Exercise 15.3

Implement a version of a Reversi/Othello board game. Each player can be either a human or the computer. Focus on getting the program correct and (then) getting the computer player smart enough to be worth playing against.

Hint: Perhaps the most "fun" exercise in this book. Requires a solid understanding of classes, expressions, and statements, and a little familiarity with the standard library (mostly input/output). No use of templates or exceptions.

Let's quickly review the rules of this deceptively simple game. It is played on an eight-by-eight grid of squares. One player plays black, the other plays white; black always opens the game. A move consists of placing a disc of your own color onto the board. Each straight-line sequence of pieces of the opposite color between your new piece and another piece of your own color then turns to your color. Figure 15.3 shows an example of this mechanism.

The gray piece indicates the next move of black. As you can see, this move causes three white pieces to turn black because they were trapped between the new black piece and previous black pieces. If there is no position in which you can capture a piece of your opponent, you must give up your turn. If the board is full or if neither player can capture a piece, the game terminates and the winner is the player who has the most pieces on the board. The initial state of the board is depicted in Figure 15.4.

In order to implement this game, it is useful to examine what the natural objects are: the pieces, the board, the players. These concepts will probably map to some user-defined types. The pieces are presumably quite simple—they are either black or white. The board could be represented as an array of states: black, white, or empty. Clearly, these states are close relatives of the pieces, and from an implementation point of view it may be more convenient to encode both in the same type, say:

```
enum Color { kEmpty = 0, kWhite, kBlack };
```

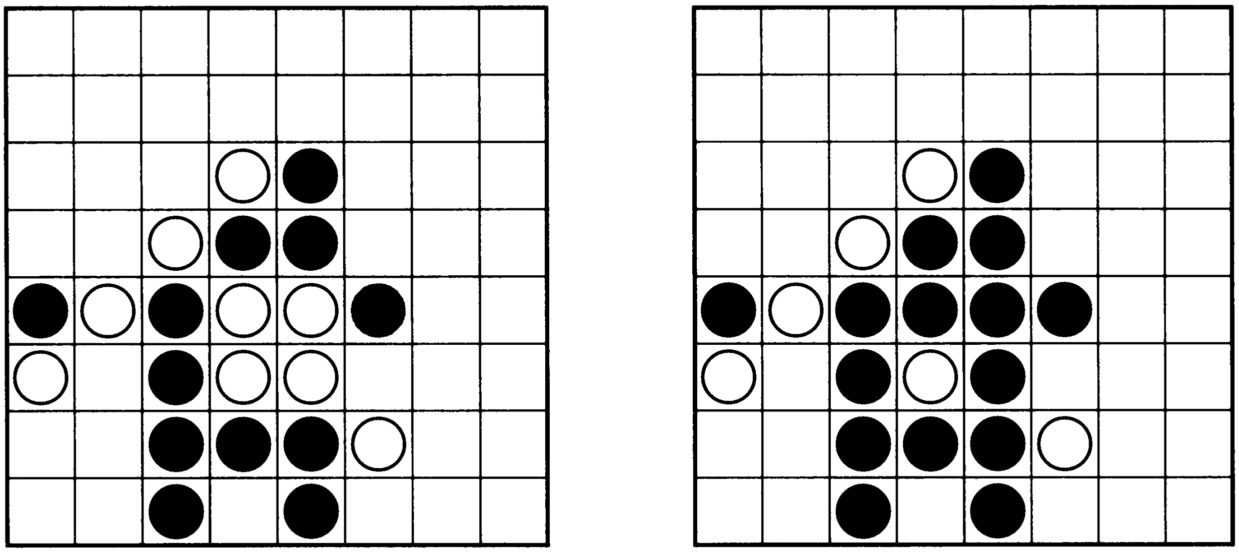

Figure 15.3 *Sample Reversi move: black plays the gray position*

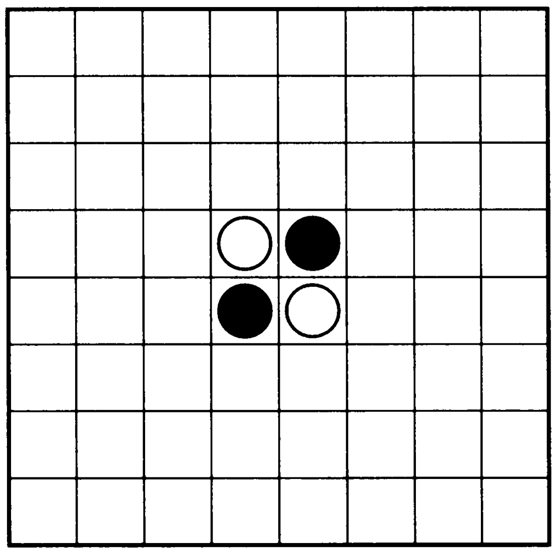

Figure 15.4 *Initial state of the board in a game of Reversi*

A board could then be defined as a two-dimensional array:

```
typedef Color Board[8][8];
```

A player is assigned a `Color` and is then repeatedly requested to make (or skip) a move until the game terminates. Hence, we could specify an abstract `Player` type (§12.3) as follows:

```
struct Player {
   virtual ~Player() {}
   virtual void get_move(Board const&,
                         int &row, int &col) = 0;
   virtual void skip_move(Board const&) {}
};
```

From this type, we could derive a `HumanTextPlayer` that implements `get_move` by taking input from `cin` and a `VirtualTextPlayer` that uses a simple algorithm to determine a move and reports it to `cout`. Other derivations that, for example, accept input from a graphical interface could also be developed.

We do not have all the elements to bring movement in the game yet. We need an entity that coordinates it all. This could be as simple as a function, but in the spirit of object orientation, let's map it onto a type `Game`:

```
struct Game {
   Game(Player *black, Player *white);
   void play();
private:
   // To be determined
};
```

We could place all the rule enforcing and administrative responsibilities in this type, but that would defeat our goal of trying to decompose a larger problem into smaller ones. Here are some of the responsibilities we may want to investigate further:

- Who initializes the board?
- Who draws the board at each stage of the game?
- Who keeps the score?
- Who determines whether a move is valid?
- Who handles situations in which a player must skip a turn?

Initializing the board sounds like the job of the Board constructor. However, our tentative Board type does not have a user-defined constructor. Hence, it may be worthwhile to make the board a class of its own. By making the board a full-fledged class, we open the door to adding more intelligence to it. For example, rather than treat the board type as an array in which we must validate all the moves and flip the colors when necessary, we can just "ask the board to attempt a move." If the move is valid, the board flips all necessary colors; if not, the move has no effect on the board state. The success of the move can be indicated by a conventional return value. Given such an approach, you may agree that the board is also the best place to keep the scores:

```
enum MoveResult { kUnsuccessful = 0, kSuccessful };

struct Board {
   Board();
   Color const& operator()(int r, int c) const
      { return b_[r][c]; }
   MoveResult move(int r, int c, Color color);
   int score(Color color) const
      { return color==kWhite? white_score_: black_score_; }
private:
   Color b_[8][8];
   int black_score_, white_score_;
};
```

I also decided to add an operator that provides a read-only interface to the state of each grid position. This is in preparation for the next question: Who draws the board at each step of the game?

You might take the view that this should also be the responsibility of the board. Because we do not want to exclude either a textual or a graphical interface, we could make Board an abstract class with a pure virtual member draw(). A derivation would then take care of rendering the board appropriately. However, what if you decide later that the same board can be viewed in multiple ways simultaneously? For example, one of the viewers could be a logging facility to a printer. The inheritance model could be restrictive in that case. Instead, I prefer having a separate BoardViewer abstract base class that only encapsulates a pointer to a Board (which remains a concrete class). This separation of a model (the board) and its viewer turns out to be an essential foundation of modern user-interface design. We will place a BoardViewer under the control of the Game object.

The above is but a sketch and needs considerable development.

```
// File: Board.H
#ifndef BOARD_H
#define BOARD_H

#include <assert.h>

enum MoveResult { kUnsuccessful = 0, kSuccessful };
enum Color { kEmpty = 0, kWhite, kBlack };

inline Color opponent_of(Color color) {
   assert((color==kWhite) or (color==kBlack));
   return color==kWhite? kBlack: kWhite;
}

struct Board {
   Board();
   Color operator()(int r, int c) const { return b_[r][c]; }
   void move(Color, int r, int c);
   MoveResult get_possible_move(Color,
                                int &r, int &c) const;
   MoveResult valid_move(Color, int r, int c) const;
   int score(Color color) const
      { return color==kWhite? white_score_: black_score_; }
   enum { kNRows = 8, kNCols = 8 };
private:
   Color b_[kNRows+2][kNCols+2];
   int black_score_, white_score_;
};
#endif
```

The refined Board class—whose interface is placed in a file with include guards (§9.3.3)—encapsulates a slightly larger array to allow for a layer of empty "guard positions" around the actual board position, as illustrated in Figure 15.5.

These guard positions simplify the tests that validate moves. There is no need to verify that a running index stays within bounds. For example, the loop

```
while (b_[row][col]==opponent)
```

will fail when hitting the boundary of the grid because then b_[row][col]==kEmpty. The enumerators kNRows and kNCols could conceivably be modified to experiment with non-traditional board sizes.

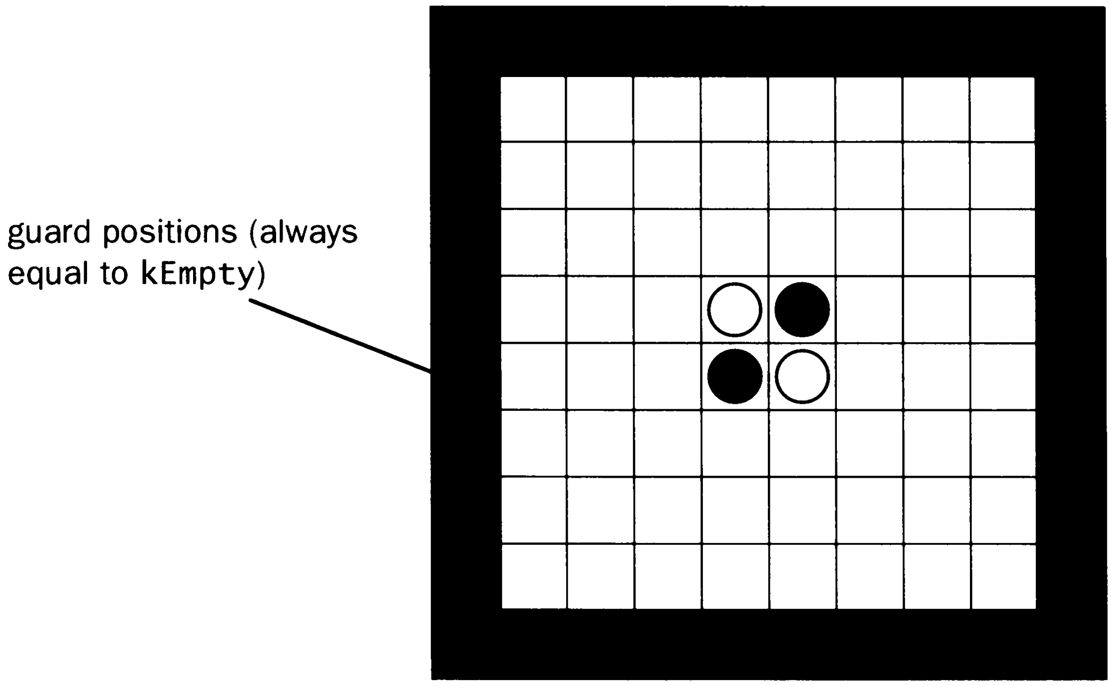

Figure 15.5 *Empty guard positions around the board*

I also added a public member function get_possible_move() that will, for example, help the Game object determine when a player must skip a turn. This function takes a color as its first argument. If a valid move exists for this color, kSuccessful is returned, and the r and c parameter are set to a valid move. Otherwise, kUnsuccessful is returned.

```
// File: Board.C
#include <assert.h>
#include "Board.H"

Board::Board() {
   for (int r = 0; r!=kNRows+2; ++r)
      for (int c = 0; c!=kNRows+2; ++c)
         b_[r][c] = kEmpty;
   b_[kNRows/2+1][kNRows/2]
      = b_[kNRows/2][kNRows/2+1] = kBlack;
   b_[kNRows/2][kNRows/2]
      = b_[kNRows/2+1][kNRows/2+1] = kWhite;
   black_score_ = white_score_ = 2;
}
```

The constructor initializes the board to contain only the initial center pieces.

The following member checks a move—a color and a position—for validity. To do this, the function starts off at the given (r, c) position and tries all eight possible directions to see if a line of consecutive opponent's pieces followed by an own piece exists. The eight directions are described by (dr, dc) pairs generated by the following code:

```
    for (int dr = -1; dr!=2; ++dr)
       for (int dc = -1; dc!=2; ++dc)
          if ((dr!=0) or (dc!=0)) {
             // ...
          }
```

Note how, in fact, nine pairs are generated, but the pair (`dr==0`, `dc==0`) is skipped because adding that pair to a position would not actually get us anywhere.

```
 MoveResult Board::valid_move(Color color,
                              int r, int c) const {
    assert((r>=1) and (r<=kNRows)
       and (c>=1) and (c<=kNCols));
    if (b_[r][c]!=kEmpty)
       return kUnsuccessful;
    Color opponent = opponent_of(color);
    for (int dr = -1; dr!=2; ++dr)
       for (int dc = -1; dc!=2; ++dc)
          if ((dr!=0) or (dc!=0)) {
             int row = r, col = c;
             do {
                row += dr;
                col += dc;
             } while (b_[row][col]==opponent);
             if ((b_[row][col]==color)
             and ((row!=r+dr) or (col!=c+dc)))
                return kSuccessful;
          }
    return kUnsuccessful;
 }
```

The next member function is similar but actually changes the color of the captured pieces. Note that the assertion '`assert(valid_move(color, r, c)`' is expensive, but it can be disabled by `#define`-ing the preprocessor symbol `NDEBUG` (§24.3.7.2).

```
 void Board::move(Color color ,int r, int c) {
    assert(valid_move(color, r, c));
    Color opponent = opponent_of(color);
    int count = 0;
    for (int dr = -1; dr!=2; ++dr)
       for (int dc = -1; dc!=2; ++dc)
          if ((dr!=0) or (dc!=0)) {
             int row = r, col = c;
             do {
                row += dr;
                col += dc;
             } while (b_[row][col]==opponent);
```

```
                // If this direction captures something,
                // change the color of the captured pieces:
                if ((b_[row][col]==color)
                and ((row!=r+dr) or (col!=c+dc)))
                   for (row = r+dr, col = c+dc;
                        b_[row][col]==opponent;
                        row += dr, col += dc) {
                      b_[row][col] = color;
                      ++count;
                   }
          }
   b_[r][c] = color;
   if (color==kBlack) {
      black_score_ += count+1;
      white_score_ -= count;
   } else {
      white_score_ += count+1;
      black_score_ -= count;
   }
}

MoveResult Board::get_possible_move(Color color,
                                    int &r, int &c) const {
   for (r = 1; r!=kNRows+1; ++r)
      for (c = 1; c!=kNCols+1; ++c)
         if (valid_move(color, r, c))
            return kSuccessful;
   return kUnsuccessful;
}
```

Let's add the code that determines the interface and the implementation of the "board-rendering system" (a big phrase for a little piece of code, but in the software industry using such phrases has occasionally proven to be successful):

```
// File: BoardViewer.H
#ifndef BOARDVIEWER_H
#define BOARDVIEWER_H

struct Board;
struct BoardViewer {
   virtual void draw(Board&) = 0;
};
#endif
```

The header file above defines an abstract class that introduces a simple interface. In this exercise, we'll limit ourselves to coding a single concrete implementation that renders a given board on `cout`.

```
// File: BoardTextViewer.H
#ifndef BOARDTEXTVIEWER_H
#define BOARDTEXTVIEWER_H

#include "BoardViewer.H"
#include <iostream>

using std::ostream;

struct BoardTextViewer: BoardViewer {
   BoardTextViewer(ostream &output): output_(output) {}
   virtual ~BoardTextViewer() {}
   virtual void draw(Board&);
private:
   ostream &output_;
};
#endif


// File: BoardTextViewer.C
#include "Board.H"
#include "BoardTextViewer.H"
#include <assert.h>

void BoardTextViewer::draw(Board &board) {
   assert((Board::kNCols<10) && (Board::kNRows<27));
   int r, c;
   output_ << "\n   ";
   for (c = 1; c!=Board::kNCols+1; ++c)
      output_ << ' ' << c;
   output_ << "\n  .-";
   for (c = 1; c!=Board::kNCols+1; ++c)
      output_ << "--";
   for (r = 1; r!=Board::kNRows+1; ++r) {
      output_ << "\n " << char('a'+r-1) << '|';
      for (c = 1; c!=Board::kNCols+1; ++c)
         if (board(r, c)==kWhite)
            output_ << " o";
```

```
          else if (board(r, c)==kBlack)
             output_ << " *";
          else
             output_ << "  ";
    }
    output_ << "\n[ Black(*): " << board.score(kBlack)
            << "    White(o): " << board.score(kWhite)
            << " ]\n\n";
}
```

Not very fancy—black pieces are displayed as asterisks, white pieces as o characters—but it will do, and it has the advantage of being portable to a wide variety of platforms. Also note that this view handles only boards with no more than nine columns and no more than 26 rows because I wanted to use only one character for the coordinates (nine digits for the columns, 26 letters for the rows).

Next, we need to define the interface for a `Player` and add the implementation for a human player using a text interface. The implementation of a virtual player (computer) will be discussed later beacuse that's a little more intricate. Assume for now that a `VirtualTextPlayer` type is available.

```
// File: Player.H
#ifndef PLAYER_H
#define PLAYER_H

#include "Board.H"

struct Player {
   virtual ~Player() {}
   virtual void get_move(Board const&, int &r, int &c) = 0;
   virtual void skip_move(Board const&) {}
   virtual void win() {}
   virtual void lose() {}
   virtual void tie() {}
};
#endif
```

The `Game` object will repeatedly ask a player for a move through the member `Player::get_move` unless no valid move exists, in which case the `Player` is given a chance to react via an invocation of `Player::skip_move`. A player need not store the board; it is passed as an argument for each move. Once the game is over, the player is notified of the result through a call to `win`, `lose`, or `tie`.

```
// File: HumanTextPlayer.H
#ifndef HUMANTEXTPLAYER_H
#define HUMANTEXTPLAYER_H
```

```
#include <iostream>
#include "Board.H"
#include "Player.H"

using std::istream;
using std::ostream;

struct HumanTextPlayer: Player {
   HumanTextPlayer(ostream &output, istream &input,
      Color color)
         : output_(output), input_(input), color_(color) {}
   virtual void get_move(Board const&, int &row, int &col);
   virtual void skip_move(Board const&);
   virtual void win();
   virtual void tie();
private:
   ostream &output_;
   istream &input_;
   Color color_;
};
#endif
```

Notice how the input and output channels are passed to the HumanTextPlayer's constructor. This should allow simple customization if cout and cin are not the desired input/output streams.

```
// File: HumanTextPlayer.C
#include <assert.h>
#include "HumanTextPlayer.H"

namespace { // anonymous namespace
   void convert(char &ch, int upper) {
      if (ch>='a' and ch<'a'+upper)
         ch = ch-'a'+1;
      else if (ch>='A' and ch<'A'+upper)
         ch = ch-'A'+1;
      else if (ch>='1' and ch<'1'+upper)
         ch = ch-'1'+1;
   }

   bool in_board(int row, int col) {
      return (row>0) and (row<=Board::kNRows)
         and (col>0) and (col<=Board::kNCols);
   }
   void print_color(ostream &output, Color color) {
      if (color==kBlack)
         output << "Black";
      else
         output << "White";
   }
} // end anonymous namespace

void HumanTextPlayer::get_move(Board const &board,
                               int &r, int &c) {
   char row, col;
   do {
      print_color(output_, color_);
      output_ << ", enter your move (e.g., a7) : ";
      input_ >> row >> col;
      convert(row, Board::kNRows);
      convert(col, Board::kNCols);
      if (in_board(row, col)
      and board.valid_move(color_, row, col))
         break;
      board.get_possible_move(color_, r, c);
      output_ << "Invalid move. Please try again.\n"
              << char('a'+r-1) << char('1'+c-1)
              << " is a valid example,"
                 " but other moves may be better.\n";
   } while (1);
   r = row; c = col;
}

void HumanTextPlayer::skip_move(Board const&) {
   print_color(output_, color_);
   output_ << ": no valid move possible.\n";
}

void HumanTextPlayer::win() {
   print_color(output_, color_);
   output_ << " wins!\n";
}

void HumanTextPlayer::tie() {
   if (color_==kBlack)
      output_ << "Game tied.\n";
   else
      output_ << "No winner!\n";
}
```

The member function `Player::lose` was intentionally left unchanged. If one player is said to win, it is not necessary to say that the other one loses.

Supplementary Exercise

(Minor correction and potential extensions) The text-based interface presented in `HumanTextPlayer` and `BoardTextViewer` makes unportable assumptions about the character set. For example, it is not guaranteed that all the lowercase letters are encoded using consecutive values. Change the code to remove that assumption. Also improve the interface in any other way you can imagine. Are there any changes you would make to the abstract (interface) classes `BoardViewer` and `Player`?

Now we're ready to show the `Game` class that will manage the board and the two players. As mentioned, you might just as well implement this as a function. The `Game` object also changed a little from the original discussion:

```
// File: Game.H
#ifndef GAME_H
#define GAME_H

#include "Board.H"

struct BoardViewer;
struct Player;

struct Game {
   Game(Player *black, Player *white)
      : black_(black), white_(white) {}
   void select_viewer(BoardViewer *viewer)
      { viewer_ = viewer; }
   void play();
private:
   Player *black_, *white_;
   Board board_;
   BoardViewer *viewer_;
};
#endif
```

The `BoardViewer` can be selected after the construction of the `Game` object. This makes it easier to have an interactive choice for board rendering (not exploited here).

```
// File: Game.C

#include "BoardViewer.H"
#include "Game.H"
#include "Player.H"
```

```
void Game::play() {
   bool more_moves;
   viewer_->draw(board_);
   do {
      int r, c;
      more_moves = false;
      if (board_.get_possible_move(kBlack, r, c)) {
         more_moves = true;
         black_->get_move(board_, r, c);
         board_.move(kBlack, r, c);
         viewer_->draw(board_);
      } else
         black_->skip_move(board_);
      if (board_.get_possible_move(kWhite, r, c)) {
         more_moves = true;
         white_->get_move(board_, r, c);
         board_.move(kWhite, r, c);
         viewer_->draw(board_);
      } else
         white_->skip_move(board_);
   } while (more_moves);
   if (board_.score(kWhite)>board_.score(kBlack)) {
      white_->win(); black_->lose();
   } else
   if (board_.score(kBlack)>board_.score(kWhite)) {
      black_->win(); white_->lose();
   } else {
      black_->tie(); white_->tie();
   }
}
```

Let's put all the pieces together (still assuming we will have a `VirtualTextPlayer` type):

```
// File: Reversi.C
#include "BoardTextViewer.H"
#include "Game.H"
#include "HumanTextPlayer.H"
#include "VirtualTextPlayer.H"
#include <iostream>

using std::cout;
using std::cin;
```

```
Player* select_player(Color color) {
   cout << (color==kWhite? "White: ": "Black: ")
        << "[H]uman or [C]omputer? ";
   char answer;
   cin >> answer;
   if (answer=='h' || answer=='H')
      return new HumanTextPlayer(cout, cin, color);
   else
      return new VirtualTextPlayer(cout, color);
}

int main() {
   Game game(select_player(kBlack), select_player(kWhite));
   game.select_viewer(new BoardTextViewer(cout));
   game.play();
   return 0;
}
```

Although we have spent several pages developing our game, you may agree that so far things have been conceptually relatively simple. At the very least, we've stayed within the domain of knowledge covered in *The C++ Programming Language*. I'm also hopeful that you'll find the decomposition of the problem natural and amenable to convenient modification and extension. You'll note the special attention to separating responsibilities and to isolating the implementations of all those responsibilities from each other. Yet I also made mistakes; I'll come back to them later on.

The last part of this implementation of the Reversi game is the code that allows the computer to choose moves. This will delve into a basic game strategy technique that you might want to skip on first reading.

An uninteresting strategy would be to evaluate every board position for validity and randomly pick any legal move. Such an opponent would be easily defeated. Instead, the computer should try to associate a value with each valid board position and select the best one. This is easier said than done. In reply to a move, the opponent usually has a variety of choices for the next move, but we do not know what the adversary's choice will be. A reasonable heuristic, however, is that the opponent will play as well as possible. Therefore, among all the valid moves we can make, we will pick (in order of preference):

- One that ends the game in our favor
- One for which the best reply of our opponent gives him or her the least advantage

The program will need to *simulate* our potential moves and the corresponding potential countermoves of our opponent on separate `Board` objects. The value associated with a simulated move of the opponent will be computed using the same method except that what is good for the opponent is bad for the player, and vice-versa. This simulation of moves and countermoves could theoretically be repeated to a large depth, but, in practice, you would find that only a few levels can be explored in a reasonable amount of time. At some point, we will stop the recursive evaluation and associate a numerical value with the board situation last simulated.

This method is often referred to as a *min-max* algorithm because it alternately selects the maximum value for our move and the minimum value for the opponent's countermove.

The following code realizes this idea:

```
// File: VirtualPlayer.H
#ifndef VIRTUALPLAYER_H
#define VIRTUALPLAYER_H

#include "Board.H"

struct VirtualPlayer {
   VirtualPlayer(Color c): color_(c), maxdepth_(6) {}
   void get_move(Board const&, int &row, int &col);
private:
   int eval_self(Board const&, int r, int c,
                 int best, int worst, int depth);
   int eval_opponent(Board const&, int r, int c,
                     int best, int worst, int depth);
   int eval_end_of_game(Board const&);
   int value_of(Board const&);
   Color color_;
   int maxdepth_;
};
#endif

// File: VirtualPlayer.C
#include "VirtualPlayer.H"
#include <limits>

void VirtualPlayer::get_move(Board const &board,
                             int &row, int &col) {
   int best = std::numeric_limits<int>::min(),
       worst = std::numeric_limits<int>::max();
   row = col = 0;
   for (int r = 1; r!=Board::kNRows+1; ++r)
      for (int c = 1; c!=Board::kNCols+1; ++c)
         if (board.valid_move(color_, r, c)) {
            int score = eval_self(board, r, c,
                                  best, worst, 1);
            if (score>=best) {
               best = score;
               row = r; col = c;
            }
         }
}
```

```
int VirtualPlayer::eval_self(Board const &board,
                             int row, int col,
                             int best, int worst, int d) {
   if (d==maxdepth_) // Maximum depth of exploration?
      return value_of(board);
   Board copy(board);
```

It may be possible that we are evaluating the virtual player's position just after it had to forfeit its simulated turn. That cast would be indicated with `row==0`.

```
   if (row>0)
      copy.move(color_, row, col);
```

We must next test every board position to see if it could be a valid response for the opponent. If this is the case, we must find out how good a move it was (using `eval_opponent`). If we find that the opponent's countermove will place us in a worse situation than that which our other moves simulated so far lead to (described by the parameter `best`), we can discard this move right away. Otherwise, we remember what the worst score is (that is, the score of the best countermove of the opponent).

```
   bool move_result = kUnsuccessful;
   for (int r = 1; r!=Board::kNRows+1; ++r)
      for (int c = 1; c!=Board::kNCols+1; ++c)
         if (copy.valid_move(opponent_of(color_), r, c)) {
            move_result = kSuccessful;
            int score = eval_opponent(copy, r, c,
                                      best, worst, d+1);
            if (score<=best) // This won't get any better:
               return score; // "prune" this branch now.
            if (score<worst)
               worst = score;
         }
```

If the opponent had a valid countermove, we return the score associated with the best one. If we can still make a move, we simulate the opponent forfeiting a turn by calling `eval_opponent` with a (0, 0) move. Otherwise, we have simulated the end of a game and return the evaluation of that.

```
   if (move_result==kSuccessful)
      return worst;
   else // The opponent had no valid response
      if (copy.get_possible_move(color_, row, col)
            ==kSuccessful)
         return eval_opponent(copy, 0, 0, best, worst, d+1);
      else // We reached an end-of-game situation
         return eval_end_of_game(copy);
}
```

Looking at the eval_self function above, you may have noticed a twist on the basic min-max algorithm. Remember that eval_self searches, for a given board situation, all the opponent's possible moves and selects the lowest possible score (indicating the best choice of the opponent). We are interested in finding the board situation that would force the highest such low point. Therefore, if we have already evaluated a low point for one board situation (the best parameter) and another situation reveals a lower score, we can abandon the search in that board right away because its low point can only decrease and we will never steer the game to that situation. The eval_opponent function will have a totally symmetric role and can, therefore, abandon searching a board as soon as that board situation promises to produce a high point that is higher than the lowest high point for previous situations. This early-termination technique is known as α–β pruning—α and β refer to the high points and low points that are carried along in the simulation (parameters best and worst in the evaluation functions). The implementation of eval_opponent below is identical to eval_self except that the roles of self and opponent are swapped.

```
int VirtualPlayer::eval_opponent(Board const &board,
                                 int row, int col,
                                 int best, int worst, int d)
{
   if (d==maxdepth_) // Maximum depth of exploration reached?
      return value_of(board);
   Board copy(board);
   if (row>0) // Row==0 when we have no valid move
      copy.move(opponent_of(color_), row, col);
   bool move_result = kUnsuccessful;
   // Explore own possible responses:
   for (int r = 1; r!=Board::kNRows+1; ++r)
      for (int c = 1; c!=Board::kNCols+1; ++c)
         if (copy.valid_move(color_, r, c)) {
            move_result = kSuccessful;
            int score = eval_self(copy, r, c,
                                  best, worst, d+1);
            if (score>=worst) // This won't get any better:
               return score;  // "prune" this branch now.
            if (score>best)
               best = score;
         }
   if (move_result==kSuccessful)
      return best;
   else // We have no valid response
      if (copy.get_possible_move(opponent_of(color_),
                                 row, col)==kSuccessful)
         return eval_self(copy, 0, 0, best, worst, d+1);
      else // We reached an end-of-game situation
         return eval_end_of_game(copy);
}
```

To complete the virtual player, we must write the functions evaluating board situations that will not be explored further. This occurs for two reasons: The situation corresponds to the end of a game, or we have reached the maximum depth that we're willing to explore. The former case is relatively inflexible. If the situation corresponds to a win, we give it a very large score. If it's a tie, we return 0. Otherwise, we return a very negative value. The second case will be essential in the quality of the game played by the computer. If we can produce values that are truly representative of the potential development of the situation under evaluation, the computer will tend to steer the game towards the good developments and possibly win. Perhaps the most straightforward evaluation function is the one that returns the material advantage of the player—how many more pieces does the virtual player have on the board than its opponent. However, *position* is also crucial. For example, a corner position can never be taken away from a player and is, therefore, something to strive for. On the other hand, having a piece next to an empty corner position can sometimes be very dangerous because the opponent can use it to capture the powerful corner position. The following function can certainly be improved on:

```
int VirtualPlayer::eval_end_of_game(Board const &board) {
   if (board.score(color_)
          >board.score(opponent_of(color_)))
      return std::numeric_limits<int>::max()-1;
   if (board.score(color_)
          >board.score(opponent_of(color_)))
      return std::numeric_limits<int>::min()+1;
   else
      return 0; // The value of a tie could be biased
}               // if desired.

namespace { // unnamed namespace
   int corner_value(Color player, Color corner,
                    Color p1, Color p2, Color p3) {
      Color opp = opponent_of(player);
      if (corner==player)
         return 30;
      if (corner==opponent)
         return -30;
      int value = 0;
      if ((p1==player) || (p2==player) || (p3==player))
         value -= 30;
      if ((p1==opp) || (p2==opp) || (p3==opp))
         value += 30;
      return value;
   }
} // end unnamed namespace
```

```
int VirtualPlayer::value_of(Board const &s) {
   Color opponent = opponent_of(color_);
   // First component: material advantage
   int value = s.score(color_)-s.score(opponent);
   // Second component: corner positions
   int t = 1, l = 1, r = Board::kNCols, b = Board::kNRows;
   value += corner_value(color_, s(t, l),
                         s(t+1, l), s(t, l+1), s(t+1, l+1));
   value += corner_value(color_, s(t, r),
                         s(t+1, r), s(t, r-1), s(t+1, r-1));
   value += corner_value(color_, s(b, r),
                         s(b-1, r), s(b, r-1), s(b-1, r-1));
   value += corner_value(color_, s(b, l),
                         s(b-1, l), s(b, l+1), s(b-1, l+1));
   return value;
}
```

All that is left to do is to plug this virtual player into the `Player` interface. Note again how we did not combine a user interface in the `VirtualPlayer` class. It is, therefore, convenient to reuse `VirtualPlayer` for a fancy graphical implementation of the game or for any other desired interface.

```
// File: VirtualTextPlayer
#ifndef VIRTUALTEXTPLAYER_H
#define VIRTUALTEXTPLAYER_H

#include "Player.H"
#include <iostream>

struct VirtualTextPlayer: Player {
   VirtualTextPlayer(std::ostream &output, Color color)
      : output_(output), color_(color) {}
   virtual void get_move(Board const&, int &row, int &col);
   virtual void skip_move(Board const&);
   virtual void win();
   virtual void tie();
private:
   std::ostream &output_;
   Color color_;
};
#endif
```

```
// File: VirtualTextPlayer.C
#include "Board.H"
#include "VirtualPlayer.H"
#include "VirtualTextPlayer.H"

namespace { // unnamed namespace
   void print_color(std::ostream &output, Color color) {
      if (color==kBlack)
         output << "Black";
      else
         output << "White";
   }
} // end unnamed namespace

void VirtualTextPlayer::get_move(Board const &board,
                                 int &row, int &col) {
   VirtualPlayer machine(color_);
   machine.get_move(board, row, col);
   print_color(output_, color_);
   output_ << " plays ["
           << char('a'+row-1) << char('1'+col-1) << "]\n";
}

void VirtualTextPlayer::skip_move(Board const&) {
   print_color(output_, color_);
   output_ << ": no valid move possible.\n";
}

void VirtualTextPlayer::win() {
   print_color(output_, color_);
   output_ << " wins!\n";
}

void VirtualTextPlayer::tie() {
   if (color_==kBlack)
      output_ << "Game tied.\n";
   else
      output_ << "No winner!\n";
}
```

Voilà. We're done.

Unfortunately, the implementation has some minor flaws. Fixing them is the object of the following Supplementary Exercise.