Chapter 8
Recursive Strategies

Tactics without strategy is the noise before defeat.

— Sun Tzu, ~5th Century BCE

When a recursive decomposition follows directly from a mathematical definition, as it does in the case of the `fact` and `fib` functions in Chapter 7, applying recursion is not particularly hard. In most cases, you can translate the mathematical definition directly into a recursive implementation by plugging the appropriate expressions into the standard recursive paradigm. The situation changes, however, as you begin to solve more complex problems.

This chapter introduces several programming problems that seem—at least on the surface—much more difficult than those in Chapter 7. In fact, if you try to solve these problems without using recursion, relying instead on more familiar iterative techniques, you will find them quite difficult to solve. By contrast, each of the problems has a recursive solution that is surprisingly short. If you exploit the power of recursion, a few lines of code are sufficient for each task.

The brevity of these solutions, however, endows them with a deceptive aura of simplicity. The hard part of solving these problems has nothing to do with the length of the code. What makes these programs difficult is finding the recursive decomposition in the first place. Doing so occasionally requires some cleverness, but what you need even more is confidence. You have to accept the recursive leap of faith.

8.1 The Towers of Hanoi

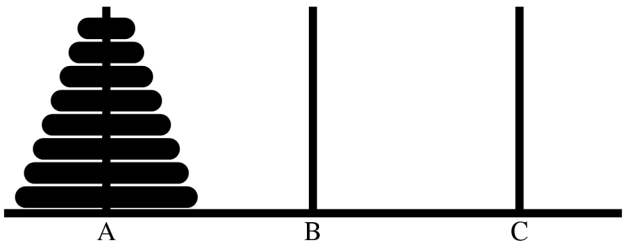
The first example in this chapter is a simple puzzle that has come to be known as the ***Towers of Hanoi.*** Invented by French mathematician Édouard Lucas in the 1880s, the Towers of Hanoi puzzle quickly became popular in Europe. Its success was due in part to the legend that grew up around the puzzle, which was described as follows in *La Nature* by the French mathematician Henri de Parville (as translated by the mathematical historian W. W. R. Ball):

> In the great temple at Benares beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly, the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When all the sixty-four disks shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

Over the years, the setting has shifted from India to Vietnam, but the puzzle and its legend remain the same.

As far as I know, the Towers of Hanoi puzzle has no practical use except one: teaching recursion to computer science students. In that domain, it has tremendous value because the solution involves nothing besides recursion. In contrast to most recursive algorithms that arise in response to real-world problems, the Towers of Hanoi problem has no extraneous complications that might interfere with your understanding and keep you from seeing how the recursive solution works. Because it works so well as an example, the Towers of Hanoi is included in most textbooks that treat recursion and has become—much like the "hello, world" program in Chapter 1—part of the cultural heritage that computer scientists share.

In commercial versions of the puzzle, the 64 golden disks of legend are replaced with eight wooden or plastic ones, which makes the puzzle considerably easier to solve (not to mention cheaper). The initial state of the puzzle looks like this:



At the beginning, all eight disks are on spire A. Your goal is to move these eight disks from spire A to spire B, but you must adhere to the following rules:

- You can only move one disk at a time.
- You are not allowed to move a larger disk on top of a smaller disk.

Framing the problem

In order to apply recursion to the Towers of Hanoi problem, you must first frame the problem in more general terms. Although the ultimate goal is moving eight disks from A to B, the recursive decomposition of the problem will involve moving smaller subtowers from spire to spire in various configurations. In the more general case, the problem you need to solve is moving a tower of a given height from one spire to another, using the third spire as a temporary repository. To ensure that all subproblems fit the original form, your recursive procedure must therefore take the following arguments:

1. The number of disks to move
2. The name of the spire where the disks start out
3. The name of the spire where the disks should finish
4. The name of the spire used for temporary storage

The number of disks to move is clearly an integer, and the fact that the spires are labeled with the letters *A, B,* and *C* suggests the use of type **char** to indicate which spire is involved. Knowing the types allows you to write a prototype for the operation that moves a tower, as follows:

```
void moveTower(int n, char start, char finish, char tmp);
```

To move the eight disks in the example, the initial call is

```
moveTower(8, 'A', 'B', 'C');
```

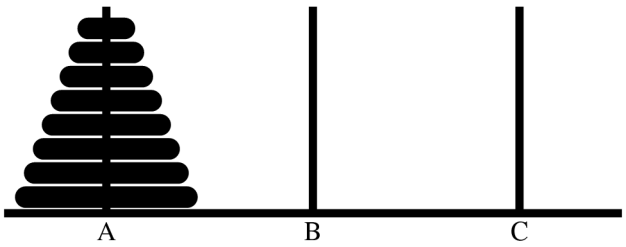
This function call corresponds to the English command "Move a tower of size 8 from spire A to spire B using spire C as a temporary." As the recursive decomposition proceeds, **moveTower** will be called with different arguments that move smaller towers in various configurations.

Finding a recursive strategy

Now that you have a more general definition of the problem, you can return to the problem of finding a strategy for moving a large tower. To apply recursion, you must first make sure that the problem meets the following conditions:

1. *There must be a simple case.* In this problem, the simple case occurs when **n** is equal to 1, which means that there is only a single disk to move. As long as you don't violate the rule of placing a larger disk on top of a smaller one, you can move a single disk as a single operation.
2. *There must be a recursive decomposition.* In order to implement a recursive solution, it must be possible to break the problem down into simpler problems in the same form as the original. This part of the problem is harder and will require closer examination.

To see how solving a simpler subproblem helps solve a larger problem, it helps to go back and consider the original example with eight disks.



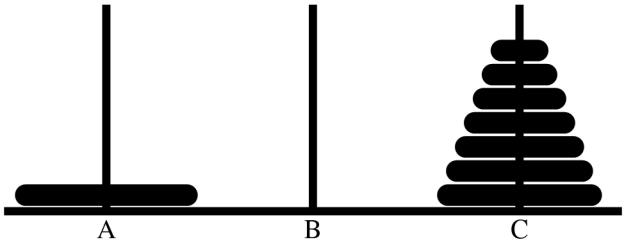
The goal here is to move eight disks from spire A to spire B. You need to ask yourself how it would help if you could solve the same problem for a smaller

number of disks. In particular, you should think about how being able to move a stack of seven disks would help you to solve the eight-disk case.

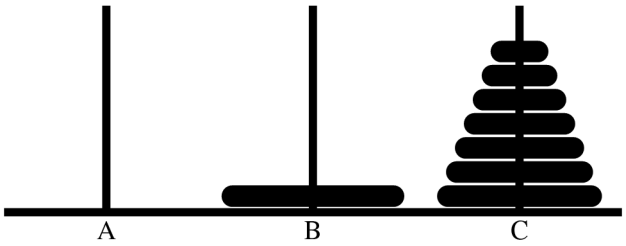
If you think about the problem for a few moments, it becomes clear that you can solve the problem by dividing it into these three steps:

1. Move the entire stack consisting of the top seven disks from spire A to spire C.
2. Move the bottom disk from spire A to spire B.
3. Move the stack of seven disks from spire C to spire B.

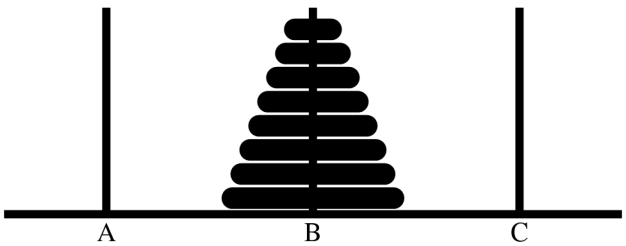
Executing the first step takes you to the following position:



Once you have gotten rid of the seven disks on top of the largest disk, the second step is simply to move that disk from spire A to spire B, which results in the following configuration:



All that remains is to move the tower of seven disks back from spire C to spire B, which is again a smaller problem of the same form. This operation is the third step in the recursive strategy, and leaves the puzzle in the desired final configuration:



That's it! You're finished. You've reduced the problem of moving a tower of size eight to one of moving a tower of size seven. More importantly, this recursive strategy generalizes to towers of size *N,* as follows:

1. Move the top *N*–1 disks from the start spire to the temporary spire.
2. Move a single disk from the start spire to the finish spire.
3. Move the stack of *N*–1 disks from the temporary spire back to the finish spire.

At this point, it is hard to avoid saying to yourself, "Okay, I can reduce the problem to moving a tower of size *N*–1, but how do I accomplish that?" The answer, of course, is that you move a tower of size *N*–1 in precisely the same way. You break that problem down into one that requires moving a tower of size *N*–2, which further breaks down into moving a tower of size *N*–3, and so forth, until there is just one disk to move. Psychologically, however, the important thing is to avoid asking that question altogether. The recursive leap of faith should be sufficient. You've reduced the scale of the problem without changing its form. That's the hard work. All the rest is bookkeeping, and it's best to let the computer take care of that.

Once you have identified the simple cases and the recursive decomposition, all you need to do is plug them into the standard recursive paradigm, which results in the following pseudocode procedure:

```
void moveTower(int n, char start, char finish, char tmp) {
   if (n == 1) {
      Move a single disk from start to finish.
   } else {
      Move a tower of size n - 1 from start to tmp.
      Move a single disk from start to finish.
      Move a tower of size n - 1 from tmp to finish.
   }
}
```

Validating the strategy

Although the pseudocode strategy is in fact correct, the derivation up to this point has been a little careless. Whenever you use recursion to decompose a problem, you must make sure that the new problems are identical in form to the original. The task of moving *N*–1 disks from one spire to another certainly sounds like an instance of the same problem and fits the **moveTower** prototype. Even so, there is a subtle but important difference. In the original problem, the destination and temporary spires are empty. When you move a tower of size *N*–1 to the temporary spire as part of the recursive strategy, you've left a disk behind on the starting spire. Does the presence of that disk change the nature of the problem and thus invalidate the recursive solution?

To answer this question, you need to think about the subproblem in light of the rules of the game. If the recursive decomposition doesn't end up violating the rules, everything should be okay. The first rule—that only one disk can be moved at a time—is not an issue. If there is more than a single disk, the recursive decomposition breaks the problem down to generate a simpler case. The steps in the pseudocode that actually transfer disks move only one disk at a time. The second rule—that you are not allowed to place a larger disk on top of a smaller one—is the critical one. You need to convince yourself that you will not violate this rule in the recursive decomposition.

The important observation to make is that, as you move a subtower from one spire to the other, the disk you leave behind on the original spire—and indeed any disk left behind at any previous stage in the operation—must be larger than anything in the current subtower. Thus, as you move those disks among the spires, the only disks below them will be larger in size, which is consistent with the rules.

Coding the solution

To complete the Towers of Hanoi solution, the only remaining step is to substitute function calls for the remaining pseudocode. The task of moving a complete tower requires a recursive call to the **moveTower** function. The only other operation is moving a single disk from one spire to another. For the purposes of writing a test program that displays the steps in the solution, all you need is a function that records its operation on the console. For example, you can implement the function **moveSingleDisk** as follows:

```
void moveSingleDisk(char start, char finish) {
   cout << start << " -> " << finish << endl;
}
```

The **moveTower** code itself looks like this:

```
void moveTower(int n, char start, char finish, char tmp) {
   if (n == 1) {
      moveSingleDisk(start, finish);
   } else {
      moveTower(n - 1, start, tmp, finish);
      moveSingleDisk(start, finish);
      moveTower(n - 1, tmp, finish, start);
   }
}
```

The complete implementation appears in Figure 8-1.

FIGURE 8-1 Program to solve the Towers of Hanoi puzzle

```
/*
 * File: Hanoi.cpp
 * ---------------
 * This program solves the Tower of Hanoi puzzle.
 */

#include <iostream>
#include "simpio.h"
using namespace std;

/* Function prototypes */

void moveTower(int n, char start, char finish, char tmp);
void moveSingleDisk(char start, char finish);

/* Main program */

int main() {
   int n = getInteger("Enter number of disks: ");
   moveTower(n, 'A', 'B', 'C');
   return 0;
}

/*
 * Function: moveTower
 * Usage: moveTower(n, start, finish, tmp);
 * ----------------------------------------
 * Moves a tower of size n from the start spire to the finish
 * spire using the tmp spire as the temporary repository.
 */

void moveTower(int n, char start, char finish, char tmp) {
   if (n == 1) {
      moveSingleDisk(start, finish);
   } else {
      moveTower(n - 1, start, tmp, finish);
      moveSingleDisk(start, finish);
      moveTower(n - 1, tmp, finish, start);
   }
}

/*
 * Function: moveSingleDisk
 * Usage: moveSingleDisk(start, finish);
 * -------------------------------------
 * Executes the transfer of a single disk from the start spire to the
 * finish spire.  In this implementation, the move is simply displayed
 * on the console; in a graphical implementation, the code would update
 * the graphics window to show the new arrangement.
 */

void moveSingleDisk(char start, char finish) {
   cout << start << " -> " << finish << endl;
}
```

Tracing the recursive process

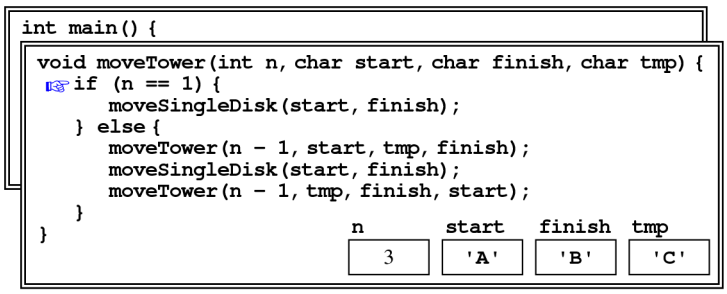
The only problem with this implementation of **moveTower** is that it seems like magic. If you're like most students learning about recursion for the first time, the solution seems so short that you feel sure there must be something missing. Where is the strategy? How can the computer know which disk to move first and where it should go?

The answer is that the recursive process—breaking a problem down into smaller subproblems of the same form and then providing solutions for the simple cases—is all you need to solve the problem. If you trust the recursive leap of faith, you're done. You can skip this section of the book and go on to the next. If, on the other hand, you're still suspicious, it may be necessary for you to go through the steps in the complete process and watch what happens.

To make the problem more manageable, consider what happens if there are only three disks in the original tower. The main program call is therefore

```
moveTower(3, 'A', 'B', 'C');
```

To trace how this call computes the steps necessary to transfer a tower of size 3, all you need to do is keep track of the operation of the program, using precisely the same strategy as in the factorial example from Chapter 7. For each new function call, you introduce a stack frame that shows the values of the parameters for that call. The initial call to **moveTower**, for example, creates the following stack frame:



As the arrow in the code indicates, the function has just been called, so execution begins with the first statement in the function body. The current value of **n** is not equal to 1, so the program skips ahead to the **else** clause and executes the statement

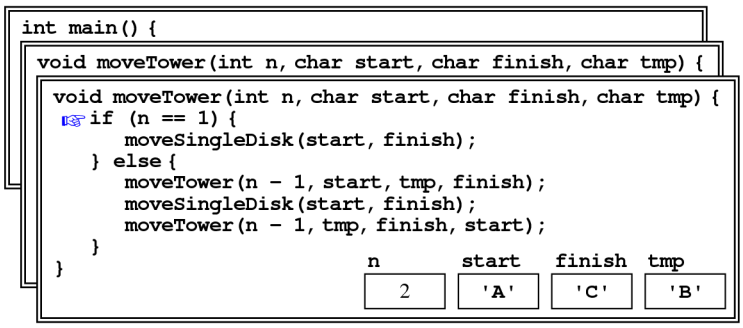
```
moveTower(n-1, start, tmp, finish);
```

As with any function call, the first step is to evaluate the arguments. To do so, you need to determine the values of the variables **n**, **start**, **tmp**, and **finish**.

Whenever you need to find the value of a variable, you use the value as it is defined in the current stack frame. Thus, the **moveTower** call is equivalent to

```
moveTower(2, 'A', 'C', 'B');
```

This operation, however, indicates another function call, which means that the current operation is suspended until the new function call is complete. To trace the operation of the new function call, you need to generate a new stack frame and repeat the process. As always, the parameters in the new stack frame are copied from the calling arguments in the order in which they appear. Thus, the new stack frame looks like this:



As the diagram illustrates, the new stack frame has its own set of variables, which temporarily supersede the variables in frames that are further down on the stack. Thus, as long as the program is executing in this stack frame, **n** will have the value 2, **start** will be **'A'**, **finish** will be **'C'**, and **tmp** will be **'B'**. The old values in the previous frame will not reappear until the subtask represented by this call to **moveTower** is complete.

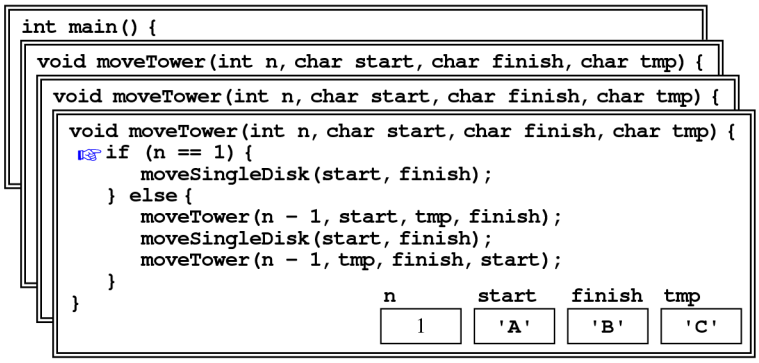
The evaluation of the recursive call to **moveTower** proceeds exactly like the original one. Once again, **n** is not 1, which requires another call of the form

```
moveTower(n-1, start, tmp, finish);
```

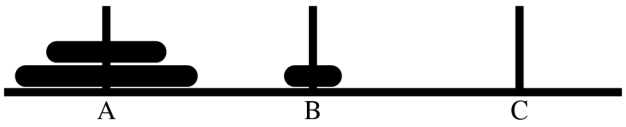
Because this call comes from a different stack frame, however, the value of the individual variables are different from those in the original call. If you evaluate the arguments in the context of the current stack frame, you discover that this function call is equivalent to

```
moveTower(1, 'A', 'B', 'C');
```

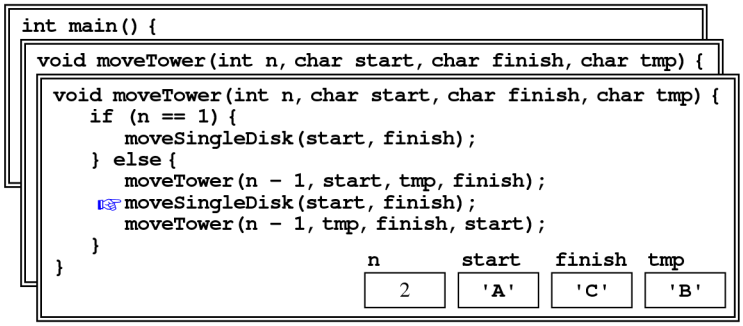
The effect of making this call is to introduce yet another stack frame for the **moveTower** function, as follows:



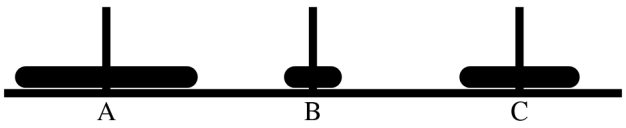
This call to **moveTower**, however, does represent the simple case. Since n is 1, the program calls the **moveSingleDisk** function to move a disk from A to B, leaving the puzzle in the following configuration:



At this point, the most recent call to **moveTower** is complete and the function returns. In the process, its stack frame is discarded, which brings the execution back to the previous stack frame, having just completed the first statement in the **else** clause:



The call to **moveSingleDisk** again represents a simple operation, which leaves the puzzle in the following state:



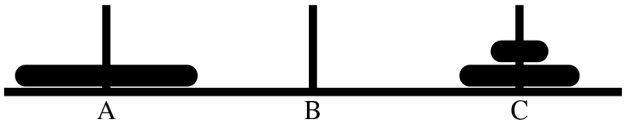
With the **moveSingleDisk** operation completed, the only remaining step required to finish the current call to **moveTower** is the last statement in the function:

```
moveTower(n-1, tmp, finish, start);
```

Evaluating these arguments in the context of the current frame reveals that this call is equivalent to

```
moveTower(1, 'B', 'C', 'A');
```

Once again, this call requires the creation of a new stack frame. By this point in the process, however, you should be able to see that the effect of this call is simply to move a tower of size 1 from B to C, using A as a temporary repository. Internally, the function determines that **n** is 1 and then calls **moveSingleDisk** to reach the following configuration:

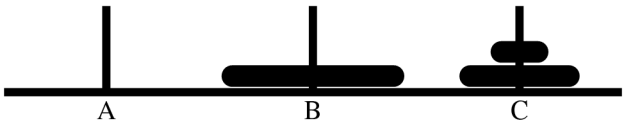


This operation again completes a call to **moveTower**, allowing it to return to its caller having completed the subtask of moving a tower of size 2 from A to C. Discarding the stack frame from the just-completed subtask reveals the stack frame for the original call to **moveTower**, which is now in the following state:

```
int main() {
   void moveTower(int n, char start, char finish, char tmp) {
      if (n == 1) {
         moveSingleDisk(start, finish);
      } else {
         moveTower(n - 1, start, tmp, finish);
      ☞ moveSingleDisk(start, finish);
         moveTower(n - 1, tmp, finish, start);
      }
   }
```

n	start	finish	tmp
3	'A'	'B'	'C'

The next step is to call **moveSingleDisk** to move the largest disk from A to B, which results in the following position:



The only operation that remains is to call

```
moveTower(n-1, tmp, finish, start);
```

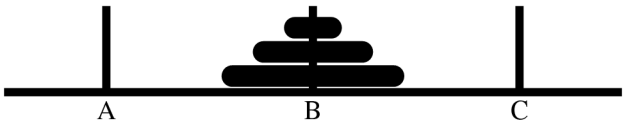
with the arguments from the current stack frame, which are

```
moveTower(2, 'C', 'B', 'A');
```

If you're still suspicious of the recursive process, you can draw the stack frame created by this function call and continue tracing the process to its ultimate conclusion. At some point, however, it is essential that you trust the recursive process enough to see that function call as a single operation having the effect of the following command in English:

Move a tower of size 2 from C to B, using A as a temporary repository.

If you think about the process in this holistic form, you can immediately see that completion of this step will move the tower of two disks back from C to B, leaving the desired final configuration:



8.2 The subset-sum problem

Although the Towers of Hanoi problem offers a wonderful illustration of the power of recursion, its effectiveness as an example is compromised by its lack of any practical application. Many people are drawn to programming because it enables them to solve practical problems. If the only examples of recursion are like the Towers of Hanoi, it's easy to conclude that recursion is useful only for solving abstract puzzles. Nothing could be further from the truth. Recursive strategies give rise to extremely efficient solutions to practical problems—most notably the problem of sorting introduced in Chapter 10—that are hard to solve in other ways.

The problem covered in this section is called the ***subset-sum problem,*** which can be defined as follows:

> Given a set of integers and a target value, determine whether it is possible to find a subset of those integers whose sum is equal to the specified target.

For example, given the set { –2, 1, 3, 8 } and the target value 7, the answer to the subset-sum question is yes, because the subset { –2, 1, 8 } adds up to 7. If the target value had been 5, however, the answer would be no, because there is no way to choose a subset of the integers in { –2, 1, 3, 8 } that adds up to 5.

It is easy to translate the idea of the subset-sum problem into C++. The concrete goal is to write a predicate function

```
bool subsetSumExists(Set<int> & set, int target);
```

that takes the required information and returns **true** if it is possible to generate the value **target** by adding up some combination of elements chosen from **set**.

Even though it might at first seem that the subset-sum problem is just as esoteric as the Towers of Hanoi, it has significant importance in both the theory and practice of computer science. As you will discover in Chapter 10, the subset-sum problem is an instance of an important class of computational problems that are hard to solve efficiently. That very fact, however, makes problems like subset-sum useful in applications where the goal is to keep information secret. The first implementation of public-key cryptography, for example, used a variant of the subset-sum problem as its mathematical foundation. By basing their operation on problems that are provably hard, modern encryption strategies are more difficult to break.

The search for a recursive solution

The subset-sum problem is difficult to solve using a traditional iterative approach. To make any headway, you need to think recursively. As always, you therefore need to identify a simple case and a recursive decomposition. In applications that work with sets, the simple case is almost always when the set is empty. If the set is empty, there is no way that you can add elements to produce a target value unless the target is zero. That discovery suggests that the code for **subsetSumExists** will start off like this:

```
bool subsetSumExists(Set<int> & set, int target) {
   if (set.isEmpty()) {
      return target == 0;
   } else {
      Find a recursive decomposition that simplifies the problem.
   }
}
```

In this problem, the hard part is finding that recursive decomposition.

When you are looking for a recursive decomposition, you need to be on the lookout for some value in the inputs—which are conveyed as arguments in the C++ formulation of the problem—that you can make smaller. In this case, what you need to do is make the set smaller, because what you're trying to do is move toward the simple case that occurs when the set is empty. If you take an element out of the set, what's left over is smaller by one element. The operations exported by the **Set**

class make it easy to choose an element from a set and then determine what's left over. All you need is the following code:

```
int element = set.first();
Set<int> rest = set - element;
```

The **first** method returns the element of the set that appears first in its iteration order, and the expression involving the overloaded **-** operator produces the set that contains every element in **set** except the value of **element**. The fact that **element** is first in iteration order is not really important here. All you really need is some way to choose some element and then to create a smaller set by removing the element you selected from the original set.

The inclusion/exclusion pattern

Making the set smaller, however, is not enough to solve this problem. The code to divide a set into an element and the rest of the set will come up again in many recursive applications and is part of a general programming pattern for working with the **Set** class. Structurally, you know that **subsetSumExists** must call itself recursively on the smaller set now stored in the variable **rest**. What you haven't yet determined is how the solution to these recursive subproblems will help to solve the original.

The key insight you need to solve this problem is that there are two ways that you might be able to produce the desired target sum after you have identified a particular element. One possibility is that the subset you're looking for *includes* that element. For that to happen, it must be possible to take the rest of the set and produce the value **target - element**. The other possibility is that the subset you're looking for *excludes* that element, in which case it must be possible to generate the value **target** using only the leftover set of elements. This insight is enough to complete the implementation of **subsetSumExists**, as follows:

```
bool subsetSumExists(Set<int> & set, int target) {
   if (set.isEmpty()) {
      return target == 0;
   } else {
      int element = set.first();
      Set<int> rest = set - element;
      return subsetSumExists(rest, target)
          || subsetSumExists(rest, target - element);
   }
}
```

Because the recursive strategy subdivides the general case into one branch that includes a particular element and another that excludes it, this strategy is sometimes

called the ***inclusion/exclusion pattern.*** As you work through the exercises in this chapter as well as several subsequent ones, you will find that this pattern, with slight variations, comes up in many different applications. Although the pattern is easiest to recognize when you are working with sets, it also comes up in applications involving vectors and strings, and you should be on the lookout for it in those situations as well.

8.3 Generating permutations

Many word games and puzzles require the ability to rearrange a set of letters to form a word. Thus, if you wanted to write a Scrabble program, it would be useful to have a facility for generating all possible arrangements of a particular set of tiles. In word games, such arrangements are generally called ***anagrams.*** In mathematics, they are known as ***permutations.***

Let's suppose you want to write a function

```
Set<string> generatePermutations(string str);
```

that returns a set containing all permutations of the string. For example, if you call

```
generatePermutations("ABC")
```

the function should return a set containing the following elements:

{ **"ABC"**, **"ACB"**, **"BAC"**, **"BCA"**, **"CAB"**, **"CBA"** }

How might you go about implementing the **generatePermutations** function? If you are limited to iterative control structures, finding a general solution that works for strings of any length is difficult. Thinking about the problem recursively, on the other hand, leads to a relatively straightforward solution.

As is usually the case with recursive programs, the hard part of the solution process is figuring out how to divide the original problem into simpler instances of the same problem. In this case, to generate all permutations of a string, you need to discover how being able to generate all permutations of a shorter string might contribute to the solution.

Before you look at my solution on the next page, stop and think about this problem for a few minutes. When you are first learning about recursion, it is easy to look at a recursive solution and believe that you could have generated it on your own. Without trying it first, however, it is hard to know whether you would have come up with the necessary recursive insight.

Finding the recursive insight

To give yourself more of a feel for the problem, it helps to consider a concrete case. Suppose you want to generate all permutations of a five-character string, such as `"ABCDE"`. In your solution, you can apply the recursive leap of faith to generate all permutations of any shorter string. Just assume that the recursive calls work and be done with it. Once again, the critical question is how being able to permute shorter strings helps you solve the problem of permuting the original five-character string.

If you focus on breaking the five-character permutation problem down into some number of instances of the permutation problem involving four-character strings, you will soon discover that the permutations of the five-character string `"ABCDE"` consist of the following strings:

- The character `'A'` followed by every possible permutation of `"BCDE"`
- The character `'B'` followed by every possible permutation of `"ACDE"`
- The character `'C'` followed by every possible permutation of `"ABDE"`
- The character `'D'` followed by every possible permutation of `"ABCE"`
- The character `'E'` followed by every possible permutation of `"ABCD"`

More generally, you can construct the set of all permutations of a string of length n by selecting each character in turn and then, for each of those n possible first characters, concatenating the selected character on to the front of every possible permutation of the remaining $n - 1$ characters. The problem of generating all permutations of $n - 1$ characters is a smaller instance of the same problem and can therefore be solved recursively.

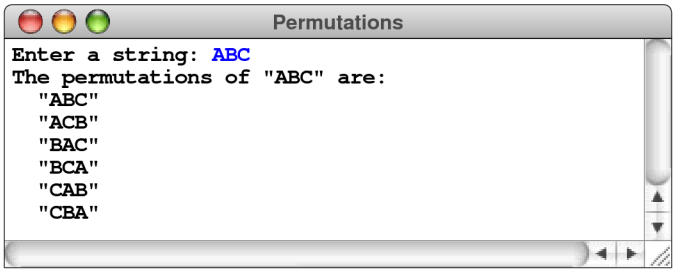
As always, you also need to define a simple case. One possibility is to check to see whether the string contains a single character. Computing all the permutations of a single-character string is easy, because there is only one possible ordering. In string processing, however, the best choice for the simple case is rarely a one-character string, because there is in fact an even simpler alternative: the empty string containing no characters at all. Just as there is only one ordering for a single-character string, there is only one way to write the empty string. If you call `generatePermutations("")`, you should get back a set containing a single element, which is the empty string.

Once you have both the simple case and the recursive insight, writing the code for `generatePermutations` becomes reasonably straightforward. The code for `generatePermutations` appears in Figure 8-2, along with a simple test program that asks the user for a string and then prints out every possible permutation of the characters in that string.

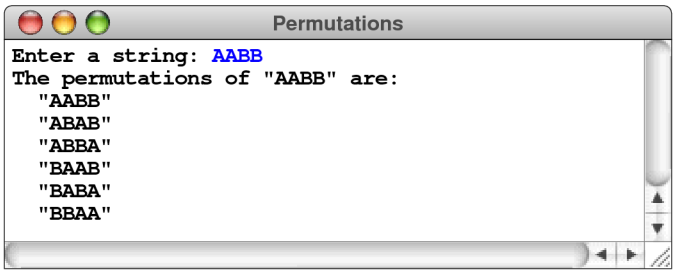
FIGURE 8-2 Program to generate all permutations of a string

```
/*
 * File: Permutations.cpp
 * ----------------------
 * This file generates all permutations of an input string.
 */

#include <iostream>
#include "set.h"
#include "simpio.h"
using namespace std;

/* Function prototypes */

Set<string> generatePermutations(string str);

/* Main program */

int main() {
   string str = getLine("Enter a string: ");
   cout << "The permutations of \"" << str << "\" are:" << endl;
   foreach (string s in generatePermutations(str)) {
      cout << "  \"" << s << "\"" << endl;
   }
   return 0;
}

/*
 * Function: generatePermutations
 * Usage: Set<string> permutations = generatePermutations(str);
 * ------------------------------------------------------------
 * Returns a set consisting of all permutations of the specified string.
 * This implementation uses the recursive insight that you can generate
 * all permutations of a string by selecting each character in turn,
 * generating all permutations of the string without that character,
 * and then concatenating the selected character on the front of each
 * string generated.
 */

Set<string> generatePermutations(string str) {
   Set<string> result;
   if (str == "") {
      result += "";
   } else {
      for (int i = 0; i < str.length(); i++) {
         char ch = str[i];
         string rest = str.substr(0, i) + str.substr(i + 1);
         foreach (string s in generatePermutations(rest)) {
            result += ch + s;
         }
      }
   }
   return result;
}
```

If you run the **Permutations** program and enter the string **"ABC"**, you see the following output:

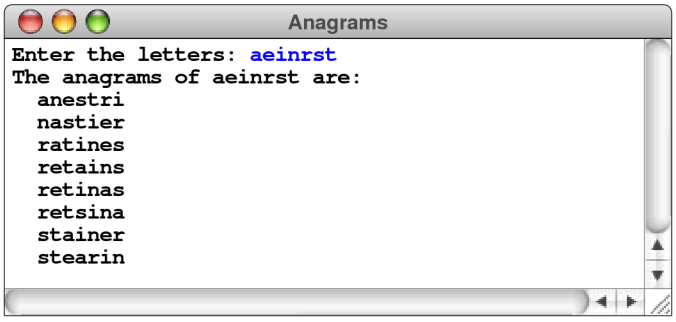


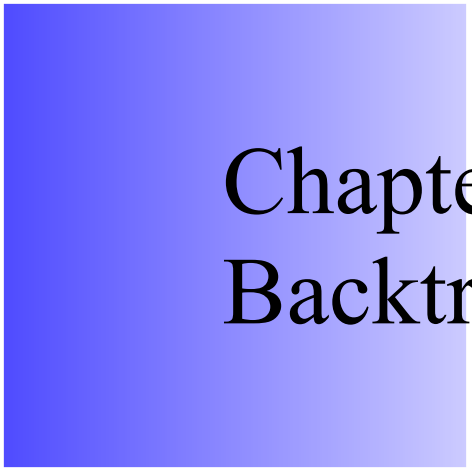
The use of sets in this application ensures that the program generates permutations in alphabetical order and that each distinct ordering of the characters appears exactly once, even if there are repeated letters in the input string. For example, if you enter the string **AABB** in response to the prompt, the program produces only six permutations, as follows:



The recursive process adds a new element to this set the full total of 24 (4!) times, but the implementation of the **Set** class ensures that no duplicate values appear.

You can use the **generatePermutations** function to generate all anagrams of a word by changing the main program from Figure 8-2 so that it checks each string against the English lexicon. If you enter the string **"aeinrst"**, you get the following output—a list that serious Scrabble players will recognize instantly:





Chapter 9
Backtracking Algorithms

Truth is not discovered by proofs but by exploration. It is always experimental.

— Simone Weil, *The New York Notebook,* 1942

For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path. If you make the correct set of choices, you end up at the solution. On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called ***backtracking algorithms.***

If you think about a backtracking algorithm as the process of repeatedly exploring paths until you encounter the solution, the process appears to have an iterative character. As it happens, however, most problems of this form are easier to solve recursively. The fundamental recursive insight is simply this: a backtracking problem has a solution if and only if at least one of the smaller backtracking problems that result from making each possible initial choice has a solution. The examples in this chapter are designed to illustrate this process and demonstrate the power of recursion in this domain.

9.1 Recursive backtracking in a maze

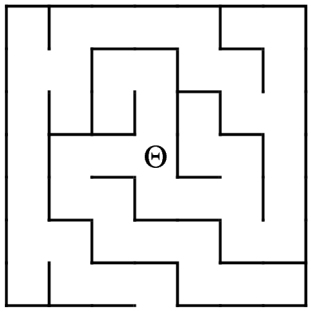
Once upon a time, in the days of Greek mythology, the Mediterranean island of Crete was ruled by a tyrannical king named Minos. From time to time, Minos demanded tribute from the city of Athens in the form of young men and women, whom he would sacrifice to the Minotaur, a fearsome beast with the head of a bull and the body of a man. To house this deadly creature, Minos forced his servant Daedalus (the engineering genius who later escaped by constructing a set of wings) to build a vast underground labyrinth at Knossos. The young sacrifices from Athens would be led into the labyrinth, where they would be eaten by the Minotaur before they could find their way out. This tragedy continued until young Theseus of Athens volunteered to be one of the sacrifices. Following the advice of Minos's daughter Ariadne, Theseus entered the labyrinth with a sword and a ball of string. After slaying the monster, Theseus was able to find his way back to the exit by unwinding the string as he went along.

The right-hand rule

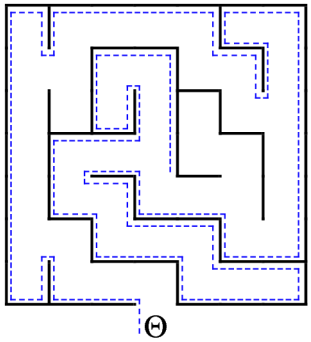
Ariadne's strategy is an algorithm for escaping from a maze, but not everyone trapped in a maze is lucky enough to have a ball of string. Fortunately, there are other strategies for solving a maze. Of these strategies, the best known is called the ***right-hand rule,*** which can be expressed in the following pseudocode form:

```
Put your right hand against a wall.
while (you have not yet escaped from the maze) {
   Walk forward keeping your right hand on a wall.
}
```

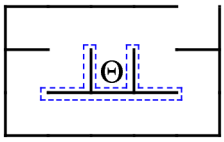
To visualize the operation of the right-hand rule, imagine that Theseus has successfully dispatched the Minotaur and is now standing in the position marked by the first character in Theseus's name, the Greek letter theta (Θ):



If Theseus puts his right hand on the wall and then follows the right-hand rule from there, he will trace out the path shown by the dashed line in this diagram:



Unfortunately, the right-hand rule does not in fact work in every maze. If there is a loop that surrounds the starting position, Theseus can get trapped in an infinite loop, as illustrated by the following simple maze:

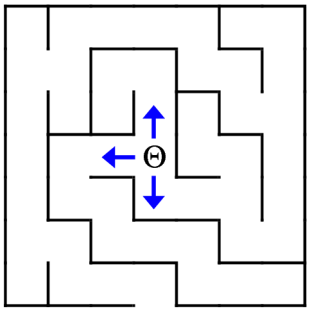


Finding a recursive approach

As the **`while`** loop in its pseudocode form makes clear, the right-hand rule is an *iterative* strategy. You can, however, also think about the process of solving a maze from a *recursive* perspective. To do so, you must adopt a different mindset. You can no longer think about the problem in terms of finding a complete path. Instead, your goal is to find a recursive insight that simplifies the problem, one step at a

time. Once you have made the simplification, you use the same process to solve each of the resulting subproblems.

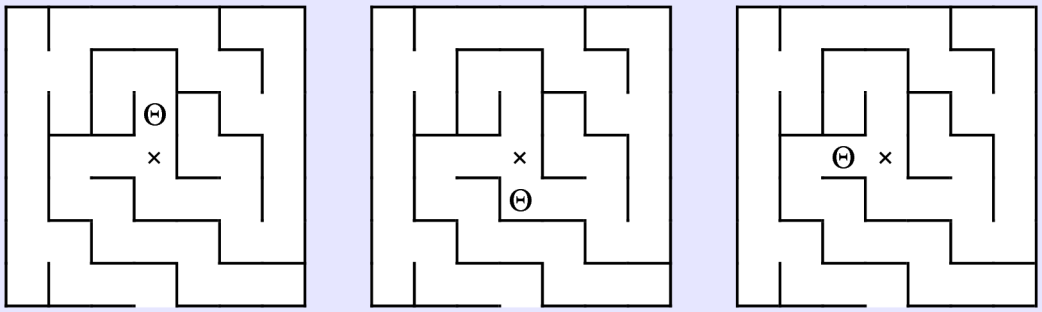
Let's go back to the initial configuration of the maze shown in the illustration of the right-hand rule. Put yourself in Theseus's place. From the initial configuration, you have three choices, as indicated by the arrows in the following diagram:



The exit, if any, must lie along one of those paths. Moreover, if you choose the correct direction, you will be one step closer to the solution. The maze has therefore become simpler along that path, which is the key to a recursive solution. This observation suggests the necessary recursive insight. The original maze has a solution if and only if it is possible to solve at least one of the new mazes shown in Figure 9-1. The × in each diagram marks the original starting square and is off-limits for any of the recursive solutions because the optimal solution will never have to backtrack through this square.

By looking at the mazes in Figure 9-1, it is easy to see—at least from your global vantage point—that the submazes labeled (a) and (c) represent dead-end paths and that the only solution begins in the direction shown in the submaze (b). If you are thinking recursively, however, you don't need to carry on the analysis all the way to

FIGURE 9-1 Recursive decomposition of a maze

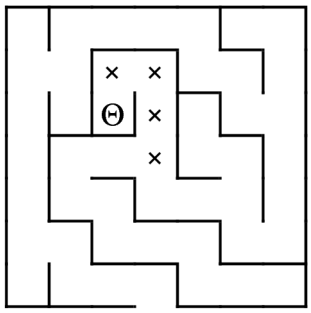


(a) (b) (c)

the solution. You have already decomposed the problem into simpler instances. All you need to do is rely on the power of recursion to solve the individual subproblems, and you're home free. You still have to identify a set of simple cases so that the recursion can terminate, but the hard work has been done.

Identifying the simple cases

What constitutes the simple case for a maze? One possibility is that you might already be standing outside the maze. If so, you're finished. Clearly, this situation represents one simple case. There is, however, another possibility. You might also reach a blind alley where you've run out of places to move. For example, if you try to solve the sample maze by moving north and then continue to make recursive calls along that path, you will eventually be in the position of trying to solve the following maze:



At this point, you've run out of room to maneuver. Every path from the new position is either marked or blocked by a wall, which makes it clear that the maze has no solution from this point. Thus, the maze problem has a second simple case in which every direction from the current square is blocked, either by a wall or a marked square.

It is easier to code the recursive algorithm if, instead of checking for marked squares as you consider the possible directions of motion, you go ahead and make the recursive calls on those squares. If you check at the beginning of the procedure to see whether the current square is marked, you can terminate the recursion at that point. After all, if you find yourself positioned on a marked square, you must be retracing your path, which means that the optimal solution must lie in some other direction.

Thus, the two simple cases for this problem are as follows:

1. If the current square is outside the maze, the maze is solved.
2. If the current square is marked, the maze is unsolvable, at least along the path you've chosen so far.

Coding the maze solution algorithm

Although the recursive insight and the simple cases are all you need to solve the problem on a conceptual level, writing a complete program to navigate a maze requires you to consider a number of implementation details as well. For example, you need to decide on a representation for the maze itself that allows you to figure out where the walls are, keep track of the current position, indicate that a particular square is marked, and determine whether you have escaped from the maze. While designing an appropriate data structure for the maze is an interesting programming challenge in its own right, it has very little to do with understanding the recursive algorithm, which is the focus of this discussion. If anything, the details of the data structure are likely to get in the way and make it more difficult for you to understand the algorithmic strategy as a whole. Fortunately, it is possible to set those details aside by introducing a new interface that hides some of the complexity. The **maze.h** interface in Figure 9-2 exports a class called **Maze** that encapsulates all the information necessary to keep track of the passages in a maze and to display that maze in the graphics window.

Once you have access to the **Maze** class, writing a program to solve a maze becomes much simpler. The goal of this exercise is to write a function

```
bool solveMaze(Maze & maze, Point pt);
```

The arguments to **solveMaze** are (1) the **Maze** object that holds the data structure and (2) the starting position, which changes for each of the recursive subproblems. To ensure that the recursion can terminate when a solution is found, the **solveMaze** function returns **true** if a solution has been found, and **false** otherwise.

Given this definition of **solveMaze**, the main program looks like this:

```
int main() {
   initGraphics();
   Maze maze("SampleMaze.txt");
   maze.showInGraphicsWindow();
   if (solveMaze(maze, maze.getStartPosition())) {
      cout << "The marked path is a solution." << endl;
   } else {
      cout << "No solution exists." << endl;
   }
   return 0;
}
```

FIGURE 9-2 The `maze.h` interface

```
/*
 * File: maze.h
 * ------------
 * This interface exports the Maze class.
 */

#ifndef _maze_h
#define _maze_h

#include <string>
#include "grid.h"
#include "gwindow.h"
#include "point.h"

/*
 * Class: Maze
 * -----------
 * This class represents a two-dimensional maze contained in a rectangular
 * grid of squares.  The maze is read in from a data file in which the
 * characters '+', '-', and '|' represent corners, horizontal walls, and
 * vertical walls, respectively; spaces represent open passageway squares.
 * The starting position is indicated by the character 'S'.  For example,
 * the following data file defines a simple maze:
 *
 *       +-+-+-+-+-+
 *       |     |
 *       + +-+ + +-+
 *       |S  |     |
 *       +-+-+-+-+-+
 */

class Maze {

public:

/*
 * Constructor: Maze
 * Usage: Maze maze(filename);
 *        Maze maze(filename, gw);
 * ------------------------------
 * Constructs a new maze by reading the specified data file.  If the
 * second argument is supplied, the maze is displayed in the center
 * of the graphics window.
 */

   Maze(std::string filename);
   Maze(std::string filename, GWindow & gw);
```

FIGURE 9-2 The `maze.h` interface (continued)

```
/*
 * Method: getStartPosition
 * Usage: Point start = maze.getStartPosition();
 * ---------------------------------------------
 * Returns a Point indicating the coordinates of the start square.
 */

   Point getStartPosition();

/*
 * Method: isOutside
 * Usage: if (maze.isOutside(pt)) . . .
 * ------------------------------------
 * Returns true if the specified point is outside the boundary of the maze.
 */

   bool isOutside(Point pt);

/*
 * Method: wallExists
 * Usage: if (maze.wallExists(pt, dir)) . . .
 * ------------------------------------------
 * Returns true if there is a wall in direction dir from the square at pt.
 */

   bool wallExists(Point pt, Direction dir);

/*
 * Method: markSquare
 * Usage: maze.markSquare(pt);
 * ---------------------------
 * Marks the specified square in the maze.
 */

   void markSquare(Point pt);

/*
 * Method: unmarkSquare
 * Usage: maze.unmarkSquare(pt);
 * -----------------------------
 * Unmarks the specified square in the maze.
 */

   void unmarkSquare(Point pt);

/*
 * Method: isMarked
 * Usage: if (maze.isMarked(pt)) . . .
 * -----------------------------------
 * Returns true if the specified square is marked.
 */

   bool isMarked(Point pt);

};

#endif
```

The code for the **solveMaze** function appears in Figure 9-3, along with the function **adjacentPoint(start, dir)**, which returns the point you reach if you move in the specified direction from the starting point.

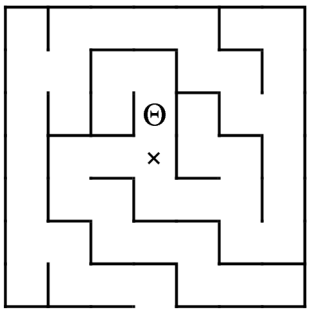
FIGURE 9-3 The implementation of the solveMaze function

```
/*
 * Function: solveMaze
 * Usage: solveMaze(maze, start);
 * ------------------------------
 * Attempts to generate a solution to the current maze from the specified
 * start point.  The solveMaze function returns true if the maze has a
 * solution and false otherwise.  The implementation uses recursion
 * to solve the submazes that result from marking the current square
 * and moving one step along each open passage.
 */

bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjacentPoint(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}

/*
 * Function: adjacentPoint
 * Usage: Point finish = adjacentPoint(start, dir);
 * ------------------------------------------------
 * Returns the point that results from moving one square from start
 * in the direction specified by dir.  For example, if pt is the
 * point (1, 1), calling adjacentPoint(pt, EAST) returns the
 * point (2, 1).  To maintain consistency with the graphics package,
 * the y coordinates increase as you move downward on the screen.  Thus,
 * moving NORTH decreases the y component, and moving SOUTH increases it.
 */

Point adjacentPoint(Point start, Direction dir) {
   switch (dir) {
    case NORTH: return Point(start.getX(), start.getY() - 1);
    case EAST:  return Point(start.getX() + 1, start.getY());
    case SOUTH: return Point(start.getX(), start.getY() + 1);
    case WEST:  return Point(start.getX() - 1, start.getY());
   }
   return start;
}
```

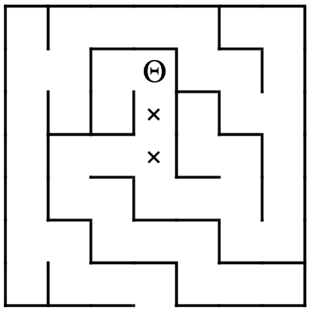
Convincing yourself that the solution works

In order to use recursion effectively, at some point you must be able to look at a recursive function like the **`solveMaze`** example in Figure 9-3 and say to yourself something like this: "I understand how this works. The problem is getting simpler because more squares are marked each time. The simple cases are clearly correct. This code must do the job." For most of you, however, that confidence in the power of recursion will not come easily. Your natural skepticism makes you want to see the steps in the solution. The problem is that, even for a maze as simple as the one shown earlier in this chapter, the complete history of the steps involved in the solution is far too large to think about comfortably. Solving that maze, for example, requires 66 calls to **`solveMaze`** that are nested 27 levels deep when the solution is finally discovered. If you attempt to trace the code in detail, you will almost certainly get lost.

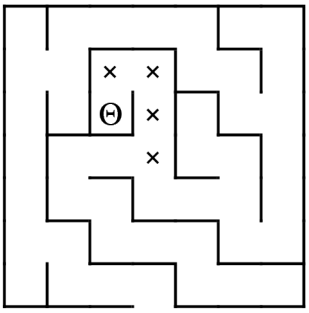
If you are not yet ready to accept the recursive leap of faith, the best you can do is track the operation of the code in a more general sense. You know that the code first tries to solve the maze by moving one square to the north, because the **`for`** loop goes through the directions in the order defined by the **`Direction`** enumeration. Thus, the first step in the solution process is to make a recursive call that starts in the following position:



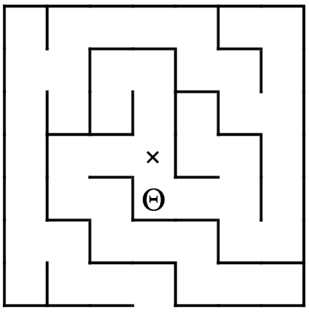
At this point, the same process occurs again. The program again tries to move north and makes a new recursive call in the following position:



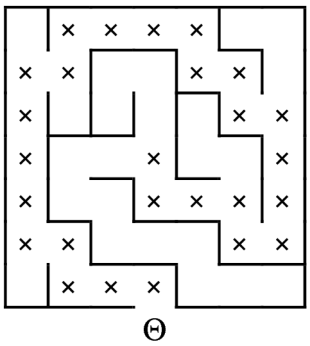
At this level of the recursion, moving north is no longer possible, so the **for** loop cycles through the other directions. After a brief excursion southward, upon which the program encounters a marked square, the program finds the opening to the west and proceeds to generate a new recursive call. The same process occurs in this new square, which in turn leads to the following configuration:



In this position, none of the directions in the **for** loop do any good; every square is either blocked by a wall or already marked. Thus, when the **for** loop at this level exits at the bottom, it unmarks the current square and returns to the previous level. It turns out that all the paths have also been explored in this position, so the program once again unmarks the square and returns to the next higher level in the recursion. Eventually, the program backtracks all the way to the initial call, having completely exhausted the possibilities that begin by moving north. The **for** loop then tries the eastward direction, finds it blocked, and continues on to explore the southern corridor, beginning with a recursive call in the following configuration:



From here on, the same process ensues. The recursion systematically explores every corridor along this path, backing up through the stack of recursive calls whenever it reaches a dead end. The only difference along this route is that eventually—after descending through an additional recursive level for every step on the path—the program makes a recursive call in the following position:



At this point, Theseus is outside the maze, so the simple case kicks in and returns `true` to its caller. This value is then propagated back through all 27 levels of the recursion, eventually returning back to the main program.

9.2 Backtracking and games

Although backtracking is easiest to illustrate in the context of a maze, the strategy is considerably more general. For example, you can apply backtracking to most two-player strategy games. The first player has several choices for an initial move. Depending on which move is chosen, the second player then has a particular set of responses. Each of these responses leads in turn to new options for the first player, and this process continues until the end of the game. The different possible positions at each turn in the game form a branching structure in which each option opens up more and more possibilities.

If you want to program a computer to take one side of a two-player game, one approach is to have the computer follow all the branches in the list of possibilities. Before making its first move, the computer would try every possible choice. For each of these choices, it would then try to determine what its opponent's response would be. To do so, it would follow the same logic: try every possibility and evaluate the possible counterplays. If the computer can look far enough ahead to discover that some move would leave its opponent in a hopeless position, it should make that move.

In theory, this strategy can be applied to any two-player strategy game. In practice, the process of looking at all the possible moves, potential responses, responses to those responses, and so on requires too much time and memory, even for modern computers. There are, however, several games that are simple enough to solve by looking at all the possibilities, yet complex enough so that the solution is not immediately obvious to the human player.

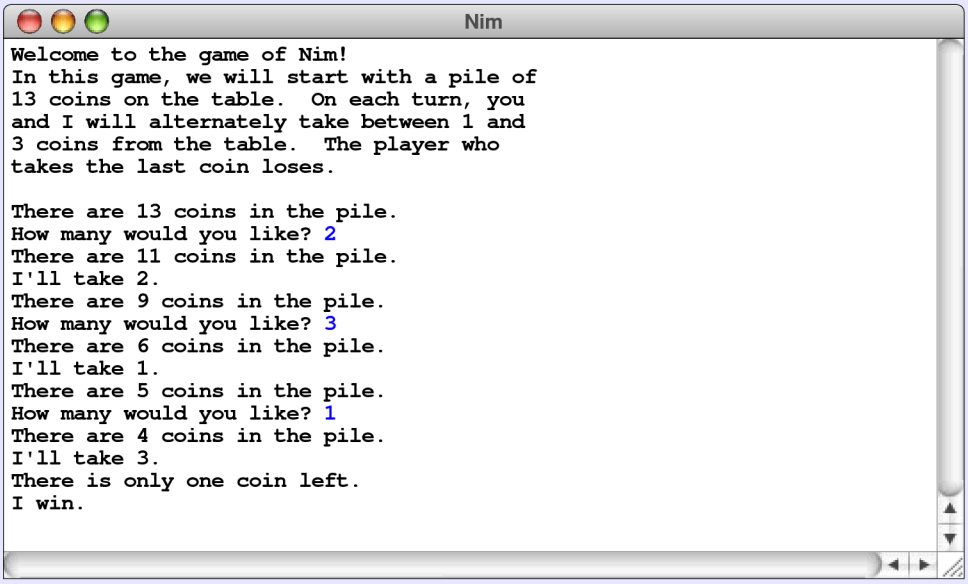
The game of Nim

To see how recursive backtracking applies to two-player games, it helps to consider a simple example such as the game of ***Nim,*** which is the generic name for an entire class of games in which players take turns removing objects from some initial configuration. In this particular version, the game begins with a pile of 13 coins. On each turn, players take either one, two, or three coins from the pile and put them aside. The object of the game is to avoid being forced to take the last coin. Figure 9-4 shows a sample game between the computer and a human player.

How would you go about writing a program to play a winning game of Nim? The mechanical aspects of the game—keeping track of the number of coins, asking the player for a legal move, determining the end of the game, and so forth—are a straightforward programming task. The interesting part of the program consists of figuring out how to give the computer a strategy for playing the best possible game.

Finding a successful strategy for Nim is not particularly hard, particularly if you work backward from the end of the game. The rules of Nim indicate that the loser is the player who takes the last coin. Thus, if you ever find yourself with just one coin on the table, you're in a bad position. You have to take that coin and lose. On the other hand, things look good if you find yourself with two, three, or four coins. In any of these cases, you can always take all but one of the remaining coins, leaving your opponent in the unenviable position of being stuck with just one coin.

FIGURE 9-4 Sample run of the Nim game



But what if there are five coins on the table? What can you do then? After a bit of thought, it's easy to see that you're also doomed if you're left with five coins. No matter what you do, you have to leave your opponent with two, three, or four coins—situations that you've just discovered represent good positions from your opponent's perspective. If your opponent is playing intelligently, you will surely be left with a single coin on your next turn. Since you have no good moves, being left with five coins is clearly a bad position.

This informal analysis reveals an important insight about the game of Nim. On each turn, you are looking for a good move. A good move is one that leaves your opponent in a bad position. But what is a bad position? A bad position is one in which there is no good move.

Even though these definitions of *good move* and *bad position* are circular, they nonetheless constitute a complete strategy for playing a perfect game of Nim. You just have to rely on the power of recursion. If you have a function **`findGoodMove`** that takes the number of coins as its argument, all it has to do is try every possibility, looking for one that leaves a bad position for the opponent. You can then assign the job of determining whether a particular position is bad to the predicate function **`isBadPosition`**, which calls **`findGoodMove`** to see if there is one. The two functions call each other back and forth, evaluating all possible branches as the game proceeds.

The mutually recursive functions **`findGoodMove`** and **`isBadPosition`** provide all the strategy that the Nim program needs to play a perfect game. To complete the program, all you need to do is write the code that takes care of the mechanics of playing Nim with a human player. This code is responsible for setting up the game, printing out instructions, keeping track of whose turn it is, asking the user for a move, checking whether that move is legal, updating the number of coins, figuring out when the game is over, and letting the user know who won.

Although none of these tasks is conceptually difficult, the **`Nim`** application is large enough that it makes sense to adopt the implementation strategy described in section 6.5, in which the program is defined as a class rather than as a collection of free functions. Figure 9-5 shows an implementation of the Nim game that adopts this design. The code for the game is encapsulated in a class called **`SimpleNim`**, along with two instance variables that keep track of the progress of play:

- An integer variable **`nCoins`** that records the number of coins in the pile.
- A variable **`whoseTurn`** that indicates which player is about to move. This value is stored using the enumerated type **`Player`**, which defines the constants **`HUMAN`** and **`COMPUTER`**. At the end of each turn, the code for the **`play`** method passes the turn to the next player by setting **`whoseTurn`** to **`opponent(whoseTurn)`**.

FIGURE 9-5 The `Nim.cpp` implementation

```
/*
 * File: Nim.cpp
 * -------------
 * This program simulates a simple variant of the game of Nim.  In this
 * version, the game starts with a pile of 13 coins on a table.  Players
 * then take turns removing 1, 2, or 3 coins from the pile.  The player
 * who takes the last coin loses.
 */

#include <iostream>
#include <string>
#include "error.h"
#include "simpio.h"
#include "strlib.h"
using namespace std;

/* Constants */

const int N_COINS = 13;       /* Initial number of coins                */
const int MAX_MOVE =  3;      /* Number of coins a player may take      */
const int NO_GOOD_MOVE = -1;  /* Marker indicating there is no good move */

/*
 * Type: Player
 * ------------
 * This enumerated type differentiates the human and computer players.
 */

enum Player { HUMAN, COMPUTER };

/*
 * Method: opponent
 * Usage: Player other = opponent(player);
 * ---------------------------------------
 * Returns the opponent of the player.  The opponent of the computer
 * is the human player and vice versa.
 */

Player opponent(Player player) {
   return (player == HUMAN) ? COMPUTER : HUMAN;
}

/*
 * Constant: STARTING_PLAYER
 * -------------------------
 * Indicates which player should start the game.
 */

const Player STARTING_PLAYER = HUMAN;
```

FIGURE 9-5 The `Nim.cpp` implementation (continued)

```
/*
 * Class: SimpleNim
 * ----------------
 * The SimpleNim class implements the simple version of Nim.
 */

class SimpleNim {

public:

/*
 * Method: play
 * Usage: game.play();
 * -------------------
 * Plays one game of Nim with the human player.
 */

   void play() {
      nCoins = N_COINS;
      whoseTurn = STARTING_PLAYER;
      while (nCoins > 1) {
         cout << "There are " << nCoins << " coins in the pile." << endl;
         if (whoseTurn == HUMAN) {
            nCoins -= getUserMove();
         } else {
            int nTaken = getComputerMove();
            cout << "I'll take " << nTaken << "." << endl;
            nCoins -= nTaken;
         }
         whoseTurn = opponent(whoseTurn);
      }
      announceResult();
   }

/*
 * Method: printInstructions
 * Usage: game.printInstructions();
 * --------------------------------
 * This method explains the rules of the game to the user.
 */

   void printInstructions() {
      cout << "Welcome to the game of Nim!"  << endl;
      cout << "In this game, we will start with a pile of" << endl;
      cout << N_COINS << " coins on the table.  On each turn, you" << endl;
      cout << "and I will alternately take between 1 and" << endl;
      cout << MAX_MOVE << " coins from the table.  The player who" << endl;
      cout << "takes the last coin loses." << endl << endl;
   }
```

FIGURE 9-5 The `Nim.cpp` implementation (continued)

```
private:

/*
 * Method: getComputerMove
 * Usage: int nTaken = getComputerMove();
 * ----------------------------------
 * Figures out what move is best for the computer player and returns
 * the number of coins taken.  The method first calls findGoodMove
 * to see if a winning move exists.  If none does, the program takes
 * only one coin to give the human player more chances to make a mistake.
 */

   int getComputerMove() {
      int nTaken = findGoodMove(nCoins);
      return (nTaken == NO_GOOD_MOVE) ? 1 : nTaken;
   }

/*
 * Method: findGoodMove
 * Usage: int nTaken = findGoodMove(nCoins);
 * ----------------------------------------
 * This method looks for a winning move, given the specified number
 * of coins.  If there is a winning move in that position, the method
 * returns that value; if not, the method returns the constant
 * NO_GOOD_MOVE.  This method depends on the recursive insight that a
 * good move is one that leaves your opponent in a bad position and a bad
 * position is one that offers no good moves.
 */

   int findGoodMove(int nCoins) {
      int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
      for (int nTaken = 1; nTaken <= limit; nTaken++) {
         if (isBadPosition(nCoins - nTaken)) return nTaken;
      }
      return NO_GOOD_MOVE;
   }

/*
 * Method: isBadPosition
 * Usage: if (isBadPosition(nCoins)) . . .
 * ---------------------------------------
 * This method returns true if nCoins is a bad position.
 * A bad position is one in which there is no good move.
 * Being left with a single coin is clearly a bad position
 * and represents the simple case of the recursion.
 */

   bool isBadPosition(int nCoins) {
      if (nCoins == 1) return true;
      return findGoodMove(nCoins) == NO_GOOD_MOVE;
   }
```

☞

FIGURE 9-5 The `Nim.cpp` implementation (continued)

```
/*
 * Method: getUserMove
 * Usage: int nTaken = getUserMove();
 * ----------------------------------
 * Asks the user to enter a move and returns the number of coins taken.
 * If the move is not legal, the user is asked to reenter a valid move.
 */

   int getUserMove() {
      while (true) {
         int nTaken = getInteger("How many would you like? ");
         int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
         if (nTaken > 0 && nTaken <= limit) return nTaken;
         cout << "That's cheating!  Please choose a number";
         cout << " between 1 and " << limit << "." << endl;
         cout << "There are " << nCoins << " coins in the pile." << endl;
      }
   }

/*
 * Method: announceResult
 * Usage: announceResult();
 * ------------------------
 * This method announces the final result of the game.
 */

   void announceResult() {
      if (nCoins == 0) {
         cout << "You took the last coin.  You lose." << endl;
      } else {
         cout << "There is only one coin left." << endl;
         if (whoseTurn == HUMAN) {
            cout << "I win." << endl;
         } else {
            cout << "I lose." << endl;
         }
      }
   }

/* Instance variables */

   int nCoins;              /* Number of coins left on the table   */
   Player whoseTurn;        /* Identifies which player moves next  */

};

/* Main program */

int main() {
   SimpleNim game;
   game.printInstructions();
   game.play();
   return 0;
}
```