Knights tour
Huub Spekman
500639106
Team namespace

Table of contents

1. Introduction

When I got the Knights tour assigment, I didn't waste any time and started programming. First I thought the program out in my head. Than I created all the neccesary header files and finally I added the logic. 4 hours of hard work later I had a basic brute force algorithm on an 8x8 board without any visual representation. It turned out brute forcing an 8x8 board took longer than expected I still decided to call it a day and keep working on it later. Today I finished the assignment, it now is a program where the user can input how big the board has to be and assign the knight a set starting position. In this report I will explain how the program works, how it meets the requirements and how many points I deserve according to the M&T assesment criteria.

2. The program

2.1 main function

```
int main(){
	int width, height;
	Knight* knight = new Knight();
	std::string startingposition;

	while (true){
		std::cout << "Please enter the boardwidth" << std::endl;
		std::cin >> width;
		std::cout << "Please enter the boardheight" << std::endl;
		std::cin >> height;
		if (width < 5 || height < 5){
			std::cout << "The boards width and height must at least be 5" <<
std::endl;
			continue;
		}
		else if (width > 26 || height > 99){
			std::cout << "The maximum width is 26 and the maximum height is
99" << std::endl;
			continue;
		}
		else if ((width * height) % 2 == 1){
			std::cout << "The board size you have chosen has an odd number of
squares, this means the Knight has to start at a black square e.g. a1 b2 c1 f6" <<
std::endl;
		}
		knight->initBoard(width, height);


		std::cout << "Please enter the knights starting position (type \"random\"
for random)" << std::endl;

		std::cin >> startingposition;
		int response = knight->startPath(startingposition.c_str());
		if (response == 0) break;
		else if (response == 1)	std::cout << "the starting position you chose
is not a square on the field the format is (letter)(number) example \"c8\"" <<
std::endl;
		else if (response == 2) std::cout << "The board has an odd size and the
chosen starting position was a white square, therefore with the give parameters there is
no knightstour possible" << std::endl;
	}
	knight->printBoard();
	system("pause");
}
```

As you can see, there isn't a whole lot of exciting things going on here. It simply waits for user input and when this input is given, it checks if the given input is ok and then calls the neccesary initializing functions of the knight. As you can see knight->startPath returns either 0, 1 or 2. with 0 meaning okay, 1 meaning an illigal starting tile has been chosen and 2 meaning that there is no knightstour possible.

2.2 The Tile Class

```
class Tile
{
private:
	char _name[4];
	int _x, _y;
	bool _visited;
public:
	int getX(){ return _x; };
	int getY(){ return _y; };
	bool isVisited(){ return _visited; };
	const char* getName(){ return _name; };
	void setVisited(bool visited) { _visited = visited; };
	Tile(int x, int y, const char* name);
	Tile();
	~Tile();
};
```

The tile class is also not that exciting it is basically just a name and location with a boolean so we can check wheter the tile is visited or not. when initializing the Tile one must specify this name and location. The constructor of this class looks like this

```
Tile::Tile(int x, int y, const char* name)
{
	_x = x;
	_y = y;
	sprintf_s(_name, name);
	_visited = false;
}
```

the reason we can't simply say _name = name is because the memory the string we pass to the constructor when initializing our tiles isntantly gets freed when we are done initializing. so we use sprintf_s to init the name of our tiles to assure it can be read later on.

2.3 The TileList structure

```
struct TileList{
	int boardwidth, boardheight;
	Tile*** grid;
	Tile** list;
	Tile* GetTileByName(const char* name){
		int i = 0;
		while (true){
			if (strcmp(this->list[i]->getName(), name) == 0){
				return list[i];
			}
			i++;
			if (i == boardheight * boardwidth) return nullptr;
		}
	};
};
```

The tilelist structure works as an chessboard with a variable size. The grid is a two-dimensional array of pointers while the list is a one-dimensional array of the same pointers. The reason for this is so that whenever it would be more usefull to use the twodimensional or the onedimensional array we are absolutely sure we are accessing the same tile. Also these are all pointers so I can save the knight objects currentTile and currentPossibleMoves to the same memory so I won't have to edit these arrays seperately. The GetTileByName functions simply searches for a tile in the array with given name. Notice that we use strcmp to compare strings so we actually compare the strings instead of the memory adresses. Later on in the code we know the memory adress of a string should be equal and we check with "==" instead for efficiancy.

2.4 The Knight object

```
class Knight
{
private:
	int stepsrequired;
	int depth;
	struct TileList* tilelist;
	const char** movelist;
	int* nextmove;
	Tile* currentPossibleMoves[8];
	Tile* currentTile;
	bool tiedOptions;

	int findPossibleMoves(Tile* referenceTile);
	int fillPossibleMoves();
	void showDigitsInNumber(int* digits, int number);
	void sortMovesAscendingByScore(int* scorearray, int array);
	void makeMove(int moveNumber);
	void revertMove();
	bool loop();
	void printRow(int row);
public:
	bool initBoard(int width, int height);
	int startPath(const char* startingname);
	void printBoard();
	Knight();
	~Knight();
};
```

As you can see, almost all of the programs logic is in the Knight object. Therefore we'll be going over all the functionality of the Knight object in 3 steps.

1. variables
2. initialization
3. finding the solution
4. printing the solution

2.4.1 The Knight object variables

We'll be going over all the Knight objects variables one by one explaining their purpose.

```
int stepsrequired;
```

This is simply a counter that gets initialized and keeps track of how many steps the knight takes to find the solution, it started out as mainly a debug variable but now is presented to the user at the end.

```
int depth;
```
This keeps track of how many moves have been made so far, when a new move gets made 1 gets added, when we have to move a step back 1 gets subtracted.

```
struct TileList* tilelist;
```
This is the chessboard, all tile related actions call upon this variable for the neccesary information.

```
const char** movelist;
```
String array that contains the names of the moves that are allready made with movelist[0] being the starting tile and movelist[boardwidth * boardhight] the finishing tile.

```
int* nextmove;
```
Integer array that keeps track of what next move to check when backtracking to an older square.

```
Tile* currentPossibleMoves[8];
```
Tilearray that gets filled with possible moves of the Knight on its current square, also gets sorted by warndorfs rule.

```
Tile* currentTile;
```
Pointer to the Tile the Knight is currently on.

```
bool tiedOptions;
```
Boolean that checks if there are moves that score equally according to warndorfs rule. When false there are tied options when true there are not. helps a lot with efficiancy on boards with an odd amount of squares.

2.4.2 The Knight object initialization

The main calls two main initialization functions

```
knight->initBoard(width, height);
knight->startPath(startingposition.c_str());
```

First the initBoard sets up all arrays that are dependant of the boardsize.

```
bool Knight::initBoard(int width, int height){
	// fix memory leak
	if (nextmove) free(nextmove);
	if (movelist) free(movelist);
	if (tilelist->list){
		for (int i = 0; i < tilelist->boardheight * tilelist->boardwidth; i++){
			delete tilelist->list[i];
		}
		free(tilelist->list);
	}
	if (tilelist->grid){
		for (int i = 0; i < tilelist->boardwidth; i++){
			free(tilelist->grid[i]);
		}
		free(tilelist->grid);
	}

	// set the widtg and height of the board
	tilelist->boardwidth = width;
	tilelist->boardheight = height;

	// initialize all the arrays with the now known width and height
	nextmove = (int*)malloc(width * height * sizeof(int));
	int number = 0;
	while (number < width * height - 8){
		nextmove[number] = 0;
		number++;
	}
	nextmove[number] = 0;
	movelist = (const char**)malloc(width * height * sizeof(const char*));


	tilelist->list = (Tile**)malloc(width * height * sizeof(Tile*));

	tilelist->grid = (Tile***) malloc(width * sizeof(Tile**));
	for (int i = 0; i < width; i++){
		tilelist->grid[i] = (Tile**)malloc(height * sizeof(Tile*));
	}

	char letters[26] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' };

	// fill the tilelist with actual tiles
	for (int x = 0; x < width; x++){
		for (int y = 0; y < height; y++){
			char name[4];
			sprintf_s(name, "%c%i", letters[x], y + 1);
			Tile* tempTile = new Tile(x, y, name);
			tilelist->grid[x][y] = tempTile;
			tilelist->list[((x * height) + y)] = tempTile;
		}
	}
	return true;
}
```

Since It is possible to repeat this step when an incorrect starting tile is selected we need to make sure to free all memory of previously allocated variables to ensure there are no memory leaks. After that its pretty straightforward initialization of all arrays and the tilelist.

Next the startPath functions sets the knights starting position and then starts the general loop

```
int Knight::startPath(const char* startingName){
	// Place the Knight at a random position
	if (strcmp(startingName, "random") == 0){
		int field = tilelist->boardheight * tilelist->boardwidth;
		currentTile = tilelist->list[rand() % field];
		while ((tilelist->boardheight * tilelist->boardwidth % 2) == 1){
			if ((currentTile->getX() + currentTile->getY()) % 2 == 1){
				currentTile = tilelist->list[rand() % field];
			}
			else {
				break;
			}
		}
	}
	// Place the night at given name
	else{
		currentTile = tilelist->GetTileByName(startingName);
		// If the given name cannot be found return errorcode 1
		if (currentTile == nullptr){
			return 1;
		}
	}
	// if there are no knightstours possible with given starting square and boardsize
size return errorcode 2
	if ((currentTile->getX() + currentTile->getY()) % 2 == 1 &&
(tilelist->boardheight * tilelist->boardwidth % 2) == 1){
		return 2;
	}

	currentTile->setVisited(true);
	movelist[0] = currentTile->getName();

	// Print the Knights starting location
	printf("The knight starts at ");
	printf(currentTile->getName());
	printf("\n");

	// Keep looping until you filled all the moves then add the last move to the list
	while (true){
		loop();
		if (depth == (tilelist->boardheight * tilelist->boardwidth) - 1) {
			movelist[depth] = currentTile->getName();
			break;
		}
	}

	// print all the moves
	for (int i = 0; i < (tilelist->boardheight * tilelist->boardwidth) - 1; i++){
		printf("Move %i: ", i);
		printf(movelist[i]);
		printf("\n");
	}
	return 0;
}
```

If the user chose to make the knight start at a random location the program starts off by assigining the knight its random position. If the number of squares are odd for instance an 7x9 board and the random starting position is a “white square” the Knightstour is impossible and it chooses a new random starting position. If a square is chosen manually it checks if the chosen square is an actual square on the board. If so it than checks if the knightstour is possible. If everything checks out it starts to loop as long as it needs to find the solution.

2.4.3 The Knight object finding the solution

We find the solution by continiously loopint through this function

```
void Knight::loop() {
	stepsrequired++;
	if (fillPossibleMoves() > nextmove[depth]){
		makeMove(nextmove[depth]);
	}
	else {
		revertMove();
	}
	printf("step %i \n", stepsrequired);
}
```

I’m not going to go over fillPossibleMoves() functionality in great detail. If you would like to know how it works in depth the source is included so you can have a look for yourself. fillPossibleMoves() returns the amount of moves possible from the current tile and fills the currentPossibleMoves array with pointers to these possibilities. When this array is filled it is sorted based on Warnsdorf's rule in the following way.

```
int Knight::fillPossibleMoves(){
	//  left out code to fill the currentPossibleMoves array
-----------------------------------------------------------------------------------------
	// going to sort the array from least possible moves to most possible moves so it
tries the least possible first according to Warnsdorf's rule
	int* movepossibilities = (int*)malloc(possiblemoves * sizeof(int));
	for (int i = 0; i < possiblemoves; i++){
		movepossibilities[i] = findPossibleMoves(currentPossibleMoves[i]);
	}

	// doing exchange sort for the array
	sortMovesAscendingByScore(movepossibilities, possiblemoves);
	// set flag for tied options
	if (possiblemoves > nextmove[depth]){
		if (movepossibilities[nextmove[depth]] !=
movepossibilities[nextmove[depth] + 1] && movelist[nextmove[depth]] !=0){
			tiedOptions = true;
		}
	}
	free(movepossibilities);
	return possiblemoves;

}

void Knight::sortMovesAscendingByScore(int* scorearray, int arraylength){
	for (int i = 0; i < arraylength - 1; i++){
		for (int j = i + 1; j < arraylength; j++){
			if (scorearray[i] > scorearray[j]){

				int temp = scorearray[i];
				scorearray[i] = scorearray[j];
				scorearray[j] = temp;

				Tile* tempTile = currentPossibleMoves[i];
				currentPossibleMoves[i] = currentPossibleMoves[j];
				currentPossibleMoves[j] = tempTile;
			}
		}
	}
}
```

First we create an array that contains the number of moves for each Tile in currentPossibleMoves with the same index. Next up we use the algortihm exchange sort to sort this array from lowest number to highest, and whenever we swap the arrayindexes around we make sure to do the same with the currentPossibleMoves array so it is sorted that index 0 has the least possible moves. When we are done we make sure we free up the memory we allocated for the movepossiblities array, we still do not want any memory leaks. When we are done sorting we check if there are tied options if that is not the case we set the boolean tiedOptions to true.

```
void Knight::loop() {
	stepsrequired++;
	if (fillPossibleMoves() > nextmove[depth]){
		makeMove(nextmove[depth]);
	}
	else {
		revertMove();
	}
	printf("step %i \n", stepsrequired);
}
```

Back to the loop function. As explained before nextmove[depth] holds the value of the move we have to try next. So the first time we are on a squre nextmove[depth] will be 0 and as long as any move is possible we will try the first (according to warnsdorfs rule) move. however if no moves are possible or we are backtracking and we tested all moves we have to revert to our last move

```
void Knight::makeMove(int moveNumber){
	movelist[depth] = currentTile->getName();
	nextmove[depth]++;

	// if there are no tied options, set nextmove to 9 so it won't explore aditional
options reduces 5x5 board starting position b4 from 682 to 54 steps
	if (tiedOptions) nextmove[depth] = 9;
	depth++;
	currentTile = currentPossibleMoves[moveNumber];
	currentTile->setVisited(true);
	printf("The knight moves to ");
	printf(currentTile->getName());
	printf("\n");
}

void Knight::revertMove(){
	nextmove[depth] = 0;
	depth--;
	currentTile->setVisited(false);
	currentTile = tilelist->GetTileByName(movelist[depth]);
	printf("The knight moves back to ");
	printf(currentTile->getName());
	printf("\n");
}
```

If we make the move we first add our current position to the list of moves made, than we add one to the nextmove array so next time we are at this square we won't try the same move again. Next if there are no tied options we set this same value to 9, this way when we backtrack we make sure we skip all other moves since according to warnsdorfs rule all of these other moves won't lead to a solution either. If you would like to see what difference it makes I advice you to comment out that line and make a 5x5 board and start at b4, it is the quickest way you can see the difference. When this is done we add one to the depth, set the new tile to visited and switch our currenttile We also print out what move is made mostly so the user sees we are still doing stuff on bigger boards.

If we revert the move, we first reset nextmove[depth] so next time we make a move on this depth we try the first one again since this should be a different square now. next we lower the depth, set our current tile to no longer be visited and revert our currentTile to the tile before this move was made.

```
	while (true){
		loop();
		if (depth == (tilelist->boardheight * tilelist->boardwidth) - 1) {
			movelist[depth] = currentTile->getName();
			break;
		}
	}
```

when the maximum depth is reached we make sure the last move is added to the array and break out of the loop, after printing all moves in chronological order we return to the main where a call to print the board is made.

2.4.4 The Knight object printing the solution

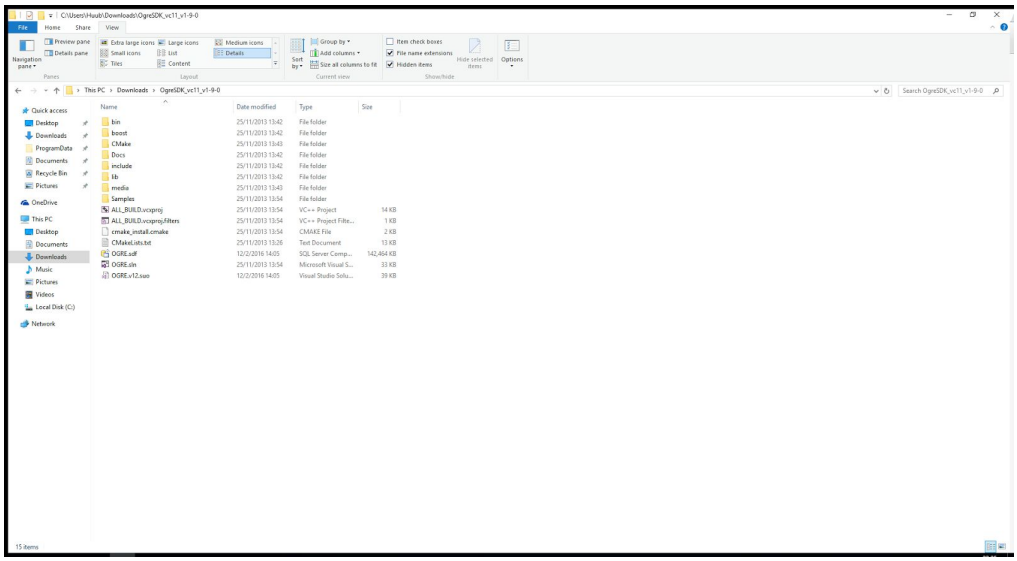
```
void Knight::printBoard(){

	for (int i = tilelist->boardheight; i > 0; i--)	printRow(i);
	if (depth == tilelist->boardheight * tilelist->boardheight){
		printf("The knight started at %s and after %i steps finished his
KnightsTour\n", movelist[0], stepsrequired);
	}
}

// add the digits in a number digits should be a reference
void Knight::showDigitsInNumber(int* digits, int number){
	if (number > 9) showDigitsInNumber(digits, number / 10);
	*digits = *digits + 1; // *digits++ didn't work
}

// print a single row of the board
void Knight::printRow(int row){
	int cols = tilelist->boardwidth;
	int digits = 0;
	showDigitsInNumber(&digits, tilelist->boardheight * tilelist->boardwidth);
	int width = (cols * (3 + digits)) + 1;

	char* dashedline;
	dashedline = (char*)malloc(width + 2); // the reason you need an extra byte is so
you can specify the end of the string
	for (int i = 0; i < width; i++){
		dashedline[i] = '-';
	}
	dashedline[width] = '\n';
	dashedline[width + 1] = '\0';

	if (row == tilelist->boardheight) printf(dashedline);


	printf("|");
	for (int i = 0; i < cols; i++){
		const char* name = tilelist->grid[i][row - 1]->getName();
		int movenumber;
		for (int j = 0; j < depth; j++){
			// reason we check like this is because they should both point to
same memory anyway
			if (movelist[j] == name){
				movenumber = j;
				break;
			}
			movenumber = 99;
		};
		printf(" ");
		int movedigits = 0;
		showDigitsInNumber(&movedigits, movenumber);
		for (movedigits; movedigits < digits; movedigits++){
			printf("0");
		}
		printf("%i |", movenumber);
	}
	printf("\n");
	printf(dashedline);

	delete dashedline; // make sure there are no memory leaks
}
```

The board is printed by printing a dashedline first. After that for each row we print the number followed by another dashed line. I use a reference here which is a requirement for the assignment.

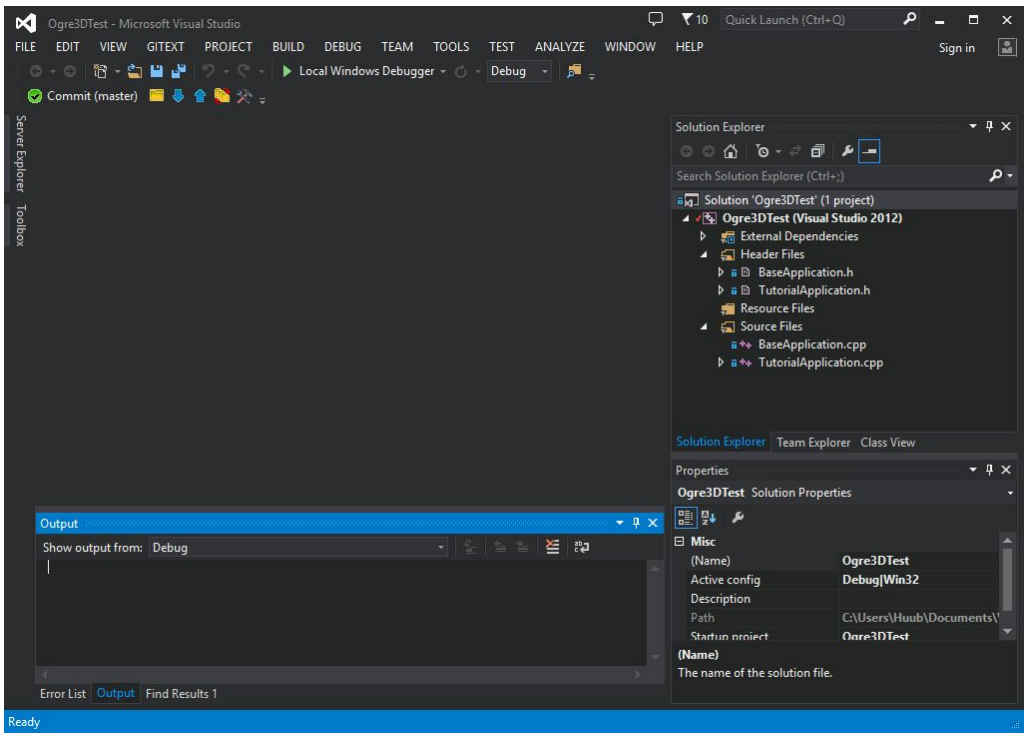
3. MTT scoring

3.1 Development basics 10 points

I have ogre installed, this is a screenshot of the ogre SDK folder



Im working in visual studio, this is a screenshot of our teams ogre test project.



You are reading this report from a git repository so obviously version control is installed. Also My assignment meets all requirements and examplars as shown in chapter 2.

3.2 C++ 9 points

I use C++ language specific concepts throughout the code, I have used multiple instances of memory allocation and explained what I did to ensure no memory leaks are present. The reason I didn't use namespaces or inheritance is because this is a relatively simple assignment and there was no reason to use them.

3.3 Tooling 8 points

I used git for all versions of this assignment, you can go check the log of this repository to find earlier builds of the application. The reason I didn't use different branches to work on the player input for starting position or player input for board size is because I allways saw that as the intended product instead of an added feature. I generally create new branches when working on new functionality that is in a different direction from the original product/branch. Also since we didn't work toghether branches are less of an requirement.

3.4 Coding standards 0 points

We have yet to set coding standards with the team.

3.5 Testing 0 points

I have no documented tests for this assignment.

3.6 porting 0 points

No efforts towards porting for this assignment.

3.7 documentation 5-6 points

I use Inline documentation throughout the code and sometimes explain why I went for that specific option.

3.8 refactoring 6 points

I refactored multiple parts of the code, such as the initialization and most notibly the random seed function, if you look back in my git commits It used to be a really ineficiant function that I decided to get rid off in favor of a much more efficiant solution.

3.9 advanced techniques 0 points

I did 0 advanced techniques for this relatively easy assignment.