**MARBLE SOLITAIRE**

DAA Course Project

**Bachelor of Technology**

In

**Computer Science and Engineering**

**School of Engineering and Sciences**

Submitted by

**RIDHI GUNTUR – AP22110011467**

**A picture containing text

Description automatically generated**

**SRM University–AP**

**Neerukonda, Mangalagiri, Guntur**

**Andhra Pradesh – 522 240**

**[April, 2024]**

**Statement of the Problem:**

Developing a computer program in C++ to implement the game of Marble Solitaire using dynamic algorithm, allowing players to play the game interactively.



**Explanation of the Problem:**

Marble Solitaire is a single-player game played on a board with a grid of holes. The player's objective is to remove marbles from the board by jumping them over adjacent marbles, until only one marble remains. The challenge lies in making strategic moves to minimize the number of marbles left on the board.

**Programming Techniques Used:**

This program demonstrates the use of functions, arrays, loops, conditional statements, and error handling techniques to implement a simple game logic. It also showcases program by breaking down the logic into separate functions for better organization and readability.

**Algorithm of the code:**

1. Initialize the Game Board:

Create a 2D array to represent the game board.

Populate the array with the initial marble positions based on the rules of Marble Solitaire.

1. Display the Initial Board:

Write a function to display the current state of the game board.

Print the initial board to the console.

1. Check for Valid Moves:

Write a function to check if a move is valid.

Iterate through the board and check each marble position to see if there are any valid moves available.

1. Player Input and Move Execution:

Prompt the player to enter the location of the marble they want to move.

Verify if the entered location is valid and corresponds to a marble.

If the location is valid, prompt the player to enter the destination for the move.

Check if the move is valid using the function from step 3.

If the move is valid, execute it by updating the board accordingly.

If the move is not valid, display an error message and prompt the player to enter a valid move.

1. Display Updated Board:

After each valid move, display the updated state of the game board.

1. Check for Winning Condition:

Write a function to check if the winning condition is met.

Check if there is only one marble left on the board.

If the winning condition is met, display a message indicating that the player has won and end the game.

If the winning condition is not met, continue the game.

1. Repeat Steps 3-6 Until No Valid Moves Remain:

Continue prompting the player for moves and updating the board until there are no valid moves left or the winning condition is met.

1. End of Game:

If the player cannot make any more valid moves and has not won the game, display a message indicating that the player cannot win the game.

1. Optional: Allow Player to Restart or Exit the Game:

After the game ends, prompt the player to choose whether they want to play again or exit the game.

1. Algorithm Termination:

End the algorithm.

**Program Code:**

#include <iostream>

using namespace std;

const int BOARD\_SIZE = 7;

// Function to display the board

void display\_board(char board[BOARD\_SIZE][BOARD\_SIZE]) {

for (int i = 0; i < BOARD\_SIZE; ++i) {

for (int j = 0; j < BOARD\_SIZE; ++j) {

cout << board[i][j] << ' ';

}

cout << endl;

}

}

// Function to check if there are any possible moves left

bool check\_possible\_moves(char board[BOARD\_SIZE][BOARD\_SIZE]) {

for (int i = 0; i < BOARD\_SIZE; ++i) {

for (int j = 0; j < BOARD\_SIZE; ++j) {

if (board[i][j] == '0') {

// Check all four directions for possible moves

if ((i >= 2 && board[i - 2][j] == '-' && board[i - 1][j] == '0') ||

(i < BOARD\_SIZE - 2 && board[i + 2][j] == '-' && board[i + 1][j] == '0') ||

(j >= 2 && board[i][j - 2] == '-' && board[i][j - 1] == '0') ||

(j < BOARD\_SIZE - 2 && board[i][j + 2] == '-' && board[i][j + 1] == '0')) {

return true; // Found at least one possible move

}

}

}

}

return false; // No possible moves found

}

// Function to move a marble on the board

void move\_marble(char board[BOARD\_SIZE][BOARD\_SIZE], int from\_row, int from\_col, int to\_row, int to\_col) {

// Check if the move is valid

if (from\_row < 0 || from\_row >= BOARD\_SIZE || from\_col < 0 || from\_col >= BOARD\_SIZE ||

to\_row < 0 || to\_row >= BOARD\_SIZE || to\_col < 0 || to\_col >= BOARD\_SIZE ||

board[from\_row][from\_col] != '0') {

cout << "Invalid move!" << endl;

return;

}

// Check if the move is in a straight line

if (!(from\_row == to\_row || from\_col == to\_col)) {

cout << "Marbles can only move in straight lines!" << endl;

return;

}

// Check if the destination is empty

if (board[to\_row][to\_col] != '-') {

cout << "Destination is not empty!" << endl;

return;

}

// Check if there is a marble to jump over

int jump\_row = (from\_row + to\_row) / 2;

int jump\_col = (from\_col + to\_col) / 2;

if (board[jump\_row][jump\_col] != '0') {

cout << "You can only jump over another marble!" << endl;

return;

}

// Perform the move

board[to\_row][to\_col] = '0';

board[jump\_row][jump\_col] = '-';

board[from\_row][from\_col] = '-';

// Display the updated board

cout << "Moved marble from (" << from\_row << ", " << from\_col << ") to (" << to\_row << ", " << to\_col << ")" << endl;

display\_board(board);

// Check for winning condition

int remaining\_marbles = 0;

for (int i = 0; i < BOARD\_SIZE; ++i) {

for (int j = 0; j < BOARD\_SIZE; ++j) {

if (board[i][j] == '0') {

remaining\_marbles++;

}

}

}

if (remaining\_marbles == 1) {

cout << "Congratulations! You have won!" << endl;

exit(0); // Exit the program

}

// Check for no possible moves left

if (!check\_possible\_moves(board)) {

cout << "No possible moves left. You lose!" << endl;

exit(0); // Exit the program

}

}

int main() {

// Initialize the board

char board[BOARD\_SIZE][BOARD\_SIZE] = {

{'X', 'X', '0', '0', '0', 'X', 'X'},

{'X', 'X', '0', '0', '0', 'X', 'X'},

{'0', '0', '0', '0', '0', '0', '0'},

{'0', '0', '0', '-', '0', '0', '0'},

{'0', '0', '0', '0', '0', '0', '0'},

{'X', 'X', '0', '0', '0', 'X', 'X'},

{'X', 'X', '0', '0', '0', 'X', 'X'}

};

// Display the initial board

display\_board(board);

// Game loop

while (true) {

// Ask for user input and move marble

int from\_row, from\_col, to\_row, to\_col;

cout << "Enter the row and column of the marble to move (0-6): ";

cin >> from\_row >> from\_col;

if (from\_row < 0 || from\_row >= BOARD\_SIZE || from\_col < 0 || from\_col >= BOARD\_SIZE) {

cout << "Invalid marble position!" << endl;

continue;

}

int direction;

cout << "Enter the direction to move (1: up, 2: down, 3: left, 4: right): ";

cin >> direction;

to\_row = from\_row;

to\_col = from\_col;

switch (direction) {

case 1: // up

to\_row -= 2;

break;

case 2: // down

to\_row += 2;

break;

case 3: // left

to\_col -= 2;

break;

case 4: // right

to\_col += 2;

break;

default:

cout << "Invalid direction!" << endl;

continue;

}

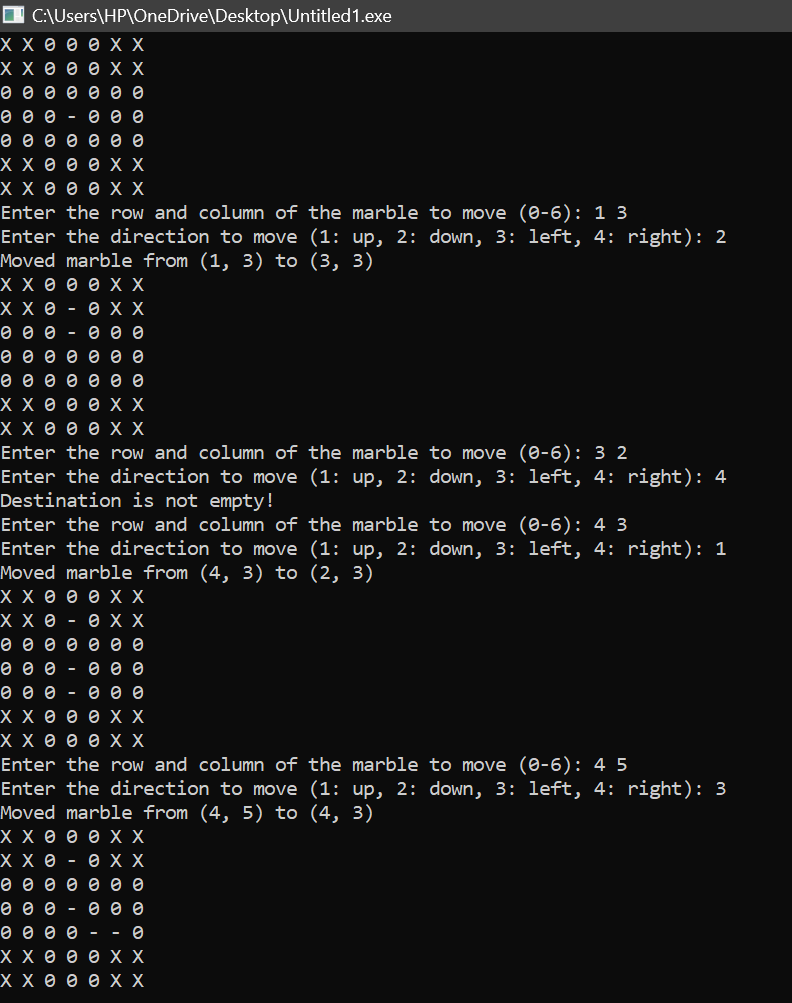
move\_marble(board, from\_row, from\_col, to\_row, to\_col);

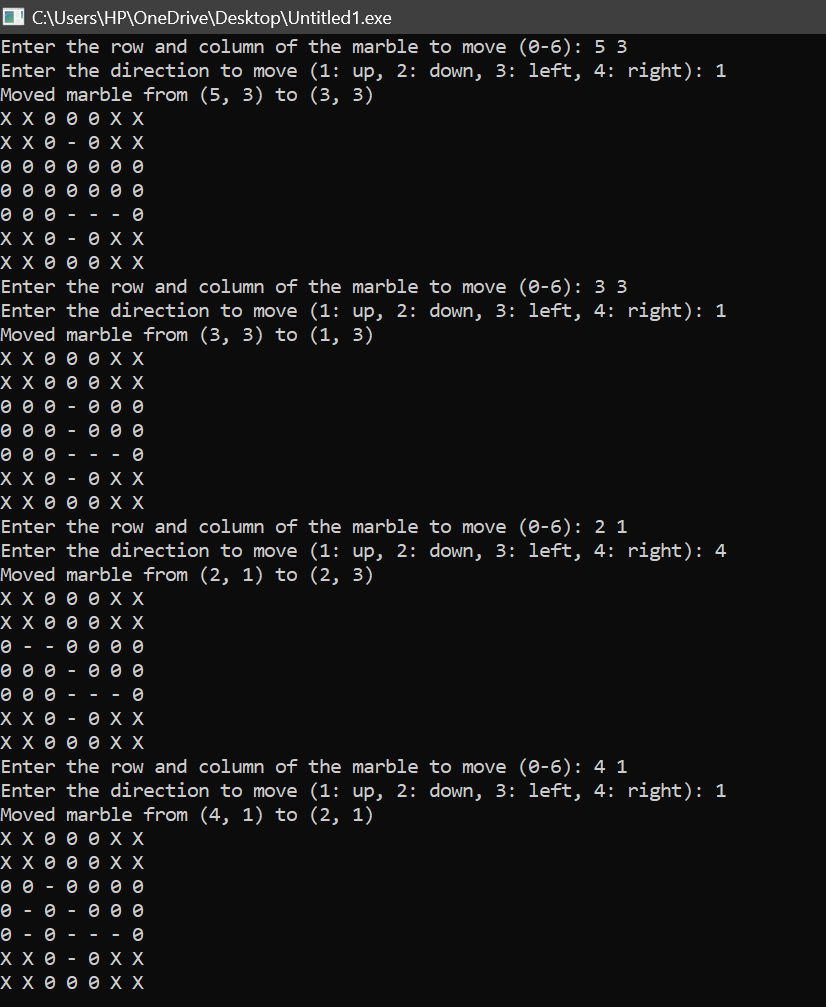
}

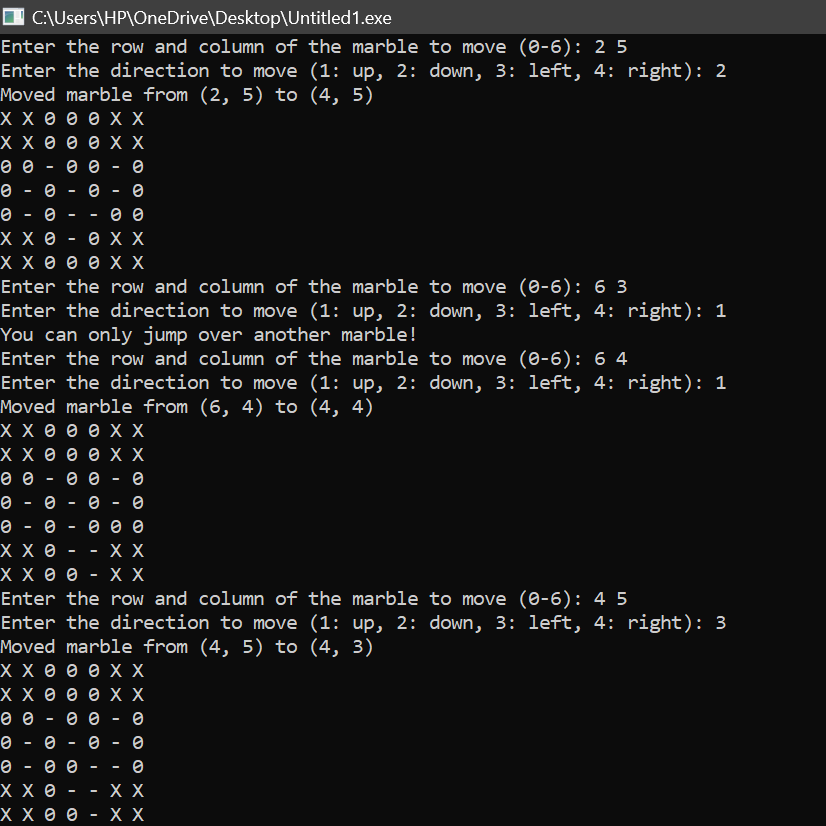
return 0;

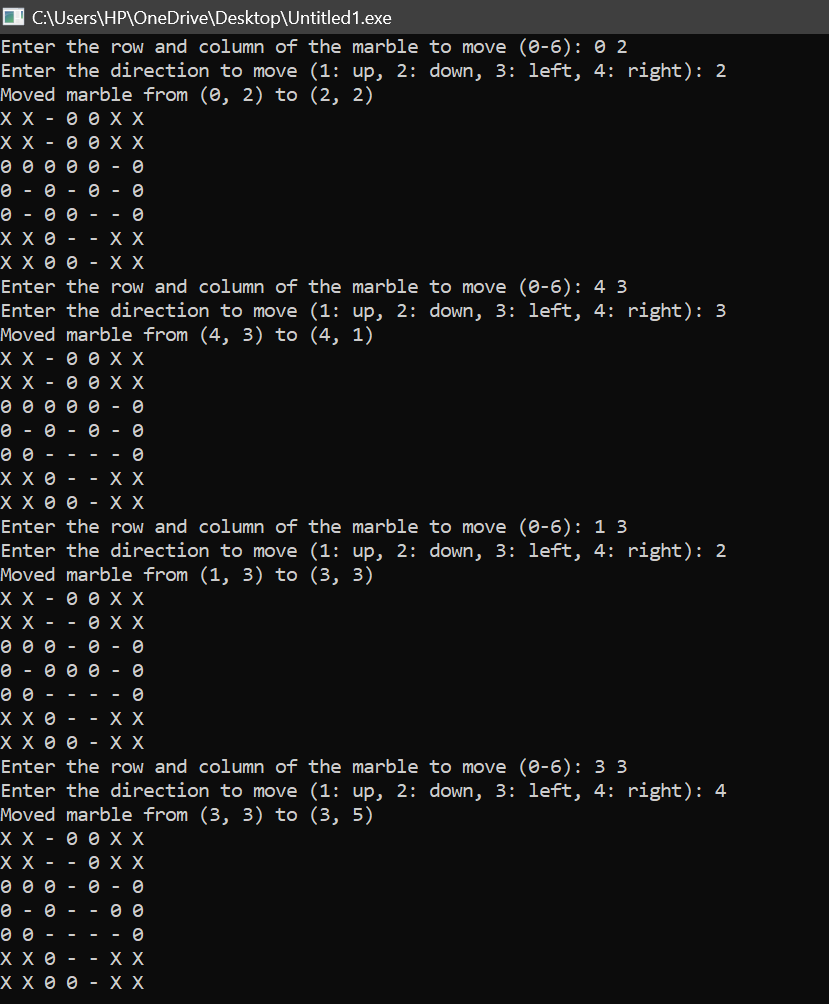
}

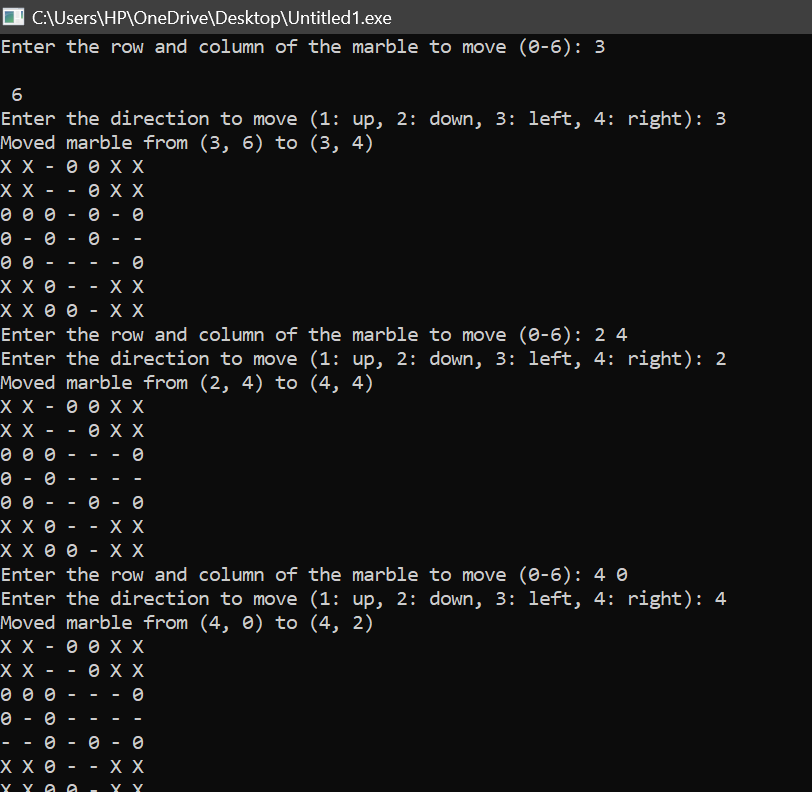
**Output of the Code:**

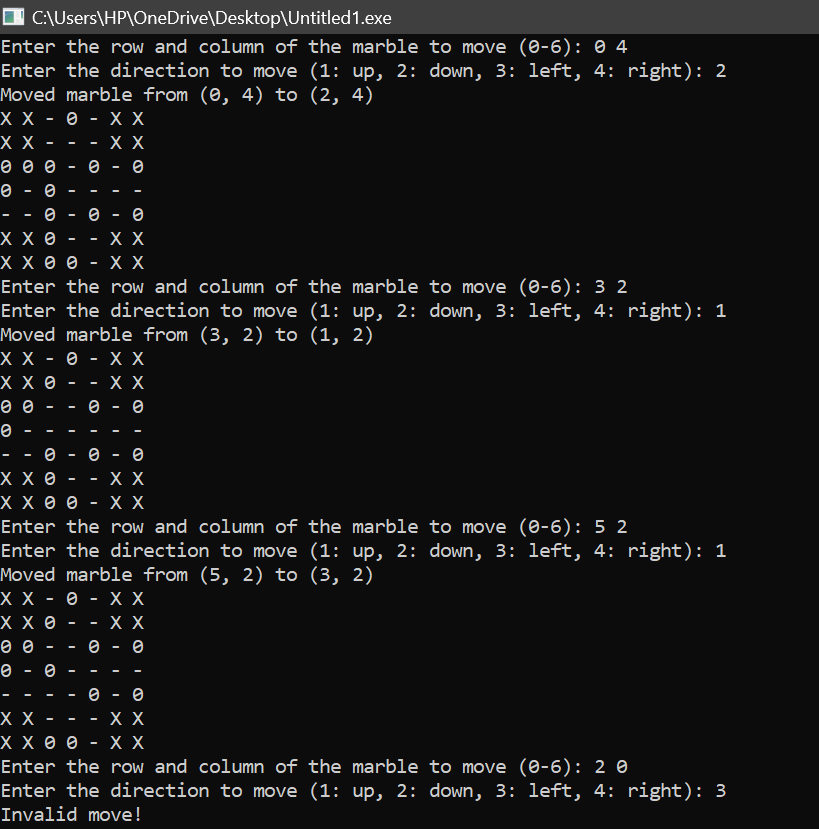


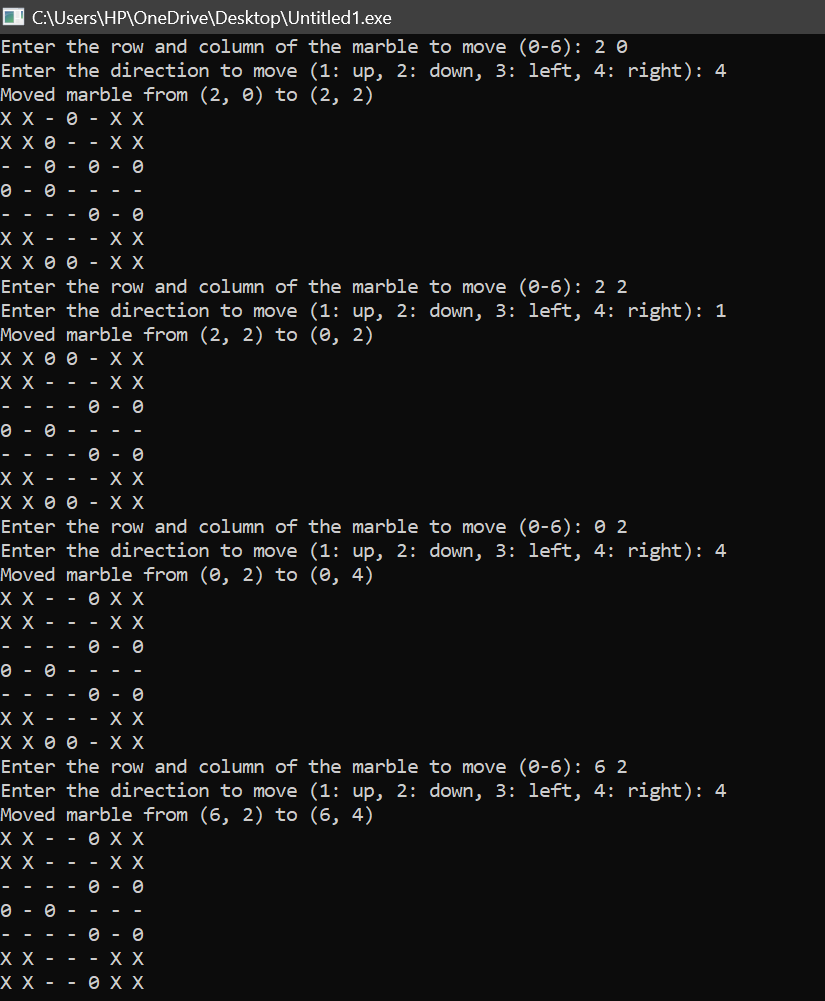


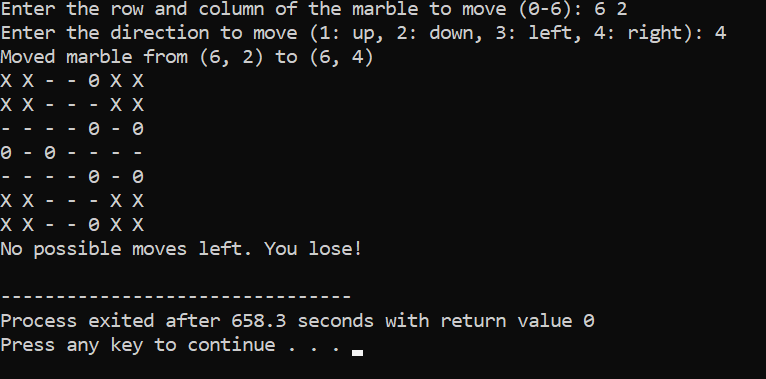












**Explanation of the code:**

1. This is the initial state of the game board, displayed using the display\_board function. 'X' represents barriers, '0' represents marbles, and '-' represents empty spaces.

*X X 0 0 0 X X*

*X X 0 0 0 X X*

*0 0 0 0 0 0 0*

*0 0 0 - 0 0 0*

*0 0 0 0 0 0 0*

*X X 0 0 0 X X*

*X X 0 0 0 X X*

1. The program prompts the user to input the row and column of the marble they want to move.

*Enter the row and column of the marble to move (0-6):*

1. If the user inputs an invalid marble position (e.g., a position outside the board boundaries), the program displays this error message and prompts for input again.

*Invalid marble position!*

1. After entering a valid marble position, the user is prompted to input the direction in which they want to move the marble.

*Enter the direction to move (1: up, 2: down, 3: left, 4: right):*

1. If the user inputs an invalid direction (not 1, 2, 3, or 4), the program displays this error message and prompts for input again.

*Invalid direction!*

1. If the user inputs a valid marble position and direction, and the move is valid according to the game rules, the program executes the move and displays the updated board. In this case, the marble at position (3, 2) is moved to position (3, 4), jumping over an empty space.

*Moved marble from (3, 2) to (3, 4)*

*X X 0 0 0 X X*

*X X 0 - 0 X X*

*0 0 0 - 0 0 0*

*0 0 - - 0 0 0*

*0 0 0 0 0 0 0*

*X X 0 0 0 X X*

*X X 0 0 0 X X*

1. If there are no possible moves left on the board, the program displays this message, indicating that the player has lost the game.

*No possible moves left. You lose!*