Chapter 7
Backtracking Algorithms

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

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

Objectives

- To appreciate how backtracking can be used as a solution strategy.
- To recognize the problem domains for which backtracking strategies are appropriate.
- To understand how recursion applies to backtracking problems.
- To be able to implement recursive solutions to problems involving backtracking.
- To comprehend the minimax strategy as it applies to two-player games.
- To appreciate the importance of developing abstract solutions that can be applied to many different problem domains.

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 results 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.

7.1 Solving a maze by recursive backtracking

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 Daedelus (the engineering genius who later escaped the island 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.

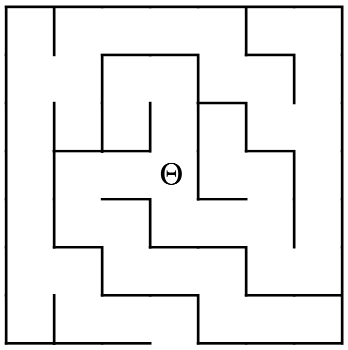
The right-hand rule

Theseus's strategy represents an algorithm for escaping from a maze, but not everyone in such a predicament is lucky enough to have a ball of string or an accomplice clever enough to suggest such an effective approach. Fortunately, there are other strategies for escaping from a maze. Of these strategies, the best known is called the **right-hand rule,** which can be expressed in the following pseudocode form:

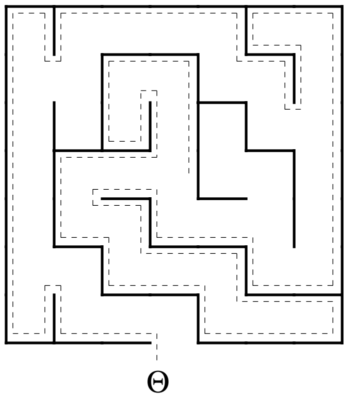
```
Put your right hand against a wall.
while (you have not yet escaped from the maze) {
    Walk forward keeping your right hand on a wall.
}
```

As you walk, the requirement that you keep your right hand touching the wall may force you to turn corners and occasionally retrace your steps. Even so, following the right-hand rule guarantees that you will always be able to find an opening to the outside of any maze.

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):



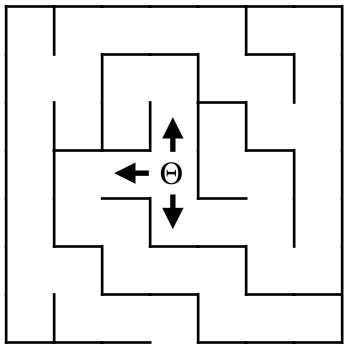
If Theseus puts his right hand on the wall and then follows the right-hand rule from there, he will trace out the path shown by the dashed line in this diagram:



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 position. From the initial configuration, you have three choices, as indicated by the arrows in the following diagram:

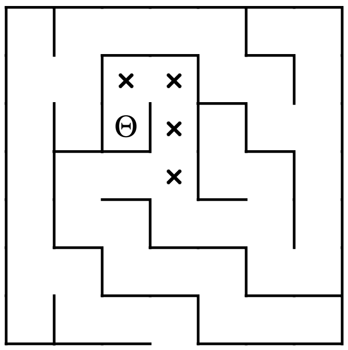


The exit, if any, must lie along one of those paths. Moreover, if you choose the correct direction, you will be one step closer to the solution. The maze has therefore become simpler along that path, which is the key to a recursive solution. This observation suggests the necessary recursive insight. 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 7-1. 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 7-1, 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 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.

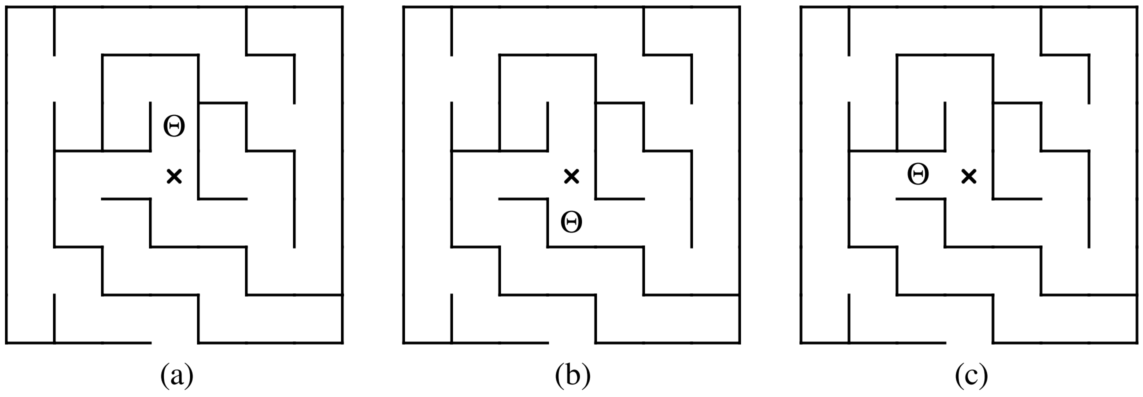
Identifying the simple cases

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.

Figure 7-1 Recursive decomposition of a maze



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 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.

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, for example, 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 put such details aside by introducing a new abstraction layer. The purpose of the abstraction is to provide the main program with access to the information it needs to solve a maze, even though the details are hidden. An interface that provides the necessary functionality is the **`mazelib.h`** interface shown in Figure 7-2.

Once you have access to the **`mazelib.h`** interface, writing a program to solve a maze becomes much simpler. The essence of the problem is to write a function **`SolveMaze`** that uses recursive backtracking to solve a maze whose specific characteristics are maintained by the **`mazelib`** module. The argument to **`SolveMaze`** is 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 must also return some indication of whether it has succeeded. The easiest way to convey this information is to define **`SolveMaze`** as a predicate function that returns **`true`** if a solution has been found, and **`false`** otherwise. Thus, the prototype for **`SolveMaze`** looks like this:

```
bool SolveMaze(pointT pt);
```

Given this definition, the main program is simply

```
int main() {
    ReadMazeMap(MazeFile);
    if (SolveMaze(GetStartPosition())) {
        cout << "The marked squares show a solution path." << endl;
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

Figure 7-2 The `mazelib.h` interface

```
/*
 * File: mazelib.h
 * ---------------
 * This interface provides a library of primitive operations
 * to simplify the solution to the maze problem.
 */

#ifndef _mazelib_h
#define _mazelib_h

#include "genlib.h"

/*
 * Type: directionT
 * ----------------
 * This type is used to represent the four compass directions.
 */

enum directionT { North, East, South, West };

/*
 * Type: pointT
 * ------------
 * The type pointT is used to encapsulate a pair of integer
 * coordinates into a single value with x and y components.
 */

struct pointT {
   int x, y;
};

/*
 * Function: ReadMazeMap
 * Usage: ReadMazeMap(filename);
 * -----------------------------
 * This function reads in a map of the maze from the specified
 * file and stores it in private data structures maintained by
 * this module.  In the data file, 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  |     |
 *       +-+-+-+-+-+
 *
 * Coordinates in the maze are numbered starting at (0,0) in
 * the lower left corner.  The goal is to find a path from
 * the (0,0) square to the exit east of the (4,1) square.
 */

void ReadMazeMap(string filename);
```

```
/*
 * Function: GetStartPosition
 * Usage: pt = GetStartPosition();
 * -------------------------------
 * This function returns a pointT indicating the coordinates of
 * the start square.
 */

pointT GetStartPosition();

/*
 * Function: OutsideMaze
 * Usage: if (OutsideMaze(pt)) . . .
 * ---------------------------------
 * This function returns true if the specified point is outside
 * the boundary of the maze.
 */

bool OutsideMaze(pointT pt);

/*
 * Function: WallExists
 * Usage: if (WallExists(pt, dir)) . . .
 * -------------------------------------
 * This function returns true if there is a wall in the indicated
 * direction from the square at position pt.
 */

bool WallExists(pointT pt, directionT dir);

/*
 * Functions: MarkSquare, UnmarkSquare, IsMarked
 * Usage: MarkSquare(pt);
 *        UnmarkSquare(pt);
 *        if (IsMarked(pt)) . . .
 * ------------------------------
 * These functions mark, unmark, and test the status of the
 * square specified by the coordinates pt.
 */

void MarkSquare(pointT pt);
void UnmarkSquare(pointT pt);
bool IsMarked(pointT pt);

#endif
```

The code for the **`SolveMaze`** function itself turns out to be extremely short and is shown in Figure 7-3. The entire algorithm fits into approximately 10 lines of code with the following pseudocode structure:

```
If the current square is outside the maze, return true to indicate that a solution has been found.
If the current square is marked, return false to indicate that this path has already been tried.
Mark the current square.
for (each of the four compass directions) {
    if (this direction is not  blocked by a wall) {
        Move one step in the indicated direction from the current square.
        Try to solve the maze from there by making a recursive call.
        If this call shows the maze to be solvable, return true to indicate that fact.
    }
}
Unmark the current square.
Return false to indicate that none of the four directions led to a solution.
```

The only function called by **`SolveMaze`** that is not exported by the **`mazelib.h`** interface is the function **`AdjacentPoint(pt, dir)`**, which returns the coordinates of the square that is one step away from **`pt`** in the direction **`dir`**. The following is a simple implementation of **`AdjacentPoint`** that copies the original point and then adjusts the appropriate coordinate value:

Figure 7-3 The `SolveMaze` function

```
/*
 * Function: SolveMaze
 * Usage: if (SolveMaze(pt)) . . .
 * -------------------------------
 * This function attempts to generate a solution to the current
 * maze from point pt.  SolveMaze 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(pointT pt) {
   if (OutsideMaze(pt)) return true;
   if (IsMarked(pt)) return false;
   MarkSquare(pt);
   for (int i = 0; i < 4; i++) {
      directionT dir = directionT(i);
      if (!WallExists(pt, dir)) {
         if (SolveMaze(AdjacentPoint(pt, dir))) {
            return true;
         }
      }
   }
   UnmarkSquare(pt);
   return false;
}
```

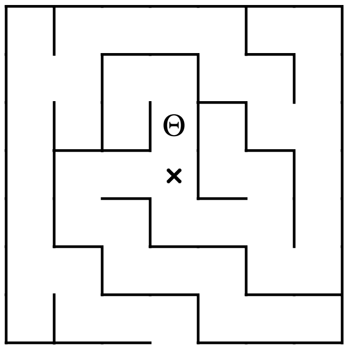
```
pointT AdjacentPoint(pointT pt, directionT dir) {
    pointT newpt = pt;
    switch (dir) {
      case North: newpt.y++; break;
      case East:  newpt.x++; break;
      case South: newpt.y--; break;
      case West:  newpt.x--; break;;
    }
    return newpt;
}
```

The code to unmark the current square at the end of the **for** loop is not strictly necessary in this implementation and in fact can reduce the performance of the algorithm if there are loops in the maze (see exercise 3). The principal advantage of including it is that doing so means that the solution path ends up being recorded by a chain of marked squares from the original starting position to the exit. If you are using a graphical implementation of this algorithm, erasing the marks as you retreat down a path makes it much easier to see the current path.

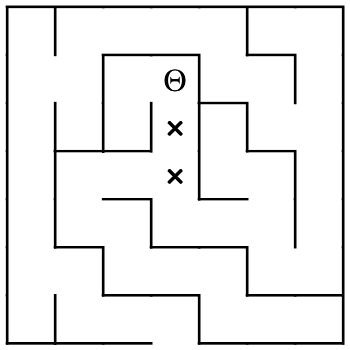
Convincing yourself that the solution works

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 7-3 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 inevitably get lost.

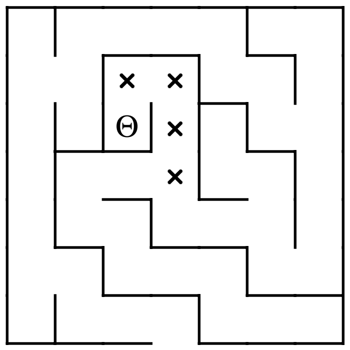
If you are not yet ready to accept the recursive leap of faith, the best you can do is track the operation of the code in a more general sense. You know that the code first tries to solve the maze by moving one square to the north, because the **for** loop goes through the directions in the order defined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



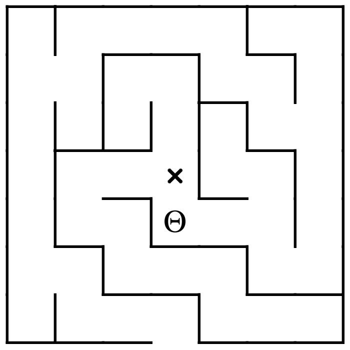
At this point, the same process occurs again. The program again tries to move north and makes a new recursive call in the following position:



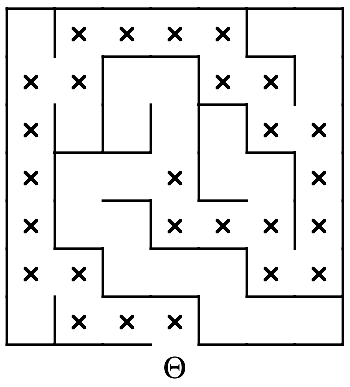
At this level of the recursion, moving north is no longer possible, so the **`for`** loop cycles through the other directions. After a brief excursion southward, upon which the program encounters a marked square, the program finds the opening to the west and proceeds to generate a new recursive call. 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. The simple case kicks in and returns **`true`** to its caller. This value is then propagated back through all 27 levels of the recursion, at which point the original call to **`SolveMaze`** returns to the main program.

7.2 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 simply 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 interesting 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. The word *nim* actually applies to a large 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 in the center of a table. 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 7-4 shows a sample game between the computer and a human player.

Figure 7-4 Sample game of nim

```
Hello.  Welcome to the game of nim.
In this game, we will start with a pile of
13 coins on the table.  On each turn, you
and I will alternately take between 1 and
3 coins from the table.  The player who
takes the last coin loses.

There are 13 coins in the pile.
How many would you like? 2
There are 11 coins in the pile.
I'll take 2.
There are 9 coins in the pile.
How many would you like? 3
There are 6 coins in the pile.
I'll take 1.
There are 5 coins in the pile.
How many would you like? 1
There are 4 coins in the pile.
I'll take 3.
There is only one coin left.
I win.
```

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 state 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. 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 move is good if it 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. Although these definitions of *good move* and *bad position* are circular, they nonetheless constitute a complete strategy for playing a perfect game of nim. All you have to do is 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 `FindGoodMove` function has the following pseudocode formulation:

```
int FindGoodMove(int nCoins) {
    for (each possible move) {
        Evaluate the position that results from making that move.
        If the resulting position is bad, return that move.
    }
    Return a sentinel value indicating that no good move exists.
}
```

The legal values returned by `FindGoodMove` are 1, 2, and 3. The sentinel indicating that no good move exists can be any integer value outside that range. For example, you can define the constant `NO_GOOD_MOVE` as follows:

```
const int NO_GOOD_MOVE = -1;
```

The code for `FindGoodMove` then looks like this:

```
int FindGoodMove(int nCoins) {
   for (int nTaken = 1; nTaken <= MAX_MOVE; nTaken++) {
       if (IsBadPosition(nCoins - nTaken)) return nTaken;
   }
   return NO_GOOD_MOVE;
}
```

The code for the `IsBadPosition` function is even easier. After checking for the simple case that occurs when there is only a single coin to take, the function simply calls `FindGoodMove` to see if a good move exists. The code for `IsBadPosition` is therefore simply

```
bool IsBadPosition(int nCoins) {
   if (nCoins == 1) return true;
   return FindGoodMove(nCoins) == NO_GOOD_MOVE;
}
```

This function encapsulates the following ideas:

- Being left with a single coin indicates a bad position.
- A position is bad if there are no good moves.

The functions `FindGoodMove` and `IsBadPosition` provide all the strategy that the nim program needs to play a perfect game. The rest of the program just takes care of the mechanics of playing nim with a human player, as shown in Figure 7-5.

A generalized program for two-player games

The code for nim shown in Figure 7-5 is highly specific to that game. The `FindGoodMove` function, for example, is written so that it incorporates directly into the structure of the code the knowledge that the computer may take one, two, or three coins. The basic idea, however, is far more general. Many two-player games can be solved using the same overall strategy, even though different games clearly require different codings of the details.

Figure 7-5 Program to play the game of nim

```
/*
 * 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.  This simulation allows a human player to compete
 * against the computer.
 */

#include "genlib.h"
#include "simpio.h"
#include <iostream>

/*
 * Constants
 * ---------
 * N_COINS       -- Initial number of coins
 * MAX_MOVE      -- The maximum number of coins a player may take
 * NO_GOOD_MOVE  -- Sentinel indicating no good move is available
 */

const int N_COINS = 13;
const int MAX_MOVE =  3;
const int NO_GOOD_MOVE = -1;

/*
 * Type: playerT
 * -------------
 * This enumeration type distinguishes the turns for the human
 * player from those for the computer.
 */

enum playerT { Human, Computer };

/* Private function prototypes */

void GiveInstructions();
void AnnounceWinner(int nCoins, playerT whoseTurn);
int GetUserMove(int nCoins);
bool MoveIsLegal(int nTaken, int nCoins);
int ChooseComputerMove(int nCoins);
int FindGoodMove(int nCoins);
bool IsBadPosition(int nCoins);
```

```
/*
 * Main program
 * ------------
 * This program plays the game of nim.  In this implementation,
 * the human player always goes first.
 */

int main() {
   int nCoins, nTaken;
   playerT whoseTurn;

   GiveInstructions();
   nCoins = N_COINS;
   whoseTurn = Human;
   while (nCoins > 1) {
      cout << "There are " << nCoins << " coins in the pile."<<endl;
      switch (whoseTurn) {
        case Human:
         nTaken = GetUserMove(nCoins);
         whoseTurn = Computer;
         break;
        case Computer:
         nTaken = ChooseComputerMove(nCoins);
         cout << "I'll take " << nTaken << "." << endl;
         whoseTurn = Human;
         break;
      }
      nCoins -= nTaken;
   }
   AnnounceWinner(nCoins, whoseTurn);
   return 0;
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * --------------------------
 * This function explains the rules of the game to the user.
 */

void GiveInstructions() {
   cout << "Hello.  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. " << endl;
   cout << "On each turn, you" << endl;
   cout << "and I will alternately take between 1 and" << endl;
   cout << MAX_MOVE << " coins from the table." << endl;
   cout << "The player who" << endl;
   cout << "takes the last coin loses." << endl;
   cout << endl;
}
```

```
/*
 * Function: AnnounceWinner
 * Usage: AnnounceWinner(nCoins, whoseTurn);
 * -----------------------------------------
 * This function announces the final result of the game.
 */

void AnnounceWinner(int nCoins, playerT whoseTurn) {
   if (nCoins == 0) {
      cout << "You took the last coin.  You lose." << endl;
   } else {
      cout << "There is only one coin left." << endl;
      switch (whoseTurn) {
        case Human:    cout << "I win." << endl; break;
        case Computer: cout << "I lose." << endl; break;
      }
   }
}

/*
 * Function: GetUserMove
 * Usage: nTaken = GetUserMove(nCoins);
 * ------------------------------------
 * This function is responsible for the human player's turn.
 * It takes the number of coins left in the pile as an argument,
 * and returns the number of coins that the player removes
 * from the pile.  The function checks the move for legality
 * and gives the player repeated chances to enter a legal move.
 */

int GetUserMove(int nCoins) {
   int nTaken, limit;

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

/*
 * Function: MoveIsLegal
 * Usage: if (MoveIsLegal(nTaken, nCoins)) . . .
 * ---------------------------------------------
 * This predicate function returns true if it is legal to take
 * nTaken coins from a pile of nCoins.
 */

bool MoveIsLegal(int nTaken, int nCoins) {
   return (nTaken > 0 && nTaken <= MAX_MOVE && nTaken <= nCoins);
}
```

```
/*
 * Function: ChooseComputerMove
 * Usage: nTaken = ChooseComputerMove(nCoins);
 * -------------------------------------------
 * This function figures out what move is best for the computer
 * player and returns the number of coins taken.  The function
 * 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 ChooseComputerMove(int nCoins) {
   int nTaken = FindGoodMove(nCoins);
   if (nTaken == NO_GOOD_MOVE) nTaken = 1;
   return nTaken;
}

/*
 * Function: FindGoodMove
 * Usage: nTaken = FindGoodMove(nCoins);
 * -------------------------------------
 * This function looks for a winning move, given the specified
 * number of coins.  If there is a winning move in that
 * position, the function returns that value; if not, the
 * function returns the constant NoWinningMove.  This function
 * 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) {
   for (int nTaken = 1; nTaken <= MAX_MOVE; nTaken++) {
      if (IsBadPosition(nCoins - nTaken)) return nTaken;
   }
   return NO_GOOD_MOVE;
}

/*
 * Function: IsBadPosition
 * Usage: if (IsBadPosition(nCoins)) . . .
 * ---------------------------------------
 * This function 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;
}
```

One of the principal ideas in this text is the notion of **abstraction,** which is the process of separating out the general aspects of a problem so that they are no longer obscured by the details of a specific domain. You may not be terribly interested in a program that plays nim; after all, nim is rather boring once you figure it out. What you would probably enjoy more is a program that is general enough to be adapted to play nim, or tic-tac-toe, or any other two-player strategy game you choose.

The first step in creating such a generalization lies in recognizing that there are several concepts that are common to all games. The most important such concept is **state.** For any game, there is some collection of data that defines exactly what is happening at any point in time. In the nim game, for example, the state consists of the number of coins on the table and whose turn it is to move. For a game like chess, the state would instead include what pieces were currently on which squares. Whatever the game, however, it should be possible to combine all the relevant data together into a single record structure and then refer to it using a single variable. Another common concept is that of a **move.** In nim, a move consists of an integer representing the number of coins taken away. In chess, a move might consist of a pair indicating the starting and ending coordinates of the piece that is moving, although this approach is in fact complicated by the need to represent various esoteric moves like castling or the promotion of a pawn. In any case, a move can also be represented by a single structure that includes whatever information is appropriate to that particular game. The process of abstraction consists partly of defining these concepts as general types, with names like `stateT` and `moveT`, that transcend the details of any specific game. The internal structure of these types will be different for different games, but the abstract algorithm can refer to these concepts in a generic form.

Consider, for example, the following main program, which comes from the tic-tac-toe example introduced in Figure 7-6 at the end of this chapter:

```
int main() {
    GiveInstructions();
    stateT state = NewGame();
    moveT move;
    while (!GameIsOver(state)) {
        DisplayGame(state);
        switch (WhoseTurn(state)) {
          case Human:
            move = GetUserMove(state);
            break;
          case Computer:
            move = ChooseComputerMove(state);
            DisplayMove(move);
            break;
        }
        MakeMove(state, move);
    }
    AnnounceResult(state);
    return 0;
}
```

At this level, the program is easy to read. It begins by giving instructions and then calls `NewGame` to initialize a new game, storing the result in the variable `state`. It then goes into a loop, taking turns for each side until the game is over. On the human player's turns, it calls a function `GetUserMove` to read in the appropriate move from the user. On its own turns, the program calls `ChooseComputerMove`, which has the task of finding the best move in a particular state. Once the move has been determined by one of these two functions, the main program then calls `MakeMove`, which updates the state of the game to show that the indicated move has been made and that it is now the other player's turn. At the end, the program displays the result of the game and exits.

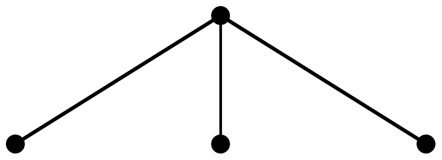
It is important to notice that the main program gives no indication whatsoever about what the actual game is. It could just as easily be nim or chess as tic-tac-toe. Each game requires its own definitions for `stateT`, `moveT`, and the various functions like `GiveInstructions`, `MakeMove`, and `GameIsOver`. Even so, the implementation of the main program as it appears here is general enough to work for many different games.

The minimax strategy

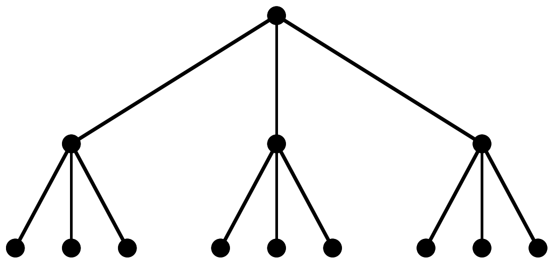
The main program, however, is hardly the most interesting part of a game. The real challenge consists of providing the computer with an effective strategy. In the general program for two-player games, the heart of the computer's strategy is the function `FindBestMove`, which is called by the function `ChooseComputerMove` in the main program. Given a particular state of the game, the role of `FindBestMove` is to return the optimal move in that position.

From the discussion of nim earlier in this chapter, you should already have some sense of what constitutes an optimal move. The best move in any position is simply the one that leaves your opponent in the worst position. The worst position is likewise the one that offers the weakest best move. This idea—finding the position that leaves your opponent with the worst possible best move—is called the **minimax** strategy because the goal is to find the move that minimizes your opponent's maximum opportunity.

The best way to visualize the operation of the minimax strategy is to think about the possible future moves in a game as forming a branching diagram that expands on each turn. Because of their branching character, such diagrams are called **game trees.** The current state is represented by a dot at the top of the game tree. If there are, for example, three possible moves from this position, there will be three lines emanating down from the current state to three new states that represent the results of these moves, as shown in the following diagram:



For each of these new positions, your opponent will also have options. If there are again three options from each of these positions, the next generation of the game tree looks like this:

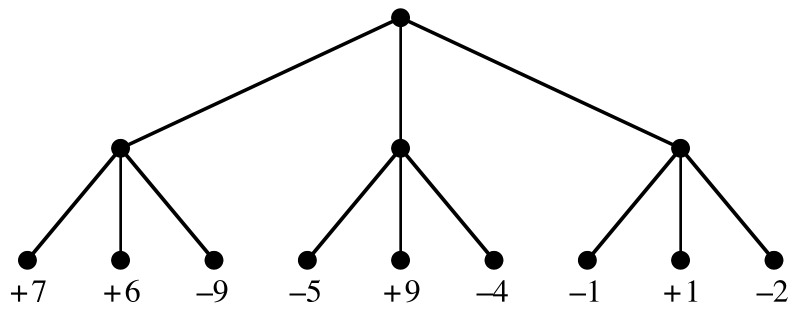


Which move do you choose in the initial position? Clearly, your goal is to achieve the best outcome. Unfortunately, you only get to control half of the game. If you were able to select your opponent's move as well as your own, you could select the path to the state two turns away that left you in the best position. Given the fact that your opponent is also trying to win, the best thing you can do is choose the initial move that leaves your opponent with as few winning chances as possible.

In order to get a sense of how you should proceed, it helps to add some quantitative data to the analysis. Determine whether a particular move is better than some alternative is much easier if it is possible to assign a numeric score to each possible move. The higher the numeric score, the better the move. Thus, a move that had a score of +7, for example, is better than a move with a rating of –4. In addition to rating each possible move, it makes sense to assign a similar numeric rating to each position in the game. Thus, one position might have a rating of +9 and would therefore be better than a position with a score of only +2.

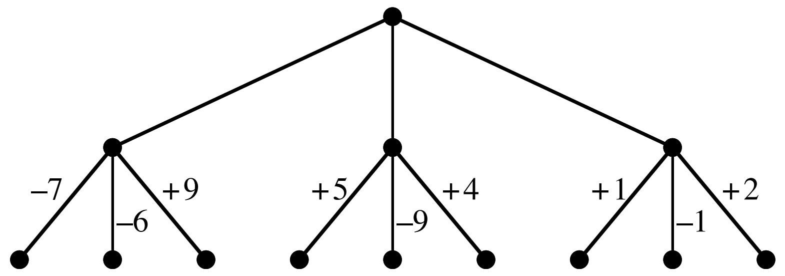
Both positions and moves are rated from the perspective of the player having the move. Moreover, the rating system is designed to be symmetric around 0, in the sense that a position that has a score of +9 for the player to move would have a score of –9 from the opponent's point of view. This interpretation of rating numbers captures the idea that a position that is good for one player is therefore a bad position for the opponent, as you saw in the discussion of the Nim game earlier in this chapter. More importantly, defining the rating system in this way makes it easy to express the relationship between the scores for moves and positions. The score for any move is simply the negative of the score for the resulting position when rated by your opponent. Similarly, the rating of any position can be defined as the rating of its best move.

To make this discussion more concrete, let's consider a simple example. Suppose that you have looked two steps ahead in the game, covering one move by you and the possible responses from your opponent. In computer science, a single move for a single player is called a **ply** to avoid the ambiguity associated with the words *move* and *turn,* which sometimes imply that both players have a chance to play. If you rate the positions at the conclusion of the two-ply analysis, the game tree might look like this:



Because the positions at the bottom of this tree are again positions in which—as at the top of the tree—you have to move, the rating numbers in those positions are assigned from your perspective. Given these ratings of the potential positions, what move should you make from the original configuration? At first glance, you might be attracted by the fact that the leftmost branch has the most positive total score or that the center one contains a path that leads to a +9, which is an excellent outcome for you. Unfortunately, none of these considerations matter much if your opponent is playing rationally. If you choose the leftmost branch, your opponent will surely take the rightmost option from there, which leaves you with a –9 position. The same thing happens if you choose the center branch; your opponent finds the worst possible position, which has a rating of –5. The best you can do is choose the rightmost branch, which only allows your opponent to end up with a –2 rating. While this position is hardly ideal, it is better for you than the other outcomes.

The situation becomes easier to follow if you add the ratings for each of your opponent's responses at the second level of the tree. The rating for a move—from the perspective of the player making it—is the negative of the resulting position. Thus, the move ratings from your opponent's point of view look like this:



In these positions, your opponent will seek to play the move with the best score. By choosing the rightmost path, you minimize the maximum score available to your opponent, which is the essence of the *minimax* strategy.

Implementing the minimax algorithm

The minimax algorithm is quite general and can be implemented in a way that does not depend on the specific characteristics of the game. In many respects, the implementation of the minimax algorithm is similar to the strategy section in **`nim.cpp`** because it consists of two mutually recursive functions, one that finds the best move and another than evaluates the quality of a position. If you want to make your algorithm as general as possible, however, you must modify the approach used in the nim program to accommodate the following extensions:

- *It must be possible to limit the depth of the recursive search.* For games that involve any significant level of complexity, it is impossible to search the entire game tree in a reasonable amount of time. If you try to apply this approach to chess, for example, a program running on the fastest computers available would require many times the lifetime of the universe to make the first move. As a result, a practical implementation of the minimax algorithm must include a provision for cutting off the search at a certain point. One possible approach is to limit the depth of the recursion to a certain number of moves. You could, for example, allow the recursion to proceed until each player had made three moves and then evaluate the position at that point using some nonrecursive approach.
- *It must be possible to assign ratings to moves and positions.* Every position in nim is either good or bad; there are no other options. In a more complex game, it is necessary—particularly if you can't perform a complete analysis—to assign ratings to positions and moves so that the algorithm has a standard for comparing them against other possibilities. The rating scheme used in this implementation assigns integers to positions and moves. Those integers extend in both the positive and the negative direction and are centered on zero, which means that a rating of –5 for one player is equivalent to a rating of +5 from the opponent's point of view. Zero is therefore the neutral rating and is given the name **`NeutralPosition`**. The maximum positive rating is the constant **`WinningPosition`**, which indicates a position in which the player whose turn it is to move will invariably win; the corresponding extreme in the negative direction is **`LosingPosition`**, which indicates that the player will always lose.

Taking these general considerations into account requires some changes in the design of the mutually recursive functions that implement the minimax algorithm, which are called **`FindBestMove`** and **`EvaluatePosition`**. Both functions take the state of the game as an argument, but each also requires the current depth of the recursion so that the recursive search can be restricted if necessary. Moreover, in order to avoid a considerable amount of redundant calculation, it is extremely useful if **FindBestMove** can return a rating along with the best move, so it uses a reference parameter along with the return value to return the two pieces of information.. Given these design decisions, the prototypes for **FindBestMove** and **EvaluatePosition** look like this:

```
moveT FindBestMove(stateT state, int depth, int & rating);
int EvaluatePosition(stateT state, int depth);
```

The strategy for **FindBestMove** can be expressed using the following pseudocode:

```
moveT FindBestMove(stateT state, int depth, int & rating) {
    for (each possible move or until you find a forced win) {
        Make the move.
        Evaluate the resulting position, adding one to the depth indicator.
        Keep track of the minimum rating so far, along with the corresponding move.
        Retract the move to restore the original state.
    }
    Store the move rating into the reference parameter.
    Return the best move.
}
```

The corresponding implementation, which follows this pseudocode outline, looks like this:

```
moveT FindBestMove(stateT state, int depth, int & rating) {
    Vector<moveT> moveList;
    GenerateMoveList(state, moveList);
    int nMoves = moveList.size();
    if (nMoves == 0) Error("No moves available");
    moveT bestMove;
    int minRating = WINNING_POSITION + 1;
    for (int i = 0; i < nMoves && minRating != LOSING_POSITION; i++) {
        moveT move = moveList[i];
        MakeMove(state, move);
        int curRating = EvaluatePosition(state, depth + 1);
        if (curRating < minRating) {
            bestMove = move;
            minRating = curRating;
        }
        RetractMove(state, move);
    }
    rating = -minRating;
    return bestMove;
}
```

The function **GenerateMoveList(state, moveArray)** is implemented separately for each game and has the effect of filling the elements in **moveArray** with a list of the legal moves in the current position; the result of the function is the number of available moves. The only other parts of this function that require some comment are the line

```
minRating = WinningPosition + 1;
```

which initializes the value of **minRating** to a number large enough to guarantee that this value will be replaced on the first cycle through the **for** loop, and the line

```
rating = -minRating;
```

which stores the rating of the best move in the reference parameter. The negative sign is included because the perspective has shifted: the positions were evaluated from the point-of-view of your opponent, whereas the ratings express the value of a move from your own point of view. A move that leaves your opponent with a negative position is good for you and therefore has a positive value.

The **`EvaluatePosition`** function is considerably simpler. The simple cases that allow the recursion to terminate occur when the game is over or when the maximum allowed recursive depth has been achieved. In these cases, the program must evaluate the current state as it exists without recourse to further recursion. This evaluation is performed by the function **`EvaluateStaticPosition`**, which is coded separately for each game. In the general case, however, the rating of a position is simply the rating of the best move available, given the current state. Thus, the following code is sufficient to the task:

```
int EvaluatePosition(stateT state, int depth) {
    int rating;
    if (GameIsOver(state) || depth >= MAX_DEPTH) {
        return EvaluateStaticPosition(state);
    }
    FindBestMove(state, depth, rating);
    return rating;
}
```

Using the general strategy to solve a specific game

The minimax strategy embodied in the functions **`FindBestMove`** and **`EvaluatePosition`** takes care of the conceptually complicated work of finding the best move from a given position. Moreover, because it is written in an abstract way, the code does not depend on the details of a particular game. Once you have these functions, the task of coding a new two-player strategy game is reduced to the problem of designing **`moveT`** and **`stateT`** structures that are appropriate to the game and then writing code for the functions that must be supplied independently for each game.

The long program in Figure 7-6 illustrates how to use the general minimax facility to construct a program that plays tic-tac-toe. The implementations of the main program and the functions **`FindBestMove`** and **`EvaluatePosition`** are completely independent of the details of tic-tac-toe, even though they are responsible for calculating most of the strategy. What makes this program play tic-tac-toe is the definition of the types and functions that surround the basic framework.

The types **`moveT`** and **`stateT`** are defined so that they are appropriate for tic-tac-toe. If you number the squares in the board like this:

1	2	3
4	5	6
7	8	9

a move can be represented as a single integer, which means that the appropriate definition for the type **`moveT`** is simply

```
typedef int moveT;
```

Figure 7-6 Program to play the game of tic-tac-toe

```
/*
 * File: tictactoe.cpp
 * -------------------
 * This program plays a game of tic-tac-toe with the user.  The
 * code is designed to make it easy to adapt the general structure
 * to other games.
 */

#include "genlib.h"
#include "simpio.h"
#include "vector.h"
#include "grid.h"
#include <iostream>

/*
 * Constants: WINNING_POSITION, NEUTRAL_POSITION, LOSING_POSITION
 * --------------------------------------------------------------
 * These constants define a rating system for game positions.  A
 * rating is an integer centered at 0 as the neutral score: ratings
 * greater than 0 are good for the current player, ratings less than
 * 0 are good for the opponent.  The constants WINNING_POSITION and
 * LOSING_POSITION are opposite in value and indicate a position that
 * is a forced win or loss, respectively.  In a game in which the
 * analysis is complete, no intermediate values ever arise.  If the
 * full tree is too large to analyze, the EvaluatePosition function
 * returns integers that fall between the two extremes.
 */

const int WINNING_POSITION  = 1000;
const int NEUTRAL_POSITION  = 0;
const int LOSING_POSITION   = -WINNING_POSITION;

/*
 * Type: playerT
 * -------------
 * This type is used to distinguish the human and computer
 * players and keep track of who has the current turn.
 */

enum playerT { Human, Computer };

/*
 * Type: moveT
 * -----------
 * For any particular game, the moveT type must keep track of the
 * information necessary to describe a move.  For tic-tac-toe,
 * a moveT is simply an integer identifying the number of one of
 * the nine squares, as follows:
 *
 *        1 | 2 | 3
 *       ---+---+---
 *        4 | 5 | 6
 *       ---+---+---
 *        7 | 8 | 9
 */

typedef int moveT;
```

```
/*
 * Type: stateT
 * ------------
 * For any game, the stateT structure records the current state of
 * the game.  As in Chapter 4, the tic-tac-toe board is represented
 * using a Grid<char>; the elements must be either 'X', 'O', or ' '.
 * In addition to the board array, the code stores a playerT value
 * to indicate whose turn it is.  In this game, the stateT structure
 * also contains the total number of moves so that functions can
 * find this value without counting the number of occupied squares.
 */

struct stateT {
   Grid<char> board;
   playerT whoseTurn;
   int turnsTaken;
};

/*
 * Constant: MAX_DEPTH
 * -------------------
 * This constant indicates the maximum depth to which the recursive
 * search for the best move is allowed to proceed.
 */

const int MAX_DEPTH = 10000;

/*
 * Constant: FIRST_PLAYER
 * ----------------------
 * This constant indicates which player goes first.
 */

const playerT FIRST_PLAYER = Computer;

/* Private function prototypes */

void GiveInstructions();
moveT FindBestMove(stateT state, int depth, int & pRating);
int EvaluatePosition(stateT state, int depth);
stateT NewGame();
void DisplayGame(stateT state);
void DisplayMove(moveT move);
char PlayerMark(playerT player);
moveT GetUserMove(stateT state);
moveT ChooseComputerMove(stateT state);
void GenerateMoveList(stateT state, Vector<moveT> & moveList);
bool MoveIsLegal(moveT move, stateT state);
void MakeMove(stateT & state, moveT move);
void RetractMove(stateT & state, moveT move);
bool GameIsOver(stateT state);
void AnnounceResult(stateT state);
playerT WhoseTurn(stateT state);
playerT Opponent(playerT player);
int EvaluateStaticPosition(stateT state);
bool CheckForWin(stateT state, playerT player);
bool CheckForWin(Grid<char> & board, char mark);
bool CheckLine(Grid<char> & board, char mark,
               int row, int col, int dRow, int dCol);
```

```
/*
 * Main program
 * ------------
 * The main program, along with the functions FindBestMove and
 * EvaluatePosition, are general in their design and can be
 * used with most two-player games.  The specific details of
 * tic-tac-toe do not appear in these functions and are instead
 * encapsulated in the stateT and moveT data structures and a
 * a variety of subsidiary functions.
 */

int main() {
   GiveInstructions();
   stateT state = NewGame();
   moveT move;
   while (!GameIsOver(state)) {
      DisplayGame(state);
      switch (WhoseTurn(state)) {
        case Human:
          move = GetUserMove(state);
          break;
        case Computer:
          move = ChooseComputerMove(state);
          DisplayMove(move);
          break;
      }
      MakeMove(state, move);
   }
   AnnounceResult(state);
   return 0;
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * --------------------------
 * This function gives the player instructions about how to
 * play the game.
 */

void GiveInstructions() {
   cout << "Welcome to tic-tac-toe.  The object of the game" << endl;
   cout << "is to line up three symbols in a row," << endl;
   cout << "vertically, horizontally, or diagonally." << endl;
   cout << "You'll be " << PlayerMark(Human) << " and I'll be "
        << PlayerMark(Computer) << "." << endl;
}
```

```
/*
 * Function: NewGame
 * Usage: state = NewGame();
 * -------------------------
 * This function starts a new game and returns a stateT that
 * has been initialized to the defined starting configuration.
 */

stateT NewGame() {
   stateT state;
   state.board.resize(3, 3);
   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
         state.board[i][j] = ' ';
      }
   }
   state.whoseTurn = FIRST_PLAYER;
   state.turnsTaken = 0;
   return state;
}

/*
 * Function: DisplayGame
 * Usage: DisplayGame(state);
 * --------------------------
 * This function displays the current state of the game.
 */

void DisplayGame(stateT state) {
   if (GameIsOver(state)) {
      cout << "The final position looks like this:" << endl << endl;
   } else {
      cout << "The game now looks like this:" << endl << endl;
   }
   for (int i = 0; i < 3; i++) {
      if (i != 0) cout << "---+---+---" << endl;
      for (int j = 0; j < 3; j++) {
         if (j != 0) cout << "|";
         cout << " " << state.board[i][j] << " ";
      }
      cout << endl;
   }
   cout << endl;
}

/*
 * Function: DisplayMove
 * Usage: DisplayMove(move);
 * -------------------------
 * This function displays the computer's move.
 */

void DisplayMove(moveT move) {
   cout << "I'll move to " << move << endl;
}
```

```
/*
 * Function: FindBestMove
 * Usage: move = FindBestMove(state, depth, pRating);
 * --------------------------------------------------
 * This function finds the best move for the current player, given
 * the specified state of the game.  The depth parameter and the
 * constant MAX_DEPTH are used to limit the depth of the search
 * for games that are too difficult to analyze in full detail.
 * The function returns the best move by storing an integer in
 * the variable to which pRating points.
 */

moveT FindBestMove(stateT state, int depth, int & rating) {
   Vector<moveT> moveList;
   GenerateMoveList(state, moveList);
   int nMoves = moveList.size();
   if (nMoves == 0) Error("No moves available");
   moveT bestMove;
   int minRating = WINNING_POSITION + 1;
   for (int i = 0; i < nMoves && minRating != LOSING_POSITION; i++) {
      moveT move = moveList[i];
      MakeMove(state, move);
      int curRating = EvaluatePosition(state, depth + 1);
      if (curRating < minRating) {
         bestMove = move;
         minRating = curRating;
      }
      RetractMove(state, move);
   }
   rating = -minRating;
   return bestMove;
}

/*
 * Function: EvaluatePosition
 * Usage: rating = EvaluatePosition(state, depth);
 * -----------------------------------------------
 * This function evaluates a position by finding the rating of
 * the best move in that position.  The depth parameter and the
 * constant MAX_DEPTH are used to limit the depth of the search.
 */

int EvaluatePosition(stateT state, int depth) {
   int rating;
   if (GameIsOver(state) || depth >= MAX_DEPTH) {
      return EvaluateStaticPosition(state);
   }
   FindBestMove(state, depth, rating);
   return rating;
}
```

```
/*
 * Function: NewGame
 * Usage: state = NewGame();
 * -------------------------
 * This function starts a new game and returns a stateT that
 * has been initialized to the defined starting configuration.
 */

stateT NewGame() {
    stateT state;

    for (int i = 1; i <= 9; i++) {
        state.board[i] = ' ';
    }
    state.whoseTurn = FIRST_PLAYER;
    state.turnsTaken = 0;
    return state;
}

/*
 * Function: DisplayGame
 * Usage: DisplayGame(state);
 * --------------------------
 * This function displays the current state of the game.
 */

void DisplayGame(stateT state) {
    if (GameIsOver(state)) {
        cout << "The final position looks like this:" << endl << endl;
    } else {
        cout << "The game now looks like this:" << endl << endl;
    }
    for (int row = 0; row < 3; row++) {
        if (row != 0) cout << "---+---+---" << endl;
        for (int col = 0; col < 3; col++) {
            if (col != 0) cout << "|";
            cout << " " << state.board[row * 3 + col + 1] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

/*
 * Function: DisplayMove
 * Usage: DisplayMove(move);
 * -------------------------
 * This function displays the computer's move.
 */

void DisplayMove(moveT move) {
    cout << "I'll move to " << move << endl;
}
```

```
/*
 * Function: PlayerMark
 * Usage: mark = PlayerMark(player);
 * ---------------------------------
 * This function returns the mark used on the board to indicate
 * the specified player.  By convention, the first player is
 * always X, so the mark used for each player depends on who
 * goes first.
 */

char PlayerMark(playerT player) {
   if (player == FIRST_PLAYER) {
      return 'X';
   } else {
      return 'O';
   }
}

/*
 * Function: GetUserMove
 * Usage: move = GetUserMove(state);
 * ---------------------------------
 * This function allows the user to enter a move and returns the
 * number of the chosen square.  If the user specifies an illegal
 * move, this function gives the user the opportunity to enter
 * a legal one.
 */

moveT GetUserMove(stateT state) {
   cout << "Your move." << endl;
   moveT move;
   while (true) {
      cout << "What square? ";
      move = GetInteger();
      if (MoveIsLegal(move, state)) break;
      cout << "That move is illegal.  Try again." << endl;
   }
   return move;
}

/*
 * Function: ChooseComputerMove
 * Usage: move = ChooseComputerMove(state);
 * ----------------------------------------
 * This function chooses the computer's move and is primarily
 * a wrapper for FindBestMove.  This function also makes it
 * possible to display any game-specific messages that need
 * to appear at the beginning of the computer's turn.  The
 * rating value returned by FindBestMove is simply discarded.
 */

moveT ChooseComputerMove(stateT state) {
   int rating;
   cout << "My move." << endl;
   return FindBestMove(state, 0, rating);
}
```

```
/*
 * Function: GenerateMoveList
 * Usage: GenerateMoveList(state, moveList);
 * -----------------------------------------
 * This function generates a list of the legal moves available in
 * the specified state.  The list of moves is returned in the
 * vector moveList, which must be allocated by the client.
 */

void GenerateMoveList(stateT state, Vector<moveT> & moveList) {
   for (int i = 1; i <= 9; i++) {
      moveT move = moveT(i);
      if (MoveIsLegal(move, state)) {
         moveList.add(moveT(i));
      }
   }
}

/*
 * Function: MoveIsLegal
 * Usage: if (MoveIsLegal(move, state)) . . .
 * ------------------------------------------
 * This function returns true if the specified move is legal.
 */

bool MoveIsLegal(moveT move, stateT state) {
   if (move < 1 || move > 9) return false;
   int row = (move - 1) / 3;
   int col = (move - 1) % 3;
   return state.board[row][col] == ' ';
}

/*
 * Function: MakeMove
 * Usage: MakeMove(state, move);
 * -----------------------------
 * This function changes the state by making the indicated move.
 */

void MakeMove(stateT & state, moveT move) {
   int row = (move - 1) / 3;
   int col = (move - 1) % 3;
   state.board[row][col] = PlayerMark(state.whoseTurn);
   state.whoseTurn = Opponent(state.whoseTurn);
   state.turnsTaken++;
}
```

```
/*
 * Function: RetractMove
 * Usage: RetractMove(state, move);
 * --------------------------------
 * This function changes the state by "unmaking" the indicated move.
 */

void RetractMove(stateT & state, moveT move) {
   int row = (move - 1) / 3;
   int col = (move - 1) % 3;
   state.board[row][col] = ' ';
   state.whoseTurn = Opponent(state.whoseTurn);
   state.turnsTaken--;
}

/*
 * Function: GameIsOver
 * Usage: if (GameIsOver(state)) . . .
 * -----------------------------------
 * This function returns true if the game is complete.
 */

bool GameIsOver(stateT state) {
   return (state.turnsTaken == 9
           || CheckForWin(state, state.whoseTurn)
           || CheckForWin(state, Opponent(state.whoseTurn)));
}

/*
 * Function: AnnounceResult
 * Usage: AnnounceResult(state);
 * -----------------------------
 * This function announces the result of the game.
 */

void AnnounceResult(stateT state) {
   DisplayGame(state);
   if (CheckForWin(state, Human)) {
      cout << "You win." << endl;
   } else if (CheckForWin(state, Computer)) {
      cout << "I win." << endl;
   } else {
      cout << "Cat's game." << endl;
   }
}
```

```
/*
 * Function: WhoseTurn
 * Usage: player = WhoseTurn(state);
 * ---------------------------------
 * This function returns whose turn it is, given the current
 * state of the game.  The reason for making this a separate
 * function is to ensure that the common parts of the code do
 * not need to refer to the internal components of the state.
 */

playerT WhoseTurn(stateT state) {
   return state.whoseTurn;
}

/*
 * Function: Opponent
 * Usage: opp = Opponent(player);
 * ------------------------------
 * This function returns the playerT value corresponding to the
 * opponent of the specified player.
 */

playerT Opponent(playerT player) {
   return (player == Human) ? Computer : Human;
}

/*
 * Function: EvaluateStaticPosition
 * Usage: rating = EvaluateStaticPosition(state);
 * ----------------------------------------------
 * This function gives the rating of a position without looking
 * ahead any further in the game tree.  Although this function
 * duplicates much of the computation of GameIsOver and therefore
 * introduces some runtime inefficiency, it makes the algorithm
 * somewhat easier to follow.
 */

int EvaluateStaticPosition(stateT state) {
   if (CheckForWin(state, state.whoseTurn)) {
      return WINNING_POSITION;
   }
   if (CheckForWin(state, Opponent(state.whoseTurn))) {
      return LOSING_POSITION;
   }
   return NEUTRAL_POSITION;
}
```

```
/*
 * Function: CheckForWin
 * Usage: if (CheckForWin(state, player)) . . .
 * -------------------------------------------
 * This function returns true if the specified player has won
 * the game.  The check on turnsTaken increases efficiency,
 * because neither player can win the game until the fifth move.
 */

bool CheckForWin(stateT state, playerT player) {
   if (state.turnsTaken < 5) return false;
   return CheckForWin(state.board, PlayerMark(player));
}

/*
 * Checks to see whether the specified player identified by mark
 * ('X' or 'O') has won the game.  To reduce the number of special
 * cases, this implementation uses the helper function CheckLine
 * so that the row, column, and diagonal checks are the same.
 */

bool CheckForWin(Grid<char> & board, char mark) {
   for (int i = 0; i < 3; i++) {
      if (CheckLine(board, mark, i, 0, 0, 1)) return true;
      if (CheckLine(board, mark, 0, i, 1, 0)) return true;
   }
   if (CheckLine(board, mark, 0, 0, 1, 1)) return true;
   return CheckLine(board, mark, 2, 0, -1, 1);
}

/*
 * Checks a line extending across the board in some direction.
 * The starting coordinates are given by the row and col
 * parameters.  The direction of motion is specified by dRow
 * and dCol, which show how to adjust the row and col values
 * on each cycle.  For rows, dRow is always 0; for columns,
 * dCol is 0.  For diagonals, the dRow and dCol values will
 * be +1 or -1 depending on the direction of the diagonal.
 */

bool CheckLine(Grid<char> & board, char mark, int row, int col,
               int dRow, int dCol) {
   for (int i = 0; i < 3; i++) {
      if (board[row][col] != mark) return false;
      row += dRow;
      col += dCol;
   }
   return true;
}
```

The state of the tic-tac-toe game is determined by the symbols on the nine squares of the board and whose turn it is to move. Thus, you can represent the essential components of the state as follows:

```
struct stateT {
    char board[(3 * 3) + 1];
    playerT whoseTurn;
};
```

The board is represented as a single-dimensional array to correspond with the numbering of the squares from 1 to 9; the fact that position 0 is unused makes it necessary to add an extra element to the array. As the complete program shows, adding a **turnsTaken** component to the state makes it much easier to determine whether the game is complete.

The rules of tic-tac-toe are expressed in the implementation of functions like **GenerateMoveList**, **MoveIsLegal**, and **EvaluateStaticPosition**, each of which is coded specifically for this game. Even though these functions require some thought, they tend to be less conceptually complex than those that implement the minimax strategy. By coding the difficult functions once and then reusing them in various different game programs, you can avoid having to go through the same logic again and again.

Summary

In this chapter, you have learned to solve problems that require making a sequence of choices as you search for a goal, as illustrated by finding a path through a maze or a winning strategy in a two-player game. The basic strategy is to write programs that can backtrack to previous decision points if those choices lead to dead ends. By exploiting the power of recursion, however, you can avoid coding the details of the backtracking process explicitly and develop general solution strategies that apply to a wide variety of problem domains.

Important points in this chapter include:

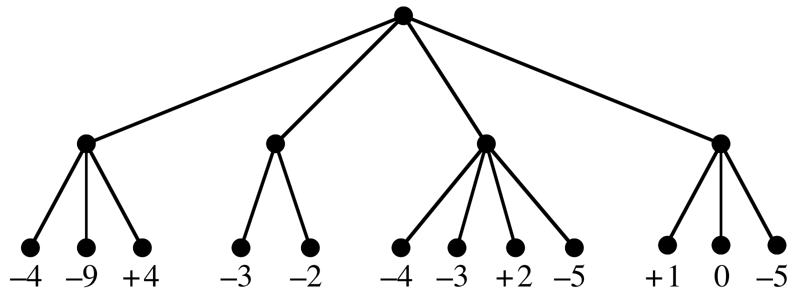
- You can solve most problems that require backtracking by adopting the following recursive approach:

```
If you are already at a solution, report success.
for (every possible choice in the current position ) {
    Make that choice and take one step along the path.
    Use recursion to solve the problem from the new position.
    If the recursive call succeeds, report the success to the next higher level.
    Back out of the current choice to restore the state at the beginning of the loop.
}
Report failure.
```

- The complete history of recursive calls in a backtracking problem—even for relatively simple applications—is usually too complex to understand in detail. For problems that involve any significant amount of backtracking, it is essential to accept the recursive leap of faith.
- You can often find a winning strategy for two-player games by adopting a recursive-backtracking approach. Because the goal in such games involves minimizing the winning chances for your opponent, the conventional strategic approach is called the *minimax algorithm*.
- It is possible to code the minimax algorithm in a general way that keeps the details of any specific game separate from the implementation of the minimax strategy itself. This approach makes it easier to adapt an existing program to new games.

Review questions

1. What is the principal characteristic of a backtracking algorithm?
2. Using your own words, state the right-hand rule for escaping from a maze. Would a left-hand rule work equally well?
3. What is the insight that makes it possible to solve a maze by recursive backtracking?
4. What are the simple cases that apply in the recursive implementation of **`SolveMaze`**?
5. Why is important to mark squares as you proceed through the maze? What would happen in the **`SolveMaze`** function if you never marked any squares?
6. What is the purpose of the **`UnmarkSquare`** call at the end of the **`for`** loop in the **`SolveMaze`** implementation? Is this statement essential to the algorithm?
7. What is the purpose of the Boolean result returned by **`SolveMaze`**?
8. In your own words, explain how the backtracking process actually takes place in the recursive implementation of **`SolveMaze`**.
9. In the simple nim game, the human player plays first and begins with a pile of 13 coins. Is this a good or a bad position? Why?
10. Write a simple C++ expression based on the value of **`nCoins`** that has the value **`true`** if the position is good for the current player and **`false`** otherwise. (Hint: Use the **`%`** operator.)
11. What is the minimax algorithm? What does its name signify?
12. Suppose you are in a position in which the analysis for the next two moves shows the following rated outcomes from your opponent's point-of-view:



    If you adopt the minimax strategy, what is the best move to make in this position? What is the rating of that move from your perspective?

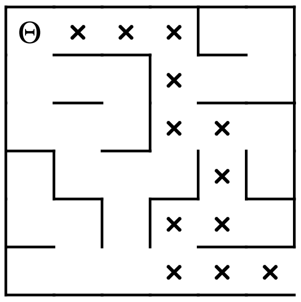
13. Why is it useful to develop an abstract implementation of the minimax algorithm?
14. What two data structures are used to make the minimax implementation independent of the specific characteristics of a particular game?
15. What is the role of each of the three arguments to the **`FindBestMove`** function?
16. Explain the role of the **`EvaluateStaticPosition`** function in the minimax implementation.

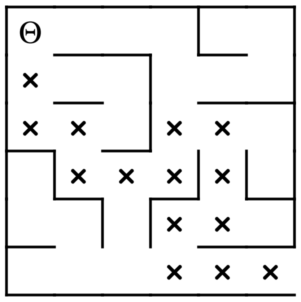
Programming exercises

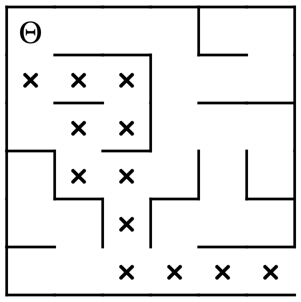
1. The **`SolveMaze`** function shown in Figure 7-3 implements a recursive algorithm for solving a maze but is not complete as it stands. The solution depends on several functions exported by the **`mazelib.h`** interface, which is specified in Figure 7-2 but never actually implemented.

   Write a file **`mazelib.cpp`** that implements this interface. Your implementation should store the data representing the maze as part of the private state of the module. This design requires you to declare static global variables within the implementation that are appropriate to the data you need to implement the operations. If you design the data structure well, most of the individual function definitions in the interface are quite simple. The hard parts of this exercise are designing an appropriate internal representation and implementing the function **`ReadMapFile`**, which initializes the internal structures from a data file formatted as described in the interface documentation.

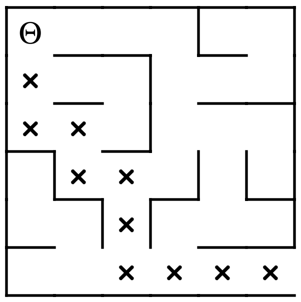
2. In many mazes, there are multiple paths. For example, the diagrams below show three solutions for the same maze:


length = 13


length = 15


length = 13

None of these solutions, however, is optimal. The shortest path through the maze has a path length of 11:



Write a function

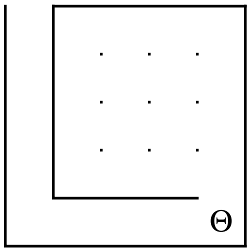
```
int ShortestPathLength(pointT pt);
```

that returns the length of the shortest path in the maze from the specified position to any exit. If there is no solution to the maze, **`ShortestPathLength`** should return the constant **`NO_SOLUTION`**, which is defined to have a value larger than the maximum permissible path length, as follows:

```
const int NO_SOLUTION = 10000;
```

3. As implemented in Figure 7-3, the **SolveMaze** function unmarks each square as it discovers there are no solutions from that point. Although this design strategy has the advantage that the final configuration of the maze shows the solution path as a series of marked squares, the decision to unmark squares as you backtrack has a cost in terms of the overall efficiency of the algorithm. If you've marked a square and then backtracked through it, you've already explored the possibilities leading from that square. If you come back to it by some other path, you might as well rely on your earlier analysis instead of exploring the same options again.

   To give yourself a sense of how much these unmarking operations cost in terms of efficiency, extend the **SolveMaze** program so that it records the number of recursive calls as it proceeds. Use this program to calculate how many recursive calls are required to solve the following maze if the call to **UnmarkSquare** remains part of the program:



   Run your program again, this time without the call to **UnmarkSquare**. What happens to the number of recursive calls?

4. As the result of the preceding exercise makes clear, the idea of keeping track of the path through a maze by using the **MarkSquare** facility in **mazelib.h** has a substantial cost. A more practical approach is to change the definition of the recursive function so that it keeps track of the current path as it goes. Following the logic of **SolveMaze**, write a function

```
bool FindPath(pointT pt, Vector<pointT> path);
```

   that takes, in addition to the coordinates of the starting position, a vector of **pointT** values called **path**. Like **SolveMaze**, **FindPath** returns a Boolean value indicating whether the maze is solvable. In addition, **FindPath** initializes the elements of the **path** vector to a sequence of coordinates beginning with the starting position and ending with the coordinates of the first square that lies outside the maze. For example, if you use **FindPath** with the following main program

```
int main() {
    ReadMazeMap(MazeFile);
    Vector<pointT> path;
    if (FindPath(GetStartPosition(), path)) {
        cout << "The following path is a solution:" << endl;
        for (int i = 0; i < path.size(); i++) {
            cout << "(" << path[i].x << ", "
                        << path[i].y << ")" << endl;
        }
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

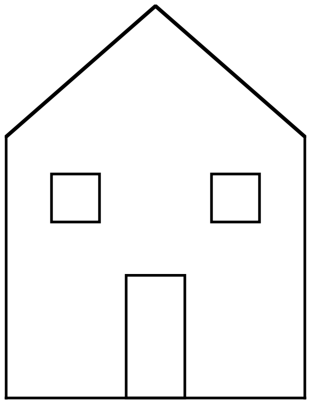
   it will display the coordinates of the points in the solution path on the screen.

For this exercise, it is sufficient for **`FindPath`** to find any solution path. It need not find the shortest one.

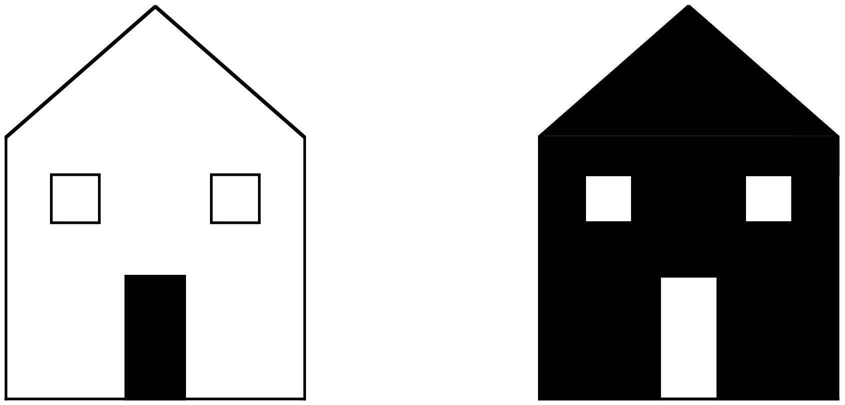
5. If you have access to the graphics library described in Chapter 6, extend the **`maze.cpp`** program so that, in addition to keeping track of the internal data structures for the maze, it also displays a diagram of the maze on the screen and shows the final solution path.

6. Most drawing programs for personal computers make it possible to fill an enclosed region on the screen with a solid color. Typically, you invoke this operation by selecting a paint-bucket tool and then clicking the mouse, with the cursor somewhere in your drawing. When you do, the paint spreads to every part of the picture it can reach without going through a line.

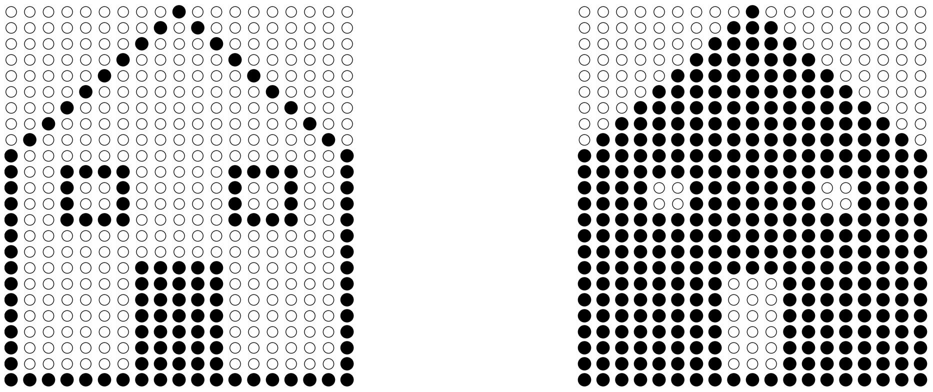
   For example, suppose you have just drawn the following picture of a house:



   If you select the paint bucket and click inside the door, the drawing program fills the area bounded by the door frame as shown at the left side of the following diagram. If you instead click somewhere on the front wall of the house, the program fills the entire wall space except for the windows and doors, as shown on the right:



   In order to understand how this process works, it is important to understand that the screen of the computer is actually broken down into an array of tiny dots called **pixels.** On a monochrome display, pixels can be either white or black. The paint-fill operation consists of painting black the starting pixel (i.e., the pixel you click while using the paint-bucket tool) along with any pixels connected to that starting point by an unbroken chain of white pixels. Thus, the patterns of pixels on the screen representing the preceding two diagrams would look like this:



Write a program that simulates the operation of the paint-bucket tool. To simplify the problem, assume that you have access to the enumerated type

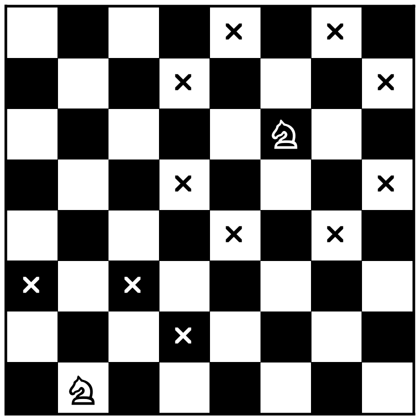
```
enum pixelStateT { WHITE, BLACK };
```

and the following functions:

```
pixelStateT GetPixelState(pointT pt);
void SetPixelState(pointT pt, pixelStateT state);
bool OutsidePixelBounds(pointT pt);
```

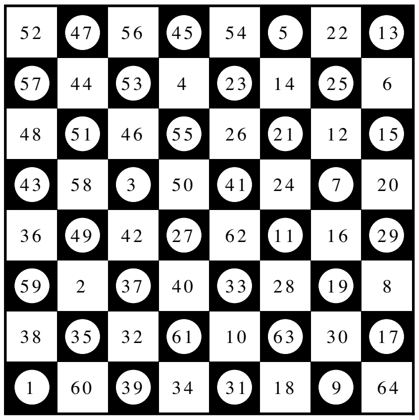
The first function is used to return the state of any pixel, given its coordinates in the pixel array. The second allows you to change the state of any pixel to a new value. The third makes it possible to determine whether a particular coordinate is outside the pixel array altogether, so that the recursion can stop at the edges of the screen.

7. In chess, a knight moves in an L-shaped pattern: two squares in one direction horizontally or vertically, and then one square at right angles to that motion. For example, the knight in the upper right side of the following diagram can move to any of the eight squares marked with a black cross:



The mobility of a knight decreases near the edge of the board, as illustrated by the bottom knight, which can reach only the three squares marked by white crosses.

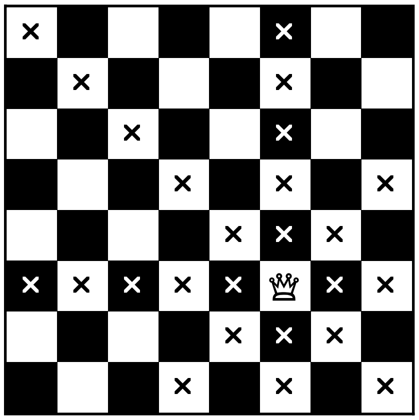
It turns out that a knight can visit all 64 squares on a chessboard without ever moving to the same square twice. A path for the knight that moves through all the squares without repeating a square is called a **knight’s tour.** One such tour is shown in the following diagram, in which the numbers in the squares indicate the order in which they were visited:



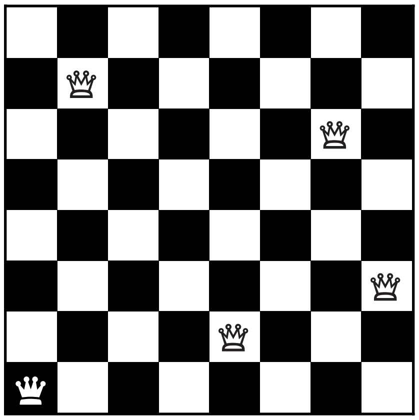
52	47	56	45	54	5	22	13
57	44	53	4	23	14	25	6
48	51	46	55	26	21	12	15
43	58	3	50	41	24	7	20
36	49	42	27	62	11	16	29
59	2	37	40	33	28	19	8
38	35	32	61	10	63	30	17
1	60	39	34	31	18	9	64

Write a program that uses backtracking recursion to find a knight's tour.

8. The most powerful piece in the game of chess is the queen, which can move any number of squares in any direction, horizontally, vertically, or diagonally. For example, the queen shown in this chessboard can move to any of the marked squares:



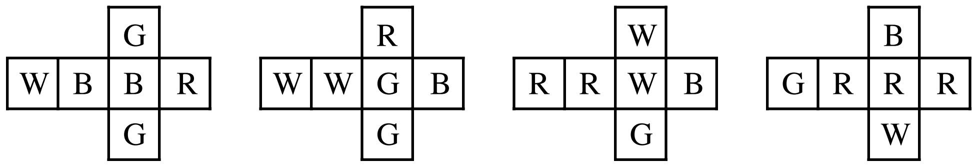
Even though the queen can cover a large number of squares, it is possible to place eight queens on a 8x8 chessboard so that none of them attacks any of the others, as shown in the following diagram:



Write a program that solves the more general problem of whether it is possible to place *N* queens on an *N*x*N* chessboard so that none of them can move to a square occupied by any of the others in a single turn. Your program should either display a solution if it finds one or report that no solutions exist.

9. In the 1960s, a puzzle called *Instant Insanity* was popular for some years before it faded from view. The puzzle consisted of four cubes whose faces were each painted with one of the colors red, blue, green, and white, represented in the rest of this problem by their initial letter. The goal of the puzzle was to arrange the cubes into a line so that if you looked at the line from any of its edges, you would see no duplicated colors.

   Cubes are hard to draw in two dimensions, but the following diagram shows what the cubes would look like if you unfolded them and placed them flat on the page:

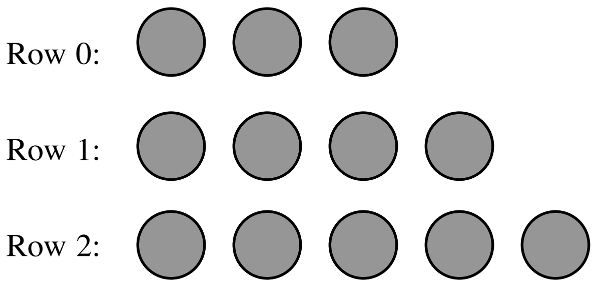


   Write a program that uses backtracking to solve the Instant Insanity puzzle.

10. Rewrite the simple nim game from Figure 7-5 so that it uses the more general structure developed for tic-tac-toe. Your new program should not change the implementations of the functions **`main`**, **`FindBestMove`**, and **`EvaluatePosition`**, which should remain exactly as they appear in Figure 7-6. Your job is to come up with appropriate definitions of **`stateT`**, **`moveT`**, and the various game-specific functions in the implementation so that the program plays nim instead of tic-tac-toe.

11. Modify the code for the simple nim game you wrote for exercise 10 so that it plays a different variant of nim. In this version, the pile begins with 17 coins. On each turn, players alternate taking one, two, three, or four coins from the pile. In the simple nim game, the coins the players took away were simply ignored; in this game, the coins go into a pile for each player. The player whose pile contains an even number of coins after the last one is taken wins the game.

12. In the most common variant of nim, the coins are not combined into a single pile but are instead arranged in three rows like this:



   A move in this game consists of taking any number of coins, subject to the condition that all the coins must come from the same row. The player who takes the last coin loses.

   Write a program that uses the minimax algorithm to play a perfect game of three-pile nim. The starting configuration shown here is a typical one, but your program should be general enough so that you can easily change the number of coins in each row.