/* From Wirth's "Knight's Tour" Pascal Program  
   * crude translation to ANSI C by Kevin Karplus  
   * Cosmetic Improvements for CE185 by David Dahle, January 30, 1997  
   *  
   * This program moves a knight around a chess board using the rules of  
   * chess. Given an initial position for the knight, the program tries  
   * to determine a way the knight could move and land on every space on  
   * the board exactly once. If there are multiple ways of accomplishing  
   * this, then the program will only find one. If there is no solution,  
   * then the program will output 'no solution'. If a solution is  
   * found, then the program will output an array of numbers, where the  
   * position in the array corresponds to a space on the board, and the  
   * number corresponds to the move that puts the knight on that space.  
   */  
#include <stdio.h>  
/* The following defines the size of the chess board. ROWS defines  
   * and number of rows (left to right orientation), and COLUMNS defines  
   * the number of columns (top to bottom orientation).  
*/  
#define ROWS 5  
#define COLUMNS 5  
#define BOARD_SIZE (ROWS*COLUMNS)  
/* The following define the starting position on the chess board for  
   * the knight. START_ROW indicates the starting row and START_COLUMN  
   * indicates the starting column. The position counting starts from 0,  
   * so START_ROW must be in the range 0 to ROWS - 1 and START_COLUMN must  
   * be in the range 0 to COLUMNS - 1.  
   *  
   * START_ROW set to 0 and START_COLUMN set to 0 indicates the top-left  
   * corner of the board. START_ROW set to ROWS - 1 and START_COLUMN set  
   * to COLUMNS - 1 indicates the lower-right corner of the board.  
*/  
#define START_ROW 0  
#define START_COLUMN 0  
/* The following tables enumerate the eight possible moves a knight can  
   * make. A knight is constrained to move in an 'L' path. For example,  
   * a knight could move two spaces forward and one space to the left,  
   * marking out the shape of an 'L'.  
   *  
   * These values are used as offsets from the current position. For  
   * example, the first possible move for the knight is one space to the  
   * right and two spaces down. Boundary checking is done in the code to  
   * ensure that the new position is still on the board.  
*/  
#define POSSIBLE_MOVES 8  
int next_row[POSSIBLE_MOVES] = { 2, 1, -1, -2, -2, -1, 1, 2 };  
int next_column[POSSIBLE_MOVES] = { 1, 2, 2, 1, -1, -2, -2, -1 };  
/* The following 2-dimensional array allocates storage for the chess  
   * board itself. The first index specifies the row number and the  
   * second index specifies the column number. Each element stores an  
   * integer greater than or equal to zero. A zero indicates that the  
   * knight has not yet landed on the corresponding space. A number  
   * greater than zero indicates the move number that caused the knight  
   * to land on that space. Thus, if the algorithm is successful, this  
   * table will contain the moves the knight should make to cover the  
   * board landing on each space exactly once.  
*/  
int chess_board[ROWS][COLUMNS];  
/* The following are used by DoKightsTour, PrintKnightsTourResult and  
   * MoveKnight to determine if a solution was found.  
*/  
#define SEARCH_SUCCESSFUL 1  
#define SEARCH_FAILED 0
/* Function: MoveKnight
 * Purpose: This function searches for a sequence of moves that will
 *         cause a knight, following the rules of chess, to land on
 *         every space on a chess board exactly once.
 * Inputs: move_number (int) - This specifies the current move number.
 *         row (int), column (int) - Together these specify the current
 *         position of the knight on the chess board.
 * Outputs: SEARCH_SUCCESSFUL - The knight was able to move.
 *          SEARCH_FAILED - The knight was not able to move.
 * Globals: chess_board (int [ROWS][COLUMNS]) - The elements in this
 *         array are modified. Each element represents a space on
 *         the chess board. The numbers stored in the array
 *         indicate the move number that moves the knight to
 *         the corresponding space on the chess board.
 *         next_row (int [POSSIBLE_MOVES]),
 *         next_column (int [POSSIBLE_MOVES]) - Together these specify
 *         the possible moves a knight can make. These tables contain
 *         offsets that are added to a current position to obtain the
 *         next position. They are not modified. */
int MoveKnight(int move_number, int row, int column)
{
    /* This function works recursively. Starting at the position
    * specified in row and column, the function attempts to move the
    * knight to one of the eight spaces that it can move to under the
    * rules of chess. If one of these spaces has not already been
    * moved to, then MoveKnight is called recursively starting at
    * that space. This recursive calling continues until all spaces
    * have been landed on. If at some point, it is not possible to
    * move to another space, then the function returns failure, and
    * one of the earlier calls to MoveKnight continues the search by
    * choosing another of the eight possible moves available.
    */

    int next_move, /* index into the next move arrays */
        new_row, new_column; /* used compute the knight’s next move */
    int status; /* stores the return value from MoveKnight */

    /* Loop through each possible move until we find a move that works. */
    for (next_move = 0; next_move < POSSIBLE_MOVES; next_move++)
    {
        /* Create new position for knight using offsets in the
         * the next_row and next_column tables.
         */
        new_row = row + next_row[next_move];
        new_column = column + next_column[next_move];

        /* If this moves keeps us on the board... */
        if((new_row >= 0 && new_row < ROWS) &&
           (new_column >= 0 && new_column < COLUMNS))
        {
            /* If the knight has not already landed on this space... */
            if(chess_board[new_row][new_column] == 0)
            {
                /* Indicate that the knight has visited this space by
                 * marking the space with the current move number.
                 */
                chess_board[new_row][new_column] = move_number;

                /* Have we landed on all the spaces? */
                if (move_number >= BOARD_SIZE)
                {
                    /* Yes, then we have found a solution. This is
                     * point that ends the recursive calls when a
                     * solution is found.
                     */
                    return SEARCH_SUCCESSFUL;
                }

                /* Move the knight to the new position and continue the search. */
                status = MoveKnight(move_number + 1, new_row, new_column);
                if (status == SEARCH_SUCCESSFUL)
                {
                    return SEARCH_SUCCESSFUL;
                }

                /* Moving the knight to the new space on the board
                 * didn’t work so mark the space as unvisited.
                 */
                chess_board[new_row][new_column] = 0;
            }
        }
    }

    /* There was no place the knight could move from the position
     * specified in row and column.
     */
    return SEARCH_FAILED;
}
/* Function: DoKnightsTour
 * Purpose: Implements the Knight’s Tour algorithm.
 * Inputs: start_row (int), start_column (int) - Together these specify
 * the starting space of the knight on the chess board.
 * Outputs: SEARCH_SUCCESSFUL - The knight was able to move to every space
 * on the board exactly once.
 * SEARCH_FAILED - The knight was not able to move to every space
 * one the board exactly once.
 * Globals: chess_board (int [ROWS][COLUMNS])
 * The elements in this array are modified. Each element
 * represents a space on the chess board. The numbers stored
 * in the array indicate the move number that put the
 * knight on the corresponding space.
 */
int DoKnightsTour(int start_row, int start_column)
{
    /* This function initializes the chess board, sets the initial
     * position of the knight on the board, and calls MoveKnight to
     * do the work of finding a solution.
     */
    int row, column; /* indices into chess_board */
    int status; /* return value from MoveKnight */

    /* Initialize the chess board. */
    for(row = 0; row < ROWS; row++)
    {
        for(column = 0; column < COLUMNS; column++)
        {
            /* A zero indicates the space has not yet been visited. */
            chess_board[row][column] = 0;
        }
    }

    /* Mark the starting space with the first move. */
    chess_board[start_row][start_column] = 1;

    /* Move the knight around the board, searching for a way to land
     * on each space exactly once. The first parameter passed to this
     * function is the next move number. The move number is started at
     * 2 because the initial position of the knight is counted as move 1.
     */
    status = MoveKnight(2, start_row, start_column);

    return (status);
}
/* Function: PrintKnightsTourResult
 * Purpose : Output the result of the knight’s tour to the screen.
 * Inputs  : status - The return value from DoKnightsTour. It is used to determine if a solution was found.
 * Outputs : The function has no return value.
 *          : The result is printed to the screen.
 * Global : chess_board (int [ROWS][COLUMNS])
 *          : The contents of this array are printed to the screen; they are not modified.
 */
void PrintKnightsTourResult(int status)
{
    int row, column; /* indices into chess_board */
    /* If the search was successful, then print an array of numbers corresponding to how the knight should move to solve the problem. Otherwise, print 'no solution'. */
    if (status == SEARCH_SUCCESSFUL)
    {
        for(row = 0; row < ROWS; row++)
        {
            for(column = 0; column < COLUMNS; column++)
            {
                printf("%d ", chess_board[row][column]);
            }
            printf("\n");
        }
    }
    else
    {
        printf("no solution\n");
    }
    return;
}
/* Function: main
 * Purpose : Perform the knight’s tour and output the results.
 * Inputs  : No explicit parameters are used.
 *           This function uses the #defines of START_ROW and
 *           START_COLUMN to determine the initial location of the
 *           knight on the chess board.
 * Outputs : The results of the knight’s tour is printed to the screen if
 *           a solution was found, or ‘no solution’ if no solution to the
 *           problem exists for the
 */
int main()
{
    int status; /* stores result of DoKnightsTour */
    status = DoKnightsTour(START_ROW, START_COLUMN);
    PrintKnightsTourResult(status);
    exit(0);
}

/* end of file ‘knight.c’ */