```
abort();                    /* Abnormal program termination */
```

This statement can appear anywhere in your program.

Calling the `exit()` function results in a normal end to a program. The function requires an integer argument that will be returned to the operating system. A value of 0 usually represents a completely normal program end. You may use other values to indicate the status of the program in some way. What happens to the value returned by `exit()` is determined by your operating system. You could call the `exit()` function like this:

```
exit(1);                    /* Normal program end - status is 1 */
```

You can call `exit()` to terminate execution from anywhere in your program.

As you have seen, you can also end a program by executing a `return` statement with an integer value in `main()`, for example:

```
return 1;
```

The return statement has a special meaning in `main()` (and *only* in `main()`, not in any other function), and it's the equivalent of calling `exit()` with the value specified in the `return` statement as the argument. Thus, the value in the `return` statement will be passed back to the operating system.

Libraries of Functions: Header Files

I've already mentioned that your compiler comes with a wide range of standard functions that are declared in **header files**, sometimes called **include files**. These represent a rich source of help for you when you're developing your own applications. You've already come across some of these because header files are such a fundamental component of programming in C. So far you have used functions declared in the header files listed in Table 9-2.

Table 9-2. *Standard Header Files You Have Used*

Header File	Functions
`<stdio.h>`	Input/output functions
`<stdarg.h>`	Macros supporting a variable number of arguments to a function
`<math.h>`	Mathematical floating-point functions
`<stdlib.h>`	Memory allocation functions
`<string.h>`	String-handling functions
`<stdbool.h>`	`bool` type and Boolean values `true` and `false`
`<complex.h>`	Complex number support
`<ctype.h>`	Character classification functions
`<wctype.h>`	Wide character conversion functions

All the header files in Table 9-2 contain declarations for a range of functions, as well as definitions for various constants. They're all ISO/IEC standard libraries, so all standard-conforming compilers will support them and provide at least the basic set of functions, but typically they'll supply much more. To comprehensively discuss the contents of even the ISO/IEC standard library header files and functions could double the size of this book, so I'll mention just the most important aspects of the standard header files and leave it to you to browse the documentation that comes with your compiler.

The header file <stdio.h> contains declarations for quite a large range of high-level input/output (I/O) functions, so I'll devote the whole of Chapter 10 to exploring some of these further, particularly for working with files.

As well as memory allocation functions, <stdlib.h> provides facilities for converting character strings to their numerical equivalents from the ASCII table. There are also functions for sorting and searching. In addition, functions that generate random numbers are available through <stdlib.h>.

You used the <string.h> file in Chapter 6. It provides functions for working with null-terminated strings.

A header file providing a very useful range of functions also related to string processing is <ctype.h>, which you also saw in Chapter 6. The <ctype.h> header includes functions to convert characters from uppercase to lowercase (and vice versa) and a number of functions for checking for alphabetic characters, numeric digits, and so on. These provide an extensive toolkit for you to do your own detailed analysis of the contents of a string, which is particularly useful for dealing with user input.

The <wctype.h> header provides wide character classification functions, and <wchar.h> provides extended multibyte-character utility functions.

I strongly recommend that you invest some time in becoming familiar with the contents of these header files and the libraries that are supplied with your compiler. This familiarity will greatly increase the ease with which you can develop applications in C.

Enhancing Performance

You have two facilities that are intended to provide cues to your compiler to generate code with better performance. One relates to how short function calls are compiled, and the other is concerned with the use of pointers. The effects are not guaranteed though and depend on your compiler implementation. I'll discuss short functions first.

Declaring Functions inline

The functional structure of the C language encourages the segmentation of a program into many functions, and the functions can sometimes be very short. With very short functions it is possible to improve execution performance by replacing each function call of a short function with inline code that implements the effects of the function. You can indicate that you would like this technique to be applied by the compiler by specifying a short function as inline. Here's an example:

```
inline double bmi(double kg_wt, double m_height)
{
  return kg_wt/(m_height*m_height);
}
```

This function calculates an adult's Body Mass Index from their weight in kilograms and their height in meters. This operation is sensibly defined as a **function** but it is also a good candidate for **inline implementation of calls** because the code is so simple. This is specified by the `inline` keyword in the function header. There's no guarantee in general that the compiler will take note of a function being declared as `inline` though.

Using the restrict Keyword

Sophisticated C compilers have the capability to optimize the performance of the object code, and this can involve changing the sequence in which calculations occur compared to the sequence in which you specify operations in the code. For such code optimization to be possible, the compiler must be sure that such resequencing of operations will not affect the result of the calculations, and pointers represent something of a problem in this respect. To allow optimization of code involving pointers, the compiler has to be certain that the pointers are not aliased—in other words the data item that each pointer references is not referenced by some other means in a given scope. The `restrict` keyword provides a way for you to tell the compiler when this is the case and thus allows code optimization to be applied.

Here's an example of a function that is declared in `<string.h>`:

```
char *strcpy(char * restrict s1, char * restrict s2)
{
  /* Implementation of the function to copy s2 to s1 */
}
```

This function copies `s2` to `s1`. The `restrict` keyword is applied to both parameters thus indicating that the strings referenced by `s1` and `s2` are only referenced through those pointers in the body of the function, so the compiler can optimize the code generated for the function. The `restrict` keyword only imparts information to the compiler and does not guarantee that any optimization will be applied. Of course, if you use the `restrict` keyword where the condition does not apply, your code may produce incorrect results.

Most of the time you won't need to use the `restrict` keyword. Only if your code is very computationally intensive will it have any appreciable effect, and even then it depends on your compiler.

Designing a Program

At this point, you've finished with functions, and because you're more than halfway through the capabilities of C, an example of reasonable complexity wouldn't come amiss. In this program, you're going to put to practical use the various elements of C that you've covered so far in the book.

The Problem

The problem that you're going to solve is to write a game. There are several reasons for choosing to write a game. First, games tend to be just as complex, if not more so, as other types of programs, even when the game is simple. And second, games are more fun.

The game is in the same vein as Othello or, if you remember Microsoft Windows 3.0, Reversi. The game is played by two players who take turns placing a colored counter on a square board. One player has black counters and the other has white counters. The board has an even number of squares along each side. The starting position, followed by five successive moves, is shown in Figure 9-4.

You can only place a counter adjacent to an opponent's counter, such that one or more of your opponent's counters—in a line diagonally, horizontally, or vertically—are enclosed between two of your own counters. The opponent's counters are then changed to counters of your own color. The person with the most counters on the board at the end of the game wins. The game ends when all the squares are occupied by counters. The game can also end if neither player can place a counter legally, which can occur if you or your opponent manage to convert all the counters to the same color.

The game can be played with any size of board, but you'll implement it here on a 6 × 6 board. You'll also implement it as a game that you play against the computer.

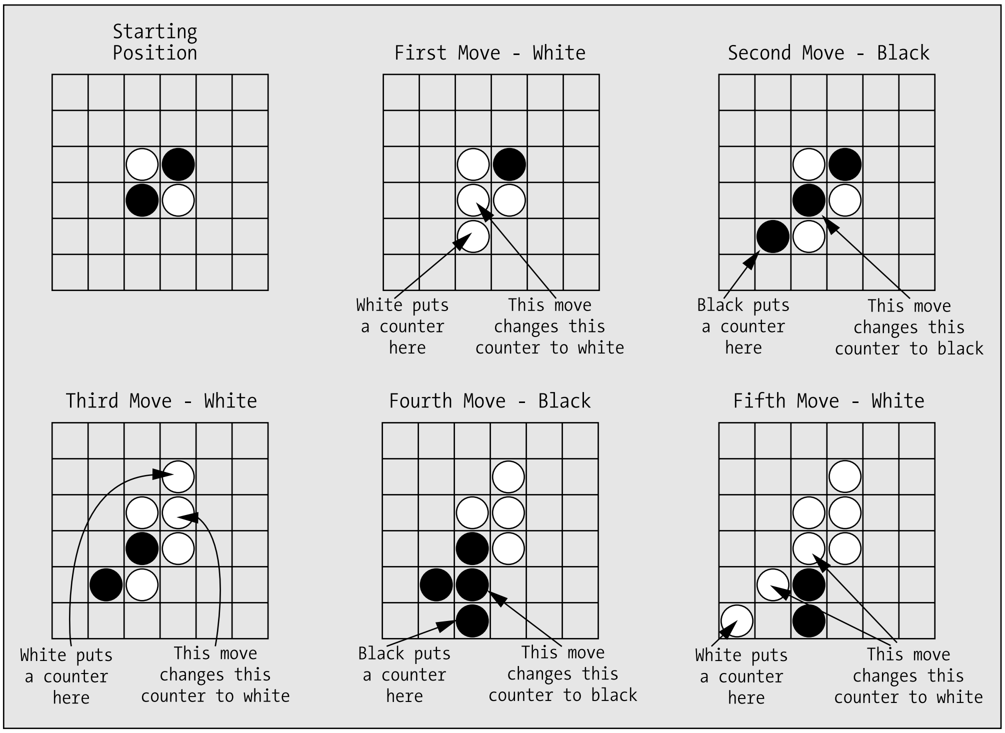
Starting Position
First Move - White
Second Move - Black
White puts a counter here
This move changes this counter to white
Black puts a counter here
This move changes this counter to black
Third Move - White
Fourth Move - Black
Fifth Move - White
White puts a counter here
This move changes this counter to white
Black puts a counter here
This move changes this counter to black
White puts a counter here
This move changes this counter to white

Figure 9-4. *Starting position and initial moves in Reversi*

The Analysis

This problem's analysis is a little different from those you've seen up to now. The whole point of this chapter is to introduce the concept of **structured programming**—in other words, breaking a problem into small pieces—which is why you've spent so long looking at functions.

A good way to start is with a diagram. You'll start with a single box, which represents the whole program, or the `main()` function, if you like. Developing from this, on the next level down, you'll show the functions that will need to be directly called by the `main()` function, and you'll indicate what these functions have to do. Below that, you'll show the functions that those functions in turn have to use. You don't have to show the actual functions; you can show just the tasks that need to be accomplished. However, these tasks *do* tend to be functions, so this is a great way to design your program. Figure 9-5 shows the tasks that your program will need to perform.

You can now go a step further than this and begin to think about the actual sequence of actions, or functions, that the program is going to perform. Figure 9-6 is a flowchart that describes the same set of functions but in a manner that shows the sequence in which they're executed and the logic involved in deciding that. You're moving closer now to a more precise specification of how the program will work.

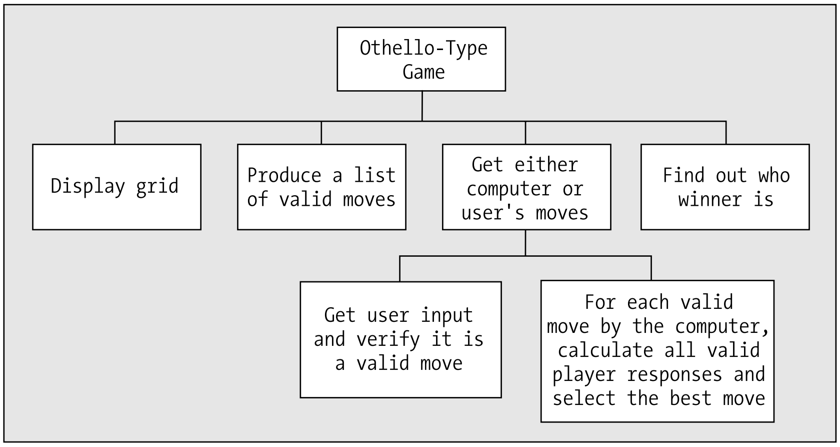
Othello-Type Game
Display grid
Produce a list of valid moves
Get either computer or user's moves
Find out who winner is
Get user input and verify it is a valid move
For each valid move by the computer, calculate all valid player responses and select the best move

Figure 9-5. *Tasks in the Reversi program*

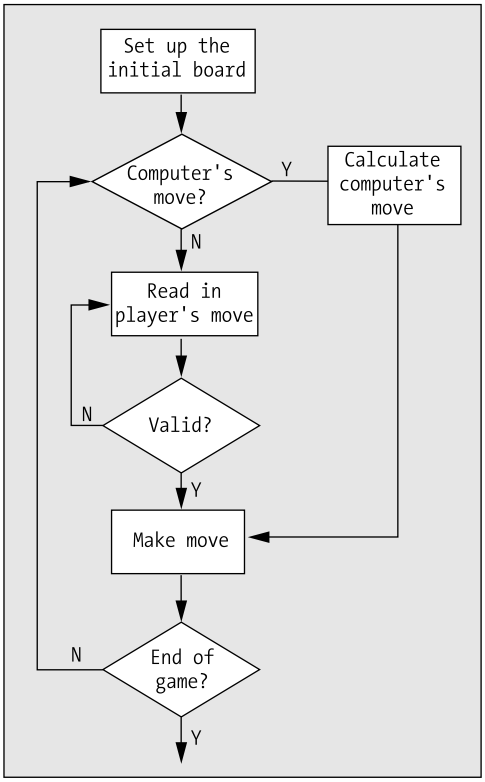

Figure 9-6. *The basic logic of the Reversi program*

This isn't absolutely fixed, of course. There's a lot of detail that you'll need to fill in. This sort of diagram can help you get the logic of the program clear in your mind, and from there you can progress to a more detailed definition of how the program will work.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

The first thing to do is to set up and display the initial board. You'll use a smaller-than-normal board (6 × 6), as this makes the games shorter, but you'll implement the program with the board size as a symbol defined by a preprocessor directive. You'll then be able to change the size later if you want. You'll display the board using a separate function, as this is a self-contained activity.

Let's start with the code to declare, initialize, and display the grid. The computer will use `'@'` as its counter, and the player will have `'O'` for his counter:

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>

const int SIZE =  6;                   /* Board size - must be even */
const char comp_c = '@';               /* Computer's counter        */
const char player_c = 'O';             /* Player's counter          */

/* Function prototypes */
void display(char board[][SIZE]);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };    /* The board          */
  int row = 0;                        /* Board row index    */
  int col = 0;                        /* Board column index */

  printf("\nREVERSI\n\n");
  printf("You can go first on the first game, then we will take turns.\n");
  printf("   You will be white - (%c)\n   I will be black   - (%c).\n",
                                                       player_c, comp_c);
  printf("Select a square for your move by typing a digit for the row\n "
                  "and a letter for the column with no spaces between.\n");
  printf("\nGood luck!  Press Enter to start.\n");
  scanf("%c", &again);

  /* Blank all the board squares */
  for(row = 0; row < SIZE; row++)
    for(col = 0; col < SIZE; col++)
      board[row][col] = ' ';

  /* Place the initial four counters in the center */
  int mid = SIZE/2;
  board[mid - 1][mid - 1] = board[mid][mid] = player_c;
  board[mid - 1][mid] = board[mid][mid - 1] = comp_c;
  display(board);                     /* Display the board  */
  return 0;
}
```

```
/***********************************************
 * Function to display the board in its       *
 * current state with row numbers and column   *
 * letters to identify squares.                *
 * Parameter is the board array.               *
 ***********************************************/
void display(char board[][SIZE])
{
  /* Display the column labels */
  char col_label = 'a';              /* Column label   */
  printf("\n ");                     /* Start top line */
  for(int col = 0 ; col<SIZE ;col++)
    printf("   %c", col_label+col);   /* Display the top line */
  printf("\n");                      /* End the top line     */

  /* Display the rows… */
  for(int row = 0; row < SIZE; row++)
  {
     /* Display the top line for the current row */
    printf("  +");
    for(int col = 0; col<SIZE; col++)
      printf("---+");
    printf("\n%2d|",row + 1);

    /* Display the counters in current row */
    for(int col = 0; col<SIZE; col++)
      printf(" %c |", board[row][col]);  /* Display counters in row */
    printf("\n");
  }

  /* Finally display the bottom line of the board */
  printf("  +");                     /* Start the bottom line   */
  for(int col = 0 ; col<SIZE ; col++)
    printf("---+");                  /* Display the bottom line */
  printf("\n");                      /* End the bottom  line    */
}
```

The function `display()` outputs the board with row numbers to identify the rows and the letters from `'a'` onward to identify each column. This will be the reference system by which the user will select a square to place his counter.

The code looks complicated, but it's quite straightforward. The first loop outputs the top line with the column label, from `'a'` to `'f'` with the board size. The next loop outputs the squares that can contain counters, a row at a time, with a row number at the start of each row. The last loop outputs the bottom of the last row. Notice how you've passed the `board` array as an argument to the `display()` function rather than making `board` a global variable. This is to prevent other functions from modifying the contents of `board` accidentally. The function will display a board of any size.

Step 2

You need a function to generate all the possible moves that can be made for the current player. This function will have two uses: first, it will allow you to check that the move that the player enters is valid, and second, it will help you to determine what moves the computer can make. But first you must decide how you're going to represent and store this list of moves.

So what information do you need to store, and what options do you have? Well, you've defined the grid in such a way that any cell can be referenced by a row number and a column letter. You could therefore store a move as a string consisting of a number and a letter. You would then need to accommodate a list of moves of varying length, to allow for the possibility that the dimensions of the board might change to 10 × 10 or greater.

There's an easier option. You can create a second array with elements of type `bool` with the same dimensions as the board, and store `true` for positions where there is a valid move, and `false` otherwise. The function will need three parameters: the `board` array, so that it can check for vacant squares; the `moves` array, in which the valid moves are to be recorded; and the identity of the current player, which will be the character used as a counter for the player.

The strategy will be this: for each blank square, search the squares around that square for an opponent's counter. If you find an opponent's counter, follow a line of opponent counters (horizontal, vertical, or diagonal) until you find a player counter. If you do in fact find a player counter along that line, then you know that the original blank square is a valid move for the current player.

You can add the function definition to the file following the definition of the `display()` function:

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>
#include <stdbool.h>

const int SIZE = 6;                    /* Board size - must be even */
const char comp_c = '@';               /* Computer's counter        */
const char player_c = 'O';             /* Player's counter          */

/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };       /* The board          */
  bool moves[SIZE][SIZE] = { false };    /* Valid moves        */
  int row = 0;                           /* Board row index      */
  int col = 0;                           /* Board column index    */

   /* Other code for main as before... */
}

/* Code for definition of display() as before... */

/***********************************************
 * Calculates which squares are valid moves    *
 * for player. Valid moves are recorded in the *
 * moves array - true indicates a valid move,  *
 * false indicates an invalid move.            *
 * First parameter is the board array          *
 * Second parameter is the moves array         *
 * Third parameter identifies the player       *
 * to make the move.                           *
 * Returns valid move count.                   *
 ***********************************************/
```

```
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player)
{
  int rowdelta = 0;                  /* Row increment around a square    */
  int coldelta = 0;                  /* Column increment around a square */
  int x = 0;                         /* Row index when searching         */
  int y = 0;                         /* Column index when searching      */
  int no_of_moves = 0;               /* Number of valid moves            */

  /* Set the opponent            */
  char opponent = (player == player_c) ? comp_c : player_c;

  /* Initialize moves array to false */
  for(int row = 0; row < SIZE; row++)
    for(int col = 0; col < SIZE; col++)
      moves[row][col] = false;

  /* Find squares for valid moves.                           */
  /* A valid move must be on a blank square and must enclose */
  /* at least one opponent square between two player squares */
  for(int row = 0; row < SIZE; row++)
    for(int col = 0; col < SIZE; col++)
    {
      if(board[row][col] != ' ')     /* Is it a blank square?  */
        continue;                    /* No - so on to the next */

      /* Check all the squares around the blank square  */
      /* for the opponents counter                      */
      for(rowdelta = -1; rowdelta <= 1; rowdelta++)
        for(coldelta = -1; coldelta <= 1; coldelta++)
        {
          /* Don't check outside the array, or the current square */
          if(row + rowdelta < 0 || row + rowdelta >= SIZE ||
             col + coldelta < 0 || col + coldelta >= SIZE ||
                                  (rowdelta==0 && coldelta==0))
            continue;

          /* Now check the square */
          if(board[row + rowdelta][col + coldelta] == opponent)
          {
            /* If we find the opponent, move in the delta direction  */
            /* over opponent counters searching for a player counter */
            x = row + rowdelta;       /* Move to          */
            y = col + coldelta;       /* opponent square  */

            /* Look for a player square in the delta direction */
            for(;;)
            {
              x += rowdelta;          /* Go to next square */
              y += coldelta;          /* in delta direction*/

              /* If we move outside the array, give up */
              if(x < 0 || x >= SIZE || y < 0 || y >= SIZE)
                break;
              /* If we find a blank square, give up */
              if(board[x][y] == ' ')
                break;

              /*  If the square has a player counter */
              /*  then we have a valid move          */
              if(board[x][y] == player)
              {
                moves[row][col] = true;  /* Mark as valid */
                no_of_moves++;           /* Increase valid moves count */
                break;                   /* Go check another square    */
              }
            }
          }
        }
    }
  return no_of_moves;
}
```

You have added a prototype for the `valid_moves()` function and a declaration for the array `moves` in `main()`.

Because the counters are either `player_c` or `comp_c`, you can set the opponent counter in the `valid_moves()` function as the one that isn't the player counter that's passed as an argument. You do this with the conditional operator. You then set the `moves` array to `false` in the first nested loop, so you only have to set valid positions to `true`. The second nested loop iterates through all the squares on the board, looking for those that are blank. When you find a blank square, you search for an opponent counter in the inner loop:

```
/* Check all the squares around the blank square  */
/* for the opponents counter                      */
for(rowdelta = -1; rowdelta <= 1; rowdelta++)
  for(coldelta = -1; coldelta <= 1; coldelta++)
    ...
```

This will iterate over all the squares that surround the blank square and will include the blank square itself, so you skip the blank square or any squares that are off the board with this `if` statement:

```
/* Don't check outside the array, or the current square */
if(row + rowdelta < 0 || row + rowdelta >= SIZE ||
   col + coldelta < 0 || col + coldelta >= SIZE ||
                          (rowdelta==0 && coldelta==0))
  continue;
```

If you get past this point, you've found a nonblank square that's on the board. If it contains the opponent's counter, then you move in the same direction, looking for either more opponent counters or a player counter. If you find a player counter, the original blank square is a valid move, so you record it. If you find a blank or run of the board, it isn't a valid move, so you look for another blank square.

The function returns a count of the number of valid moves, so you can use this value to indicate whether the function returns any valid moves. Remember, any positive integer is `true` and 0 is `false`.

Step 3

Now that you can produce an array that contains all the valid moves, you can fill in the game loop in `main()`. You'll base this on the flowchart that you saw earlier. You can start by adding two nested do-while loops: the outer one will initialize each game, and the inner one will iterate over player and computer turns.

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>
#include <stdbool.h>

const int SIZE = 6;                   /* Board size - must be even */
const char comp_c = '@';              /* Computer's counter        */
const char player_c = 'O';            /* Player's counter          */

/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };      /* The board           */
  bool moves[SIZE][SIZE] = { false };   /* Valid moves         */
  int row = 0;                          /* Board row index     */
  int col = 0;                          /* Board column index  */
  int no_of_games = 0;                  /* Number of games     */
  int no_of_moves = 0;                  /* Count of moves      */
  int invalid_moves = 0;                /* Invalid move count  */
  int comp_score = 0;                   /* Computer score      */
  int user_score = 0;                   /* Player score        */
  char again = 0;                       /* Replay choice input */

  /* Player indicator: true for player and false for computer */
  bool next_player = true;

  /* Prompt for how to play - as before */

  /* The main game loop */
  do
  {
    /* On even games the player starts; */
    /* on odd games the computer starts */
    next_player = !next_player;
    no_of_moves = 4;                  /* Starts with four counters */

    /* Blank all the board squares */
    for(row = 0; row < SIZE; row++)
      for(col = 0; col < SIZE; col++)
        board[row][col] = ' ';

    /* Place the initial four counters in the center */
    int mid = SIZE/2;
    board[mid - 1][mid - 1] = board[mid][mid] = player_c;
    board[mid - 1][mid] = board[mid][mid - 1] = comp_c;
    /* The game play loop */
    do
    {
      display(board);                 /* Display the board  */
      if(next_player = !next_player)
      { /*   It is the player's turn                   */
        /* Code to get the player's move and execute it */
      }
      else
      { /* It is the computer's turn                   */
        /* Code to make the computer's move            */
      }
    }while(no_of_moves < SIZE*SIZE && invalid_moves<2);

    /* Game is over */
    display(board);                   /* Show final board  */

    /* Get final scores and display them */
    comp_score = user_score = 0;
    for(row = 0; row < SIZE; row++)
      for(col = 0; col < SIZE; col++)
      {
        comp_score += board[row][col] == comp_c;
        user_score += board[row][col] == player_c;
      }
    printf("The final score is:\n");
    printf("Computer %d\n    User %d\n\n", comp_score, user_score);

    printf("Do you want to play again (y/n): ");
    scanf(" %c", &again);             /* Get y or n             */
  }while(tolower(again) == 'y');      /* Go again on y       */

  printf("\nGoodbye\n");
  return 0;
}

/* Code for definition of display() */

/* Code for definition of valid_moves() */
```

I recommend that you don't run this program yet because you haven't written the code to handle input from the user or moves from the computer. At the moment, it will just loop indefinitely, printing a board with no new moves being made. You'll sort out those parts of the program next.

The variable `player` determines whose turn it is. When `player` is `false`, it's the computer's turn, and when `player` is `true`, it's the player's turn. This is set initially to `true`, and setting `player` to `!player` in the `do-while` loop will alternate who goes first. To determine who takes the next turn, you invert the value of the variable `player` and test the result in the `if` statement, which will alternate between the computer and the player automatically.

The game ends when the number of counters in `no-of_moves` reaches `SIZE*SIZE`, the number of squares on the board. It will also end if `invalid_moves` reaches 2. You set `invalid_moves` to 0 when a valid move is made and increment it each time no valid move is possible. Thus, it will reach 2 if there's no valid option for two successive moves, which means that neither player can go. At the end of a game, you output the final board and the results and offer the option of another game.

You can now add the code to `main()` that will make the player and computer moves:

```
/* Program 9.9 REVERSI An Othello type game */
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>

const int SIZE = 6;                   /* Board size - must be even */
const char comp_c = '@';              /* Computer's counter        */
const char player_c = 'O';            /* Player's counter          */

/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);
void make_move(char board[][SIZE], int row, int col, char player);
void computer_move(char board[][SIZE], bool moves[][SIZE], char player);

int main(void)
{
  char board [SIZE][SIZE] = { 0 };    /* The board           */
  bool moves[SIZE][SIZE] = { false }; /* Valid moves         */
  int row = 0;                        /* Board row index     */
  int col = 0;                        /* Board column index  */
  int no_of_games = 0;                /* Number of games     */
  int no_of_moves = 0;                /* Count of moves      */
  int invalid_moves = 0;              /* Invalid move count  */
  int comp_score = 0;                 /* Computer score      */
  int user_score = 0;                 /* Player score        */
  char y = 0;                         /* Column letter       */
  int x = 0;                          /* Row number          */
  char again = 0;                     /* Replay choice input */

  /* Player indicator: true for player and false for computer */
  bool next_player = true;

  /* Prompt for how to play - as before */

  /* The main game loop */
  do
  {
    /* The player starts the first game */
    /* then they alternate              */
    next_player = !next_player;
    no_of_moves = 4;                  /* Starts with four counters */

    /* Blank all the board squares */
    for(row = 0; row < SIZE; row++)
      for(col = 0; col < SIZE; col++)
        board[row][col] = ' ';

    /* Place the initial four counters in the center */
    board[SIZE/2 - 1][SIZE/2 - 1] = board[SIZE/2][SIZE/2] = 'O';
    board[SIZE/2 - 1][SIZE/2] = board[SIZE/2][SIZE/2 - 1] = '@';
```

```
/* The game play loop */
do
{
  display(board);                   /* Display the board  */
  if(next_player=!next_player)      /* Flip next player */
  { /*   It is the player's turn                     */
    if(valid_moves(board, moves, player_c))
    {
      /* Read player moves until a valid move is entered */
      for(;;)
      {
        printf("Please enter your move (row column): ");
        scanf(" %d%c", &x, &y);    /* Read input        */
        y = tolower(y) - 'a';      /* Convert to column index */
        x--;                       /* Convert to row index    */
        if( x>=0 && y>=0 && x<SIZE && y<SIZE && moves[x][y])
        {
          make_move(board, x, y, player_c);
          no_of_moves++;          /* Increment move count */
          break;
        }
        else
          printf("Not a valid move, try again.\n");
      }
    }
    else                          /* No valid moves */
      if(++invalid_moves<2)
      {
        printf("\nYou have to pass, press return");
        scanf("%c", &again);
      }
      else
        printf("\nNeither of us can go, so the game is over.\n");
  }
  else
  { /* It is the computer's turn                     */
    if(valid_moves(board, moves, '@')) /* Check for valid moves */
    {
      invalid_moves = 0;              /* Reset invalid count   */
      computer_move(board, moves, '@');
      no_of_moves++;                  /* Increment move count  */
    }
    else
    {
      if(++invalid_moves<2)
        printf("\nI have to pass, your go\n"); /* No valid move */
      else
        printf("\nNeither of us can go, so the game is over.\n");
    }
  }
}while(no_of_moves < SIZE*SIZE && invalid_moves<2);

/* Game is over */
display(board);                   /* Show final board */
```

```
    /* Get final scores and display them */
    comp_score = user_score = 0;
    for(row = 0; row < SIZE; row++)
      for(col = 0; col < SIZE; col++)
      {
        comp_score += board[row][col] == comp_c;
        user_score += board[row][col] == player_c;
      }
    printf("The final score is:\n");
    printf("Computer %d\n    User %d\n\n", comp_score, user_score);

    printf("Do you want to play again (y/n): ");
    scanf(" %c", &again);              /* Get y or n             */
  }while(tolower(again) == 'y');       /* Go again on y          */

  printf("\nGoodbye\n");
  return 0;
}

/* Code for definition of display() */

/* Code for definition of valid_moves() */
```

The code to deal with game moves uses two new functions for which you add prototypes. The function `make_move()` will execute a move, and the `computer_move()` function will calculate the computer's move. For the player, you calculate the `moves` array for the valid moves in the `if` statement:

```
        if(valid_moves(board, moves, player_c))
        ...
```

If the return value is positive, there are valid moves, so you read the row number and column letter for the square selected:

```
            printf("Please enter your move: "); /* Prompt for entry          */
            scanf(" %d%c", &x, &y);              /* Read input                */
```

You convert the row number to an index by subtracting 1 and the letter to an index by subtracting `'a'`. You call `tolower()` just to be sure the value in `y` is lowercase. Of course, you must include the `ctype.h` header for this function. For a valid move, the index values must be within the bounds of the array and `moves[x][y]` must be `true`:

```
            if( x>=0 && y>=0 && x<SIZE && y<SIZE && moves[x][y])
            ...
```

If you have a valid move, you execute it by calling the function `make_move()`, which you'll write in a moment (notice that the code won't compile yet, because you make a call to this function without having defined it in the program).

If there are no valid moves for the player, you increment `invalid_moves`. If this is still less than 2, you output a message that the player can't go, and continue with the next iteration for the computer's move. If `invalid_moves` isn't less than 2, however, you output a message that the game is over, and the `do-while` loop condition controlling game moves will be `false`.

For the computer's move, if there are valid moves, you call the `computer_move()` function to make the move and increment the move count. The circumstances in which there are no valid moves are handled in the same way as for the player.

Let's add the definition of the `make_move()` function next. To make a move, you must place the appropriate counter on the selected square and flip any adjacent rows of opponent counters that are bounded at the opposite end by a player counter. You can add the code for this function at the end of the source file—I won't repeat all the other code:

```
/********************************************************************
 * Makes a move. This places the counter on a square and reverses   *
 * all the opponent's counters affected by the move.                *
 * First parameter is the board array.                              *
 * Second and third parameters are the row and column indices.      *
 * Fourth parameter identifies the player.                          *
 ********************************************************************/
void make_move(char board[][SIZE], int row, int col, char player)
{
  int rowdelta = 0;                    /* Row increment              */
  int coldelta = 0;                    /* Column increment           */
  int x = 0;                           /* Row index for searching    */
  int y = 0;                           /* Column index for searching */

  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  board[row][col] = player;            /* Place the player counter   */

  /* Check all the squares around this square */
  /* for the opponents counter                */
  for(rowdelta = -1; rowdelta <= 1; rowdelta++)
    for(coldelta = -1; coldelta <= 1; coldelta++)
    {
      /* Don't check off the board, or the current square */
      if(row + rowdelta < 0 || row + rowdelta >= SIZE ||
         col + coldelta < 0 || col + coldelta >= SIZE ||
                              (rowdelta==0 && coldelta== 0))
        continue;

      /* Now check the square */
      if(board[row + rowdelta][col + coldelta] == opponent)
      {
        /* If we find the opponent, search in the same direction */
        /* for a player counter                                   */
        x = row + rowdelta;            /* Move to opponent */
        y = col + coldelta;            /* square           */

        for(;;)
        {
          x += rowdelta;               /* Move to the      */
          y += coldelta;               /* next square      */

          /* If we are off the board give up */
          if(x < 0 || x >= SIZE || y < 0 || y >= SIZE)
            break;

          /* If the square is blank give up */
          if(board[x][y] == ' ')
            break;
          /* If we find the player counter, go backward from here  */
          /* changing all the opponents counters to player         */
           if(board[x][y] == player)
          {
            while(board[x-=rowdelta][y-=coldelta]==opponent) /* Opponent? */
              board[x][y] = player;   /* Yes, change it */
            break;                    /* We are done    */
          }
        }
      }
    }
}
```

The logic here is similar to that in the `valid_moves()` function for checking that a square is a valid move. The first step is to search the squares around the square indexed by the parameters `row` and `col` for an opponent counter. This is done in the nested loops:

```
  for(rowdelta = -1; rowdelta <= 1; rowdelta++)
    for(coldelta = -1; coldelta <= 1; coldelta++)
    {
      ...
    }
```

When you find an opponent counter, you head off in the same direction looking for a player counter in the indefinite `for` loop. If you fall off the edge of the board or find a blank square, you break out of the `for` loop and continue the outer loop to move to the next square around the selected square. If you do find a player counter, however, you back up, changing all the opponent counters to player counters.

```
          /* If we find the player counter, go backward from here  */
          /* changing all the opponents counters to player         */
          if(board[x][y] == player)
          {
            while(board[x-=rowdelta][y-=coldelta]==opponent) /* Opponent? */
              board[x][y] = player;   /* Yes, change it */
            break;                    /* We are done    */
          }
```

The `break` here breaks out of the indefinite `for` loop.

Now that you have this function, you can move on to the trickiest part of the program, which is implementing the function to make the computer's move. You'll adopt a relatively simple strategy for determining the computer's move. You'll evaluate each of the possible valid moves for the computer. For each valid computer move, you'll determine what the best move is that the player could make and determine a score for that. You'll then choose the computer move for which the player's best move produces the lowest score.

Before you get to write `computer_move()`, you'll implement a couple of helper functions. Helper functions are just functions that help in the implementation of an operation, in this case implementing the move for the computer. The first will be the function `get_score()` that will calculate the score for a given board position. You can add the following code to the end of the source file for this:

```
/*******************************************************************
 * Calculates the score for the current board position for the     *
 * player. player counters score +1, opponent counters score -1    *
 * First parameter is the board array                               *
 * Second parameter identifies the player                           *
 * Return value is the score.                                       *
 *******************************************************************/
int get_score(char board[][SIZE], char player)
{
  int score = 0;                      /* Score for current position */

  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  /* Check all board squares */
  for(int row = 0; row < SIZE; row++)
    for(int col = 0; col < SIZE; col++)
    {
      score -= board[row][col] == opponent; /* Decrement for opponent */
      score += board[row][col] == player;   /* Increment for player   */
    }
  return score;
}
```

This is quite simple. The score is calculated by adding 1 for every player counter on the board, and subtracting 1 for each opponent counter on the board.

The next helper function is `best_move()`, which will calculate and return the score for the best move of the current set of valid moves for a player. The code for this is as follows:

```
/*******************************************************************
 * Calculates the score for the best move out of the valid moves   *
 * for player in the current position.                              *
 * First parameter is the board array                               *
 * Second parameter is the moves array defining valid moves.       *
 * Third parameter identifies the player                            *
 * The score for the best move is returned                          *
 *******************************************************************/
int best_move(char board[][SIZE], bool moves[][SIZE], char player)
{
  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;

  char new_board[SIZE][SIZE] = { 0 }; /* Local copy of board    */
  int score = 0;                      /* Best score             */
  int new_score = 0;                  /* Score for current move */

  /* Check all valid moves to find the best */
  for(int row = 0 ; row<SIZE ; row++)
    for(int col = 0 ; col<SIZE ; col++)
    {
      if(!moves[row][col])            /* Not a valid move?      */
        continue;                     /* Go to the next         */
      /* Copy the board */
      memcpy(new_board, board, sizeof(new_board));

      /* Make move on the board copy */
      make_move(new_board, row, col, player);

      /* Get score for move */
      new_score = get_score(new_board, player);

      if(score<new_score)             /* Is it better?              */
        score = new_score;            /* Yes, save it as best score */
    }
  return score;                       /* Return best score          */
}
```

Remember that you must add function prototypes for both of these helper functions to the other function prototypes before `main()`:

```
/* Function prototypes */
void display(char board[][SIZE]);
int valid_moves(char board[][SIZE], bool moves[][SIZE], char player);
void make_move(char board[][SIZE], int row, int col, char player);
void computer_move(char board[][SIZE], bool moves[][SIZE], char player);
int best_move(char board[][SIZE], bool moves[][SIZE], char player);
int get_score(char board[][SIZE], char player);
```

Step 4

The last piece to complete the program is the implementation of the `computer_move()` function. The code for this is as follows:

```
/*******************************************************************
 * Finds the best move for the computer. This is the move for      *
 * which the opponent's best possible move score is a minimum.     *
 * First parameter is the board array.                             *
 * Second parameter is the moves array containing valid moves.     *
 * Third parameter identifies the computer.                        *
 *******************************************************************/
void computer_move(char board[][SIZE], bool moves[][SIZE], char player)
{
  int best_row = 0;                   /* Best row index          */
  int best_col = 0;                   /* Best column index       */
  int new_score = 0;                  /* Score for current move  */
  int score = 100;                    /* Minimum opponent score  */
  char temp_board[SIZE][SIZE];        /* Local copy of board     */
  bool temp_moves[SIZE][SIZE];        /* Local valid moves array */

  /* Identify opponent */
  char opponent = (player == player_c) ? comp_c : player_c;
  /* Go through all valid moves */
  for(int row = 0; row < SIZE; row++)
    for(int col = 0; col < SIZE; col++)
    {
      if( !moves[row][col] )
        continue;

      /* First make copies of the board array */
      memcpy(temp_board, board, sizeof(temp_board));

      /* Now make this move on the temporary board */
      make_move(temp_board, row, col, player);

      /* find valid moves for the opponent after this move */
      valid_moves(temp_board, temp_moves, opponent);

      /* Now find the score for the opponent's best move */
      new_score = best_move(temp_board, temp_moves, opponent);

      if(new_score<score)            /* Is it worse?                     */
      {                              /* Yes, so save this move           */
        score = new_score;           /* Record new lowest opponent score */
        best_row = row;              /* Record best move row             */
        best_col = col;              /* and column                       */
      }
    }
  /* Make the best move */
  make_move(board, best_row, best_col, player);
}
```

This isn't difficult with the two helper functions. Remember that you're going to choose the move for which the opponent's subsequent best move is a minimum.

In the main loop that is controlled by the counters `row` and `col`, you make each valid move, in turn, on the copy of the current board that's stored in the local array `temp_board`. After each move, you call the `valid_moves()` function to calculate the valid moves for the opponent in that position and store the results in the `temp_moves` array. You then call the `best_move()` function to get the score for the best opponent move from the valid set stored in the array `temp_moves`. If that score is less than any previous score, you save the score, the row, and the column index for that computer move, as a possible best move.

The variable `score` is initialized with a value that's higher than any possible score, and you go about trying to minimize this (because it's the strength of the opponent's next move) to find the best possible move for the computer. After all of the valid computer moves have been tried, `best_row` and `best_col` contain the row and column index for the move that minimizes the opponent's next move. You then call `make_move()` to make the best move for the computer.

You can now compile and execute the game. The game starts something like this:

```
    a   b   c   d   e   f
  +---+---+---+---+---+---+
 1|   |   |   |   |   |   |
  +---+---+---+---+---+---+
 2|   |   |   |   |   |   |
  +---+---+---+---+---+---+
 3|   |   | O | @ |   |   |
  +---+---+---+---+---+---+
 4|   |   | @ | O |   |   |
  +---+---+---+---+---+---+
 5|   |   |   |   |   |   |
  +---+---+---+---+---+---+
 6|   |   |   |   |   |   |
  +---+---+---+---+---+---+
Please enter your move: 3e
    a   b   c   d   e   f
  +---+---+---+---+---+---+
 1|   |   |   |   |   |   |
  +---+---+---+---+---+---+
 2|   |   |   |   |   |   |
  +---+---+---+---+---+---+
 3|   |   | O | O | O |   |
  +---+---+---+---+---+---+
 4|   |   | @ | O |   |   |
  +---+---+---+---+---+---+
 5|   |   |   |   |   |   |
  +---+---+---+---+---+---+
 6|   |   |   |   |   |   |
  +---+---+---+---+---+---+
```

The computer doesn't play too well because it looks only one move ahead, and it doesn't have any favoritism for edge and corner cells.

Also, the board is only 6 × 6. If you want to change the board size, just change the value of `SIZE` to another even number. The program will work just as well.

Summary

If you've arrived at this point without too much trouble, you're well on your way to becoming a competent C programmer. This chapter and the previous one have covered all you really need to write well-structured C programs. A functional structure is inherent to the C language, and you should keep your functions short with a well-defined purpose. This is the essence of good C code. You should now be able to approach your own programming problems with a functional structure in mind right from the outset.

Don't forget the flexible power that pointers give you as a C programmer. They can greatly simplify many programming problems, and you should frequently find yourself using them as function arguments and return values. After a while, it will be a natural inclination. The real teacher is experience, so going over the programs in this chapter again will be extremely useful if you don't feel completely confident. And once you feel confident with what's in this book, you should be raring to have a go at some problems of your own.

There's still one major new piece of language territory in C that you have yet to deal with, and it's all about data and how to structure it. You'll look at data in Chapter 11. But before you do that, you need to cover I/O in rather more detail than you have so far. Handling input and output is an important and fascinating aspect of programming, so that's where you're headed next.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 9-1. A function with the prototype

```
double power(double x, int n);
```

should calculate and return the value of x^n. That is, the expression `power(5.0, 4)` will evaluate `5.0 * 5.0 * 5.0 * 5.0`, which will result in the value 625.0.

Implement the `power()` function as a recursive function (so it should call itself) and demonstrate its operation with a suitable version of `main()`.

Exercise 9-2. Implement functions with the prototypes

```
double add(double a, double b);        /* Returns a+b */
double subtract(double a, double b);   /* Returns a-b */
double multiply(double a, double b);   /* Returns a*b */
double array_op(double array[], int size, double (*pfun)(double,double));
```

The parameters for the `array_op()` function are the array to be operated on, the number of elements in the array, and a pointer to a function defining the operation to be applied between successive elements. The `array_op()` function should be implemented so that when the `subtract()` function is passed as the third argument, the function combines the elements with alternating signs. So for an array with four elements, `x1`, `x2`, `x3`, and `x4`, it computes the value of `x1 - x2 + x3 - x4`.

Demonstrate the operation of these functions with a suitable version of `main()`.

Exercise 9-3. Define a function that will accept an array of pointers to strings as an argument and return a pointer to a string that contains all the strings joined into a single string, each terminated by a newline character. If an original string in the input array has newline as its last character, the function shouldn't add another to the string. Write a program to demonstrate this function in operation by reading a number of strings from the keyboard and outputting the resultant combined string.

Exercise 9-4. Implement a function that has the prototype

```
char *to_string(int count, double first, ...);
```

This function should return a string that contains string representations of the second and subsequent arguments, each to two decimal places and separated by commas. The first argument is a count of the number of arguments that follow. Write a suitable version of `main()` to demonstrate the operation of your function.