策略基础。

图 7-6 中的代码通过让函数 isEven 和 isOdd 获取的参数类型为 unsigned 的方式确保了实现条件，C++ 使用 unsigned 类型表示不小于 0 的整数。

```
/*
 * Function: isOdd
 * Usage: if (isOdd(n)) . . .
 * --------------------------
 * Returns true if the unsigned number n is odd.  A number is odd
 * if it is not even.
 */

bool isOdd(unsigned int n) {
   return !isEven(n);
}

/*
 * Function: isEven
 * Usage: if (isEven(n)) . . .
 * ---------------------------
 * Returns true if the unsigned number n is even.  A number is even
 * either (1) if it is zero or (2) if its predecessor is odd.
 */

bool isEven(unsigned int n) {
   if (n == 0) {
      return true;
   } else {
      return isOdd(n - 1);
   }
}
```

图 7-6 isEven 和 isOdd 的间接递归定义

7.7 递归地思考

对于大多数人来说，递归不是一个易于掌握的概念。学习有效地使用递归需要大量的练习，并且它迫使你用全新的方法来解决问题。成功的关键在于形成正确的习惯——学习如何递归地进行思考。本章其余部分将帮助你实现这个目标。

7.7.1 保持全局的观点

当你学习编程时，我认为牢记整体论与简化论的哲学理念将会对你有很大的帮助。简单地说，**简化论**（reductionism）就是这样一种理念，它仅仅通过理解构成对象的某一部分就可理解整个对象。与之对立的就是**整体论**（holism），它认为整体总是比构成它的部分总和更为重要。当你学习编程时，它帮助你能够交错这两种视角，有时候将程序的行为当作一个整体，而在其他时刻，需要探究执行的细节。然而，当你试图理解递归时，这种平衡似乎被改变了。递归地思考需要你从整体的角度来思考。在递归领域，简化论是理解的敌人，它总是妨碍你的理解。

为了保持全局的视角，你必须习惯于采用本章 7.2 节所介绍的递归的稳步跳跃的理念。无论你编写一个递归程序或者尝试理解一个递归程序的行为，你都必须找到在一个单独的递归调用中那些可以被忽略的细节。只要你选择了正确的分解，确定了适当的简单情况，并且正确地实现了你的策略，这些递归调用将会起作用，你不必考虑它们的具体实现细节。

遗憾的是，除非你有大量处理递归函数的经验，否则有效地运用递归的稳步跳跃这一概念并非易事。采用递归需要暂缓你的怀疑，并且对程序的正确性做出假设，它完全违反你以往的经验。毕竟，当你编写一个程序时，这是很有可能的（即使你是一个有经验的程序员），

你的程序第一次也不会正确地工作。事实上，很可能是由于你选择了错误的分解，搞乱了简单情况的定义，或者在你试图实现你的策略时莫名其妙的将事情弄得一团糟。如果你已经做了这些事中的任何一个，那么你的递归调用将不会工作。 337～338

当事情出错时（由于它们不可避免地会发生），你必须牢记在合适的地方寻找错误。问题是你在递归实现中某个地方发生了错误，不是递归机制本身有什么问题。如果递归程序有问题，你应该能够通过查找单个的递归层次来找到其中的错误。向下查找递归调用的其他层次不会有帮助。如果简单情况能工作，并且递归分解是正确的，那么子调用会正常工作。否则，问题一定出现在递归分解的公式中。

7.7.2 避免常见的错误

随着你不断地获得递归相关的经验，编写和调试递归程序将会变得更自然。然而，刚开始找出你在一个递归程序中需要注意的东西是很困难的。下面是一个清单，它将帮助你辨别最常见错误的根源。

- *你的递归实现是以检查简单情况开始吗？* 在你尝试通过将一个问题转化成一个递归的子问题的方式来解决之前，必须先检查这个问题是否足够简单以至于这样的分解是不是必须的。在绝大多数情况下，递归函数以关键字 `if` 开始。如果你的函数不是这样，应该仔细地检查你的程序，并确保你知道自己正在做什么。
- *你是否已经正确地解决了这些简单情况？* 由于使用错误的方法解决简单情况，会在递归程序中产生令人惊讶的错误数量。如果简单情况是错误的，更复杂问题的递归方案将会继承同样的错误。例如，如果你错误地将 `fact（0）` 定义成 0 而不是 1，用任何参数调用 `fact` 都将返回 0。
- *你的递归分解让问题变简单了吗？* 对于能正确运行的递归，求解问题会变得越来越简单。更正式地讲，这里一定有某种**度量标准**（metric）（一种根据问题的难度从而给该问题赋一个整数值的标准的测量方法）其值会随着计算过程的进行而逐渐地变小。对于像 `fact` 以及 fib 这样的数学函数来说，以整型参数值作为度量标准。在每一次递归调用中，参数值都将变小。对于函数 `isPalindrome` 来说，由于字符串在每一次调用中都不断地变短，因此合适的度量标准就是字符串参数的长度。如果问题实例没有变得更简单，分解过程将会不断地产生越来越多的调用，会产生类似于死循环一样的被称为**无穷递归**（nonterminating recursion）的递归调用。
- *这些简单化的处理最终能到达简单情况吗，或者你遗漏了一些其他的可能性？* 错误的一般根源是，对于所有可以作为递归分解的结果情况中，没有包括简单情况测试。 339 例如，在如图 7-3 所示的 `isPalindrome` 实现中，函数中检查空字符以及单个字符的情况是非常重要的，即使用户从来不打算以空字符串调用函数 `isPallindrome`。随着递归分解的进行，字符串参数在每一层的递归调用中都将缩短两个字符。如果原始字符串参数的长度是偶数，那么递归分解永远不会进入单个字符的情况。
- *你的函数递归调用表示与原始问题的子问题在形式上是完全相同的吗？* 当你使用递归来分解一个问题时，子问题和原问题具有相同的形式是很重要的。如果分解调用改变了问题的本质或者违背了一个初始的假设，那么整个过程将会失败。正如本章中的一些示例所示，定义公有接口的函数作为一个简单的包装器函数是非常有用的，它调用一个更一般的私有的递归函数。由于私有函数具有更一般的形式，因此，它

通常更易于将原始问题进行分解，并使得子问题仍具有递归的结构。

- 当你运用递归稳步跳跃时，递归子问题的求解是否为原始问题提供了一个完整的解决方案？将一个问题分解成递归的子问题只是递归过程的一部分。一旦你获得了这些子问题的解，你还必须能够将它们重新整合以形成问题的一个完整解。检查其处理过程是否产生了问题的真实解的方法就是核查分解，这需要严谨地运用递归的稳步跳跃。检查当前函数调用的每一步，但假设每一个递归调用都生成了正确的答案。如果遵循这一过程并且产生了正确的解，你的程序应该能正常工作。

本章小结

- 本章介绍了递归的概念，它是一种强大的编程策略，其中，复杂的问题可以被分解成形式相同的更简单的问题。本章的重点包括：
- 递归与逐步求精法类似，两种策略都是将一个问题分解成更易于处理的简单问题。
340 递归的不同之处在于简单的子问题必须和原始问题具有相同的形式。
- 为了使用递归，你必须能够确定问题解的简单情况，以及允许你将任何复杂问题分解成具有相同类型的更简单问题的递归分解。
- 采用 C++，递归函数通常都有以下范例形式（范型）：

```
if (test for simple case) {
    Compute a simple solution without using recursion.
} else {
    Break the problem down into subproblems of the same form.
    Solve each of the subproblems by calling this function recursively.
    Reassemble the subproblem solutions into a solution for the whole.
}
```

- 和其他任何函数调用一样，递归函数也是使用完全相同的机制实现的。每次调用都创建了一个新的包含了调用中的局部变量的栈帧。由于计算机为每一次函数调用创建了一个单独的新栈帧，因此每一层递归分解的局部变量都是相互独立。
- 在你能够有效地使用递归之前，必须学会将你的分析限定在递归分解的一个单独的层次上，并且在没有跟踪整个计算过程的前提下确保所有更简单的递归调用的正确性。相信这些更简单的调用能够正确地工作被称为递归的稳步跳跃。
- 数学函数经常用递归关系的形式来表达它们的递归性质，其中，一个序列中的每个元素都根据它前面的元素定义。
- 即使某些递归函数可能与它们对应的迭代表示相比效率更低，但是递归本身并没有问题。和所有典型的各类算法一样，某些递归策略要比其他策略更为有效。
- 为了确保一个递归分解产生的子问题与原问题具有相同的形式，经常有必要使问题一般化。因此，采用一种简单的包装器函数实现一个特定问题的求解方案通常是非常有用的，包装器函数的唯一目的是调用一个辅助函数处理更一般的情况。
- 递归不一定由一个调用自身的函数组成，它可能涉及几个在一个循环模式中彼此调用的函数。涉及不止一个函数的递归被称为间接递归。
- 如果你能保持全局的观点而非简化的视角，那么你会在理解递归程序上更加成功。

以正确的方式思考递归问题并非易事。学习有效地使用递归需要不断地练习。对于大多数学生而言，掌握递归的概念就要花费数年。但是，由于递归将会成为你编程技能中最强大

的技术之一，因此值得花费时间去学习。 341

复习题

1. 定义术语*递归*和*迭代*。一个函数可以同时使用这两种策略实现吗？
2. 递归和传统的逐步求精法根本的不同之处是什么？
3. 在函数 collectContributions 的伪码中，if 语句如下所示：

```
if (n <= 100)
```

使用 <= 操作符代替简单地检查 n 是否等于 100 为什么很重要？

4. 标准的递归范型是什么？
5. 采用递归来有效地求解一个问题，该问题必须拥有的两个性质是什么？
6. 为什么术语*分而治之*适用于递归技术？
7. *递归的稳步跳跃*的含义是什么？作为一个程序员，这个概念对你为什么很重要？
8. 7.2.2 节给出了很长的一段分析来说明当 fact（4）被调用时内部发生了什么。采用这节作为模型，跟踪 fib(3) 的执行过程，并画出递归过程中创建的每一个栈帧。
9. 什么是*递归关系*？
10. 通过引入额外的规则，即一对兔子在生下三对兔子之后将会停止繁殖，请修改斐波那契兔子问题。这个假设会如何改变其递归关系？你需要在简单情况上做出什么改变？
11. 当使用图 7-1 给出的递归实现来计算 fib(n) 时，fib(1) 被调用了多少次？
12. 什么是*包装器函数*？为什么在编写递归函数时，它经常很有用？ 342
13. 如果你在函数 additiveSequence 中删除了 if(n==1) 的检测语句，将会发生什么？其实现代码如下所示：

```
int additiveSequence(int n, int t0, int t1) {
   if (n == 0) return t0;
   return additiveSequence(n - 1, t1, t0 + t1);
}
```

这个函数还能工作吗？为什么能或者不能？

14. 为什么在图 7-3 中函数 isPalindrome 的实现对空串以及单个字符的字符串的检查是很重要的？如果这个函数不检查单个字符的情况，取而代之的是只检查字符串的长度是否为 0，将会发生什么？这个函数还能正确地工作吗？
15. 解释图 7-4 给出的 isPalindrome 实现中的以下函数调用结果：

```
isPalindrome(str, p1 + 1, p2 - 1)
```

16. 什么是*间接递归*？
17. 如果像下面一样定义 isEven 和 isOdd，将会发生什么：

```
bool isEven(unsigned int n) {
   return !isOdd(n);
}

bool isOdd(unsigned int n) {
   return !isEven(n);
}
```

这个例子说明了 7.7.2 节中的哪一种错误？

18. 下面关于 isEven 和 isOdd 的定义也是不正确的：