

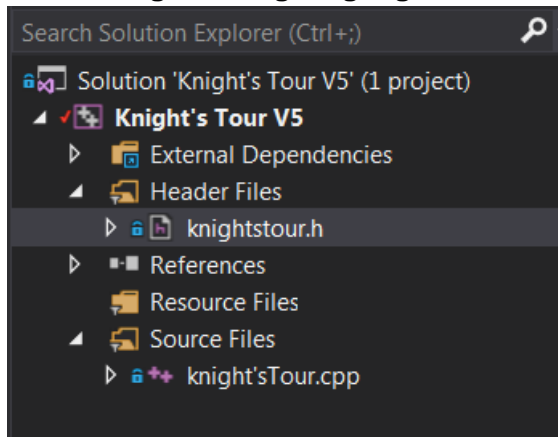
Knight's Tour

Dylan Gijsbertsen

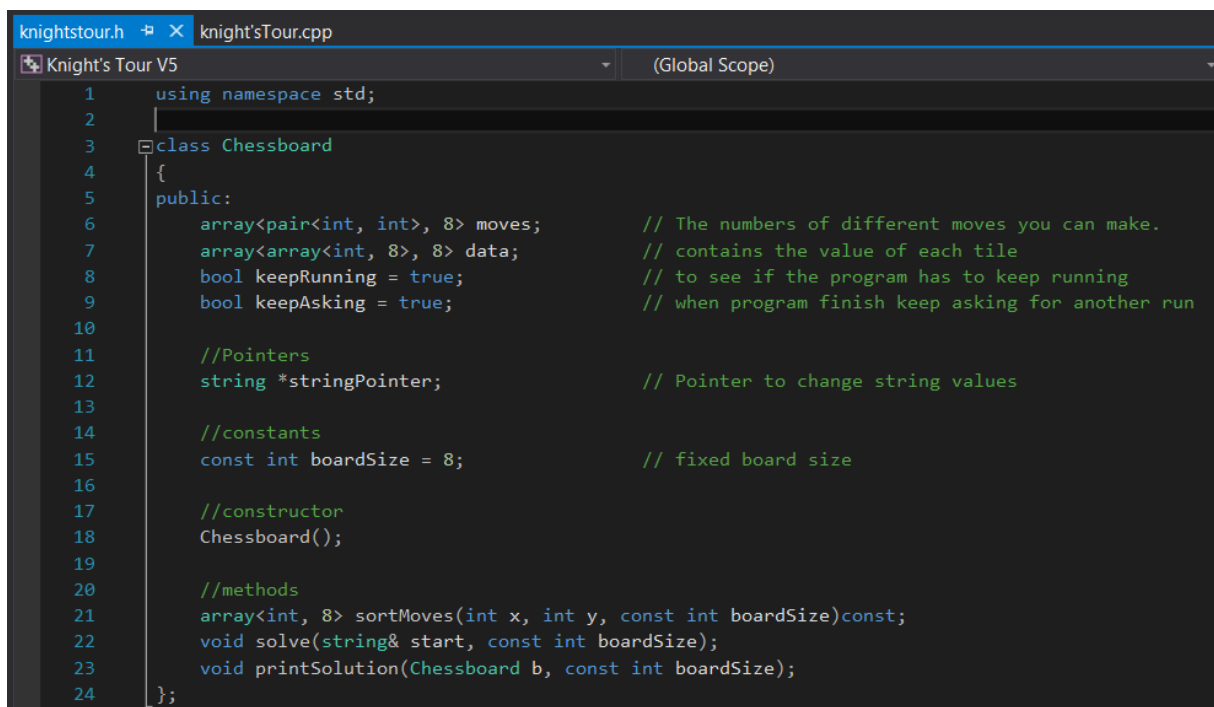
Studentnumber: 500687199

Requirements:

1. Programming language has to be C++



I created a new Visual C++ project (Win32 console application) and add a new C++ class called knight'stour.cpp. I also made a header file to set my variables and methods of the class *Chessboard*.



2. The program makes use of pointers and references

```
//Pointers
string *stringPointer;           // Pointer to change string values
```

This pointer I use to change input that the user can fill in.

```
chessBoard.stringPointer = &inputStart;

printf("\nchoose a letter where you start

//put in start position
cin >> *chessBoard.stringPointer;
```

Now I refer my pointer to the inputStart string and in the solve method I give that input value to the parameter.

```
chessBoard.solve(inputStart, chessBoard.boardSize);           //Solve knight's tour
chessBoard.printSolution(chessBoard, chessBoard.boardSize);   //print the solution
```

```
void Chessboard::solve(string &start, const int boardSize)
{
    //reset all values to 0
    for (int v = 0; v < boardSize; ++v)
        for (int u = 0; u < boardSize; ++u)
            data[v][u] = 0;

    // first start position of the knight
    int x0 = start[0] - 'a';
    int y0 = boardSize - (start[1] - '0');
    data[y0][x0] = 1;
```

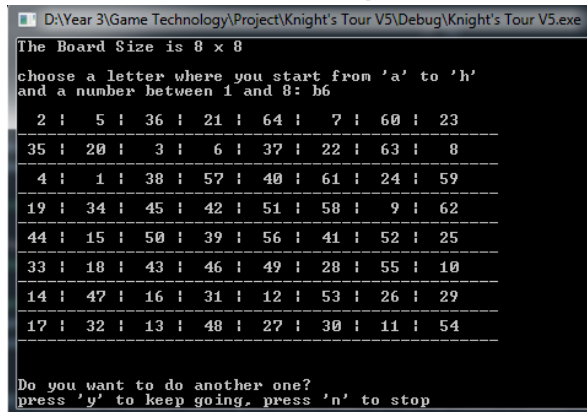
The solve method reference to the start string what is the inputStart.

```
string y = "y";           //yes
string n = "n";           //no
string input;
chessBoard.stringPointer = &input;
cin >> *chessBoard.stringPointer;

if (input == y) // do the tour again
{
    chessBoard.keepAsking = false;
    chessBoard.keepRunning = true;
    printf("\n\n");
}
else if (input == n) // Stop application
{
    return 0;
}
else // error message
{
    printf("ERROR: wrong input!");
    chessBoard.keepAsking = true;
}
```

When the program has solved the tour it will ask you to do it again or quit. Now I refer my pointer to another string called *input* and change that value. After the input it will check if you put in the right letter to keep going if so the program runs again else the program will quit and if the user put in the wrong letter it will show the user a error and has to give the right letter.

3. There is a visual representation



```
D:\Year 3\Game Technology\Project\Knight's Tour V5\Debug\Knight's Tour V5.exe
The Board Size is 8 x 8
choose a letter where you start from 'a' to 'h'
and a number between 1 and 8: b6
 2 | 5 | 36 | 21 | 64 | 7 | 60 | 23
35 | 20 | 3 | 6 | 37 | 22 | 63 | 8
 4 | 1 | 38 | 57 | 40 | 61 | 24 | 59
19 | 34 | 45 | 42 | 51 | 58 | 9 | 62
44 | 15 | 50 | 39 | 56 | 41 | 52 | 25
33 | 18 | 43 | 46 | 49 | 28 | 55 | 10
14 | 47 | 16 | 31 | 12 | 53 | 26 | 29
17 | 32 | 13 | 48 | 27 | 30 | 11 | 54

Do you want to do another one?
press 'y' to keep going, press 'n' to stop
```

This is the visual output that the program shows. I did this with my `printSolution` method. It will loop through the board size and print for every row the values in the columns after a row has come to the end it will go down one row and print a line. This will keep going until there is a 8x8 board with all the values.

```
// print the solution that is found with a width field of 3
void Chessboard::printSolution(Chessboard b, const int boardSize)
{
    for (int v = 0; v < boardSize; ++v)
    {
        for (int u = 0; u < boardSize; ++u)
        {
            if (u != 0)
            {
                printf(" | ");
            }
            cout << setw(3) << b.data[v][u];
        }
        printf("\n");
        printf("-----\n");
    }
}
```

4. the board is at least 5 x 5 in size

```
knightsTour.h  X knight'sTour.cpp
Knight's Tour V5  Chessboard
1 using namespace std;
2
3 class Chessboard
4 {
5 public:
6     array<pair<int, int>, 8> moves;           // The numbers of different moves you can make.
7     array<array<int, 8>, 8> data;           // contains the value of each tile
8     bool keepRunning = true;               // to see if the program has to keep running
9     bool keepAsking = true;               // when program finish keep asking for another run
10
11     //Pointers
12     string *stringPointer;                 // Pointer to change string values
13
14     //constants
15     const int boardSize = 8;               // fixed board size
16
17     //constructor
18     Chessboard();
19
20     //methods
21     array<int, 8> sortMoves(int x, int y, const int boardSize) const;
22     void solve(string& start, const int boardSize);
23     void printSolution(Chessboard b, const int boardSize);
24 };
```

I've made a constant int variable called boardSize and initialized 8 to it, so the board size will be 8x8.

```
chessBoard.solve(inputStart, chessBoard.boardSize);           //Solve knight's tour
chessBoard.printSolution(chessBoard, chessBoard.boardSize);    //print the solution
```

In my solve and printSolution method I have a parameter for the board size and I give the const value to the parameter.

```
// print the solution that is found with a width field of 3
void Chessboard::printSolution(Chessboard b, const int boardSize)
{
    for (int v = 0; v < boardSize; ++v)
    {
        for (int u = 0; u < boardSize; ++u)
        {
            if (u != 0)
            {
                printf(" | ");
            }
            cout << setw(3) << b.data[v][u];
        }
        printf("\n");
        printf("-----\n");
    }
}
```

This is the printSolution method I put here two for loops to loop through the size of the board to print the result. For my solve method I just the board size to calculate the moves and position of the knight.

5. The knight has to move according to its movement rules

```
public:
    array<pair<int, int>, 8> moves;           // The numbers of different moves you can make.
```

This array moves has a length of 8 because a knight can make 8 different moves. For every index place I make a pair of two integers for a combination that the knight can move.

```
Chessboard::Chessboard()
{
    // all the moves a knight can make.
    moves[0] = make_pair(2, 1);
    moves[1] = make_pair(1, 2);
    moves[2] = make_pair(-1, 2);
    moves[3] = make_pair(-2, 1);
    moves[4] = make_pair(-2, -1);
    moves[5] = make_pair(-1, -2);
    moves[6] = make_pair(1, -2);
    moves[7] = make_pair(2, -1);
}
```

In the constructor I set for every index a move that a knight can make by pairing them together.

```
array<int, 8> Chessboard::sortMoves(int x, int y, const int boardSize) const
{
    array<tuple<int, int>, 8> moveCounts;
    for (int i = 0; i < boardSize; ++i)
    {
        //get move
        int dx = get<0>(moves[i]);
        int dy = get<1>(moves[i]);

        int c = 0;
        for (int j = 0; j < boardSize; ++j)
        {
            //calculate next move
            int x2 = x + dx + get<0>(moves[j]);
            int y2 = y + dy + get<1>(moves[j]);

            // check if out side the chess board
            if (x2 < 0 || x2 >= boardSize || y2 < 0 || y2 >= boardSize)
                continue;
            // check if the tile is already filled
            if (data[y2][x2] != 0)
                continue;

            c++;
        }
        // contain all the moves that were made
        moveCounts[i] = make_tuple(c, i);
    }

    // sort the moves count
    sort(moveCounts.begin(), moveCounts.end());

    //return the new move
    array<int, 8> out;
    for (int i = 0; i < boardSize; ++i)
        out[i] = get<1>(moveCounts[i]);
    return out;
}
```

The moveCounts keeps track to all the moves that the program made. First I calculate the next move for the next position, and that calculation is the current x and y position + the move dx and dy + another move to see the next position.

Then I check if the calculated move is out of the board or the tile is already filled. Than all the successful moves are stored in the moveCounts variable and sort it. Now I return the new move that will be used to solve the tour.

```
void Chessboard::solve(string &start, const int boardSize)
{
    //reset all values to 0
    for (int v = 0; v < boardSize; ++v)
        for (int u = 0; u < boardSize; ++u)
            data[v][u] = 0;

    // first start position of the knight
    int x0 = start[0] - 'a';
    int y0 = boardSize - (start[1] - '0');
    data[y0][x0] = 1;
}
```

In these pictures is the solve method to complete the tour. First I set all the data values of each tile to 0 so that there will not be any confusion. After that I set the first position where the algorithm has to start.

```
// a collection of the order for the movement
array<tuple<int, int, array<int, 8>>, 8 * 8> order;
order[0] = make_tuple(x0, y0, sortMoves(x0, y0, boardSize));

const int totalMoves = 8;

int n = 0;
while (n < boardSize * boardSize - 1)
{
    //get start position
    int x = get<0>(order[n]);
    int y = get<1>(order[n]);

    bool ok = false;
    for (int i = 0; i < totalMoves; ++i)
    {
        // set next move
        int dx = moves[get<2>(order[n])[i]].first;
        int dy = moves[get<2>(order[n])[i]].second;

        //check if move is out of bounds
        if (x + dx < 0 || x + dx >= boardSize || y + dy < 0 || y + dy >= boardSize)
            continue;
        // check if the next tile is already filled
        if (data[y + dy][x + dx] != 0)
            continue;

        ++n;
        data[y + dy][x + dx] = n + 1;
        order[n] = make_tuple(x + dx, y + dy, sortMoves(x + dx, y + dy, boardSize));
        ok = true;
        break;
    }

    if (!ok) // Failed. Backtrack.
    {
        data[y][x] = 0;
        --n;
    }
}
}
```

I have here created an *order* array that has a collection of data. It needs a starting point of x and y and it needs a sort move.

First I get the start/current position then I'll check with a Boolean to see if it has to back track to solve.

Now it's going to loop through all the moves. To get the move I used dx and dy for that. After that I check if the current position + the move (x + dx / y + dy) is out of bounds or the tile it was supposed to go is already filled with a number. If that is the case then go to the next move until there is a opening found else the program is going to backtrack to find another solution.

When there is a opening we go to that position and give that the next number as value, now the order is different so once again I collect the elements with their updated values.

6. The knight can only visit each square once

```
//check if move is out of bounds
if (x + dx < 0 || x + dx >= boardSize || y + dy < 0 || y + dy >= boardSize)
    continue;
// check if the next tile is already filled
if (data[y + dy][x + dx] != 0)
    continue;
```

Here I check if the move is out of bounds or that the tile is already been taken. So it can never visit the same square multiple times. Because if it lands on the square that has already been taken it will first look to the next move and if every move has been taken then it will backtrack and try another way.

```
if (!ok) // Failed. Backtrack.
{
    data[y][x] = 0;
    --n;
}
```

Exemplar:

1. The user can decide where the knight starts

```
string inputStart;
bool chooseStart = false;
do
{
    chessBoard.stringPointer = &inputStart;

    printf("\nchoose a letter where you start from 'a' to 'h'\nand a number between 1 and 8: ");

    //put in start position
    cin >> *chessBoard.stringPointer;
    printf("\n");

    // check if the user put in right input
    if (inputStart.size() > 2 || inputStart.size() < 2){
        printf("\nError: The input you have given was to short or to long!\n");
        chooseStart = false;
    }
    else if (string::npos == inputStart.find_first_of("0123456789"))
    {
        printf("\nError: The input you have given did not contain a number\n");
        chooseStart = false;
    }
    else
    {
        chooseStart = true;
    }
} while (!chooseStart);
```

First I tell the user to give a letter between 'a' and 'h' and also set a number between 1 and 8. Just like on a chessboard you have than a1,b1,c1 etc. After that I check if the input you gave is the correct input to start the solution. as long you put in the wrong input it will keep looping through it until you get it right.

```
chessBoard.solve(inputStart, chessBoard.boardSize); //Solve knight's tour
chessBoard.printSolution(chessBoard, chessBoard.boardSize); //print the solution
```

Now that you have put in the correct position of the grid. In the solve method I send a argument with the input string and the board size to solve the tour.

```
void Chessboard::solve(string &start, const int boardSize)
{
    //reset all values to 0
    for (int v = 0; v < boardSize; ++v)
        for (int u = 0; u < boardSize; ++u)
            data[v][u] = 0;

    // first start position of the knight
    int x0 = start[0] - 'a';
    int y0 = boardSize - (start[1] - '0');
    data[y0][x0] = 1;
}
```

The calculation for x0 is been done by getting the first index from the string start and subtract the character 'a'.

example:

- string input = a1;
- $x0 = 'a' - 'a' = 0$

So $x0 = 0$, and that is the index number for *data* to set the x position.

To calculate y0 you get the second index from the string start and subtract '0' and then with that result you subtract that from boardSize.

example:

- boardSize = 8;
- string input = a1;
- $y0 = 8 - (1 - '0') = 7$;

So $y0 = 7$, and that is the index number for *data* to set the y position.

Now that we have the calculated the x0 and y0 we set that position in *data* and give that the value of 1. This will result that the starting position with the input 'a1' will be at the left down corner.

2. Uses a more efficient solution than brute force

```
// a collection of the order for the movement
array<tuple<int, int, array<int, 8>>, 8 * 8> order;
order[0] = make_tuple(x0, y0, sortMoves(x0, y0, boardSize));

const int totalMoves = 8;

int n = 0;
while (n < boardSize * boardSize - 1)
{
    //get start position
    int x = get<0>(order[n]);
    int y = get<1>(order[n]);

    bool ok = false;
    for (int i = 0; i < totalMoves; ++i)
    {
        // set next move
        int dx = moves[get<2>(order[n])[i]].first;
        int dy = moves[get<2>(order[n])[i]].second;

        //check if move is out of bounds
        if (x + dx < 0 || x + dx >= boardSize || y + dy < 0 || y + dy >= boardSize)
            continue;
        // check if the next tile is already filled
        if (data[y + dy][x + dx] != 0)
            continue;

        ++n;
        data[y + dy][x + dx] = n + 1;
        order[n] = make_tuple(x + dx, y + dy, sortMoves(x + dx, y + dy, boardSize));
        ok = true;
        break;
    }

    if (!ok) // Failed. Backtrack.
    {
        data[y][x] = 0;
        --n;
    }
}
```

I used warnsdorff's rule "*We move the knight so that we always proceed to the square from which the knight will have the fewest onward moves*" I do not count moves that revisit any square already visited. On the contrary brute force will calculate all the squares to find a solution. For example, on an 8x8 board there are approximately 4×10^{51} possible move sequences.