？

**答：**按引用返回的效率高得多。这样做不仅可以节省内存，程序的运行速度也更快。

## 9.15 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

### 9.15.1 测验

1. 引用和指针之间有什么区别？
2. 什么时候必须使用指针而不是引用？
3. 如果没有足够的内存来创建新对象，new 将返回什么？
4. 什么是 const 引用？
5. 按引用传递和传递引用之间有何区别？
6. 下面哪种声明引用的方式是正确的？

   a. `int& myRef = myInt;`

   b. `int & myRef = myInt;`

   c. `int &myRef = myInt;`

### 9.15.2　练习

1. 编写一个程序，声明一个 int、一个 int 引用和一个 int 指针，然后使用指针和引用来操纵 int 变量的值。

2. 编写一个程序，声明一个指向 const int 变量的 const 指针。将该指针初始化为指向 int 变量 varOne，并将 6 赋给 varOne。使用该指针将 7 赋给 varOne。创建另一个 int 变量 varTwo，给指针重新赋值，使之指向 varTwo。暂不要编译这个程序。

3. 编译练习 2 中的程序。将出现什么错误？什么警告？

4. 编写一个生成迷失（stray）指针的程序。

5. 修复练习 4 中程序的问题。

6. 编写一个导致内存泄漏的程序。

7. 修复练习 6 中程序的问题。

8. 查错：下面的程序有什么错误？

```
#include <iostream>
using namespace std;
class CAT
{
  public:
    CAT(int age) { itsAge = age; }
    ~CAT(){}
    int GetAge() const { return itsAge;}
  private:
    int itsAge;
};

CAT & MakeCat(int age);
int main()
{
  int age = 7;
  CAT Boots = MakeCat(age);
  cout << "Boots is " << Boots.GetAge()
       << " years old" << endl;
 return 0;
}

CAT & MakeCat(int age)
{
  CAT * pCat = new CAT(age);
  return *pCat;
}
```

9. 修复练习 8 中程序的问题。