University of the Witswatersrand, Johannesburg

Faculty of Engineering and the Built Environment

School of Electrical and Information Engineering

Automated Snakes and Ladders in C++

Software Development I

Dylan Baker – 2093671

2 June 2021

Abstract: This report will cover the implementation of snakes and ladders in C++ but with no user input or output in the console. File handling is used instead. There are two players and the board size chosen was 100 to make testing easy. The design only covers the implementation for two players. Although the project was a success, there were features that were not implemented, such as having multiple games in one output file and allowing up to six players to play.

Section 1: Introduction

This report will cover the design and implementation of a snakes and ladders game in C++. The input from a user will determine the size of the board where the snakes and ladders will be placed. The implementation of the design will explain how the board is made, how the players are moved and who wins. The design will be explained in **Section 2: Design**, followed by the results of the implemented design in **Section 3: Results.** A discussion on what could have been better and how the design choices made were successful can be seen in **Section 4: Discussion**. Finally, the conclusion can be seen in **Section 5: Conclusion**.

Section 2: Design

The first step in the design process was to make the board. Determining the board size, where the snakes and ladders would be placed and how many was done first. The size of the board was chosen as 100 because it allows for easier testing and debugging. The ladders were placed in a position that would allow a player to make significant progress, but they were spaced out. They were spaced to prevent a player from continuously landing on ladders. The snakes were positioned so that they were spaced out to prevent the number of rounds reaching the board size.

The student number 2093671 is converted to binary to determine the number of snakes and ladders. When converted it gives 000111111111001001100111. The number of '0s' gives the number of ladders. The number of '1s' gives the number of snakes. From the binary number there are nine '0s' and fifteen '1s'.

The ratio of snakes to ladders is unbalanced and so equation 1 is used to decrease the number of snakes. Equation 1 is as follows:

$$y = \frac{x}{2} - 1 \text{ .... } (1)$$

x is the input and is the number of current snakes. y is the number of snakes after the equation is solved. $x = 15$ and therefore $y = 7.5$. y was rounded down as making two more ladders than snakes was assumed to make the game more balanced. This will allow the number of rounds played to be reduced.

Players were then designed. Attributes of a player consist of their name, position, if it is their turn and if they have won. This will allow the code to determine who is taking their turn and where to move them.

To move the player a random number generator was made to act as a dice roll. A check is needed to see if the player has landed on a snake, ladder, or a normal position on the board. When the player lands on a snake or ladder an algorithm finds the snake or ladders position and moves the player according to that snake or ladders length.

To manage the game a game handler was made. This initialized the players and the round number. It controlled which players turn it is and moved them accordingly. If the round number reached the board size it will then determine who wins according to who is furthest on the board.

The flow chart for the design can be seen in appendix B.

Section 3: Results

Two tests were done. The first being how many times a player won compared to the other. The second test determined if the players moved correctly and how many snakes and ladders were used by each player overall. It was done for the overall result as for each game it is determined by chance and a computer cannot generate true random numbers.

The first test was used to see if the random number generator worked correctly. The test would fail if one player won eight out of the ten games. The failure could be because of the random number generator or the placement of the snakes and ladders. The first test can be seen below.

<u>Table showing the player that won and lost in a total of ten games</u>

Game	Player that won
1	P-1
2	P-2
3	P-1
4	P-2
5	P-2
6	P-1
7	P-2
8	P-2
9	P-1
10	P-2

P-1 won four times and P-2 won six times. This is almost a 50/50 between the players. From this it was decided that the random number generator was working as there was not an unbalanced number of wins compared to another player. The placement of the snakes and ladders still had to be tested. Test two was used for this as test one only proves that the random number generator works.

The results for test two can be seen below.

<u>Table showing if P-1 moves were done correctly and how many snakes and ladders it landed on</u>

Game	Move Correctly?	How many snakes landed on?	How many ladders landed on?
1	Yes	3	2
2	Yes	1	1
3	Yes	0	2
4	Yes	0	2
5	Yes	0	0

<u>Table showing if P-2 moves were done correctly and how many snakes and ladders it landed on</u>

Game	Move Correctly?	How many snakes landed on?	How many ladders landed on?
1	Yes	0	2
2	Yes	3	2
3	Yes	0	0
4	Yes	0	0
5	Yes	2	2

The results show that the players movement algorithm worked as both players moved correctly. P-1 landed on four snakes and seven ladders over the five games while P-2 landed on five snakes and six ladders. The distribution of snakes and ladders appeared to be fair from this observation. However, if we look at each game it could be unfair, as in the first game P1 landed on three snakes and two ladders whereas P-2 landed on zero snakes and two ladders. This could be unfair towards P-1, but it was noted that in this first game P-1 still won the game. Since snakes and ladders is a game based only on chance the overall amount of how many snakes and ladders were landed on was taken into consideration. The placement of the snakes and ladders was not changed as the distribution is fair.

<u>Section 4: Discussion</u>

Using the board size of 100 and number of players as two it did make testing easier as not as many lines of output had to be checked. Therefore testing and iterating was fast.

The system that was implemented only allows for two players. It was initially done with three players, but it also only allowed for three.

Two was decided to be used as it allowed for easy testing.

The current system can be improved by dynamically allocating the players according to a user input. That would be possible to implement as there is a player class. When the input is read in it can check how many players there are and then dynamically create that number of player objects.

The initial algorithm that placed the snakes and ladders on the board did not work correctly. The algorithm would check if the player has landed on a snake head and then go to the nearest tail. For the ladder it would go to the nearest 'head' of the ladder. This did not work as the next 'head' could be the 'head' of a different ladder or tail for a snake. An example is shown below:

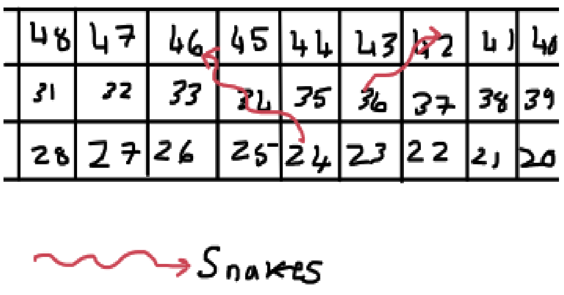


Figure 1: Image showing the issue with old player movement algorithm

In figure 1 there is a snake head at position 46 and a snake head at position 42. If the player landed on 46, they would move backwards to the nearest tail. The nearest tail here is at position 36 and so the player would not move down the correct position which is 24.

This was then fixed by storing the snakes and ladders starting position along with its length. When the player lands on the snake for example, it will then move backwards according to the snake's length.

The time complexities for the algorithm used for moving the player could have been decreased. When the player moved it would then check if they were on a snake head or ladders foot. It used a 'for loop' for each of these. This has a time complexity of O(n) for the worst case as it will iterate n times. A map could have been used to improve the time complexity to O(log n). This can be seen in figure 2 below. [1]

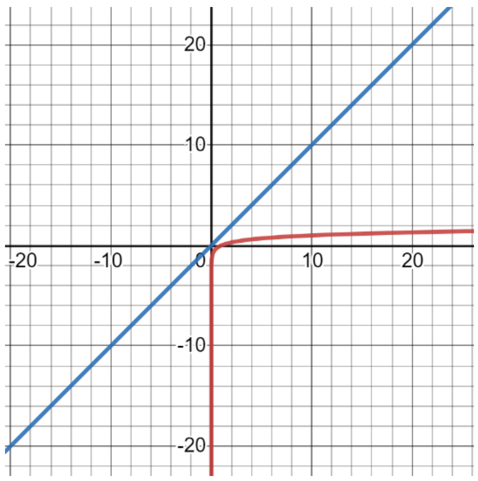


Figure 2: Graph showing *y* = *log n* in red and *y* = *x* in blue

The log graph is faster than the linear graph especially when the size of the input increases. Therefore, using a map will make the time complexity faster. [2]

To make the board design better the snakes and ladders could be allocated dynamically. The possible approach to this would be to generate two random numbers. One for the starting position of the snake or ladder and one for the length. Then, when storing the starting position and length, it needs to check if it is valid to have a snake or ladder in that position and with that length.

The program did not implement multiple games. When trying to implement this it was difficult because of how the input was read in. The algorithm made it difficult to adjust the code as it was not clean. The time management also had an impact but that is discussed in appendix A.

The snakes and ladders were positioned as follows: if the snakes were not spaced out there would be a higher chance of a player landing on a snake hence making the game last longer. The design approach was to make the number of rounds not exceed the board size. Out of all the tests that were done none of them exceeded the board size. The tests showed that the players did land on snakes and so the placement of the snakes worked. However, to not make the game too quick some snakes were placed near the top of ladders. Most of the lengths used for the snakes were shorter than the ladders. Again, to make the game not last too long.

Placement of the ladders was also spaced out but not as much as the snakes. Most of the ladder lengths were also longer to make the progression greater than the progression lost from a snake. This can be seen in the results section. P-1 landed on three snakes and two ladders in the first game. P-2 landed on zero snakes and two ladders but lost.

The assumption that the game would be more balanced by rounding down the number of snakes to 7 after solving equation 1 was correct. The results show that the games did not take long and if the player did land on a snake there was always a chance to win.

The design implemented worked but there is one aspect that can be improved. This was how the code was written. The one header file used named Board.h had an algorithm. The algorithm read in the input from input.txt and then made the board. The implementation was not neat and could have been done better by making use of string manipulation.

Section 5: Conclusion

The design works and the planning done for it allowed for an easy development process with quick testing and iterating. The final design works but can be improved as not all features were added.

The features not implemented were allowing multiple games to be played and the ability to have up to six players in the game. By implementing these the program would have been better.

Section 6: References

[1] J. Coffin, "stack overflow," 12 February 2014. [Online]. Available: https://stackoverflow.com/questions/21740893/what-is-the-time-complexity-of-stdmap. [Accessed 31 May 2021].

[2] "desmos," [Online]. Available: https://www.desmos.com/calculator. [Accessed 2 June 2021].

Appendix A: Project Time Management

Sunday - 9 May	Monday - 10 May	Tuesday - 11 May	Wednesday - 12 May	Thursday - 13 May	Friday - 14 May	Saturday - 15 May	Total Hours (h)
12:00 - 13:00 Reading brief and looking and possible solutions			16:00 - 18:00 Read part of the brief again and made a simple board	00:00 - 02:00 - Adding snakes and ladders algorithm 21:30 - 22:30 Adding to snakes and ladders algorithm	00:30 - 02:00 Changed adding snakes and ladders algorithm/Added dice and tested player handler	12:00 - 16:00 Working on random number generator	10.5

Sunday - 16 May	Monday - 17 May	Tuesday - 18 May	Wednesday - 19 May	Thursday - 20 May	Friday - 21 May	Saturday - 22 May	Total Hours (h)
	10:30 - 11:00 Fixed random number generator issue. Added player movement function	19:00 - 22:00 Simplified code and tested outputs.		22:00 - 01:00 Debugging and testing			6.5

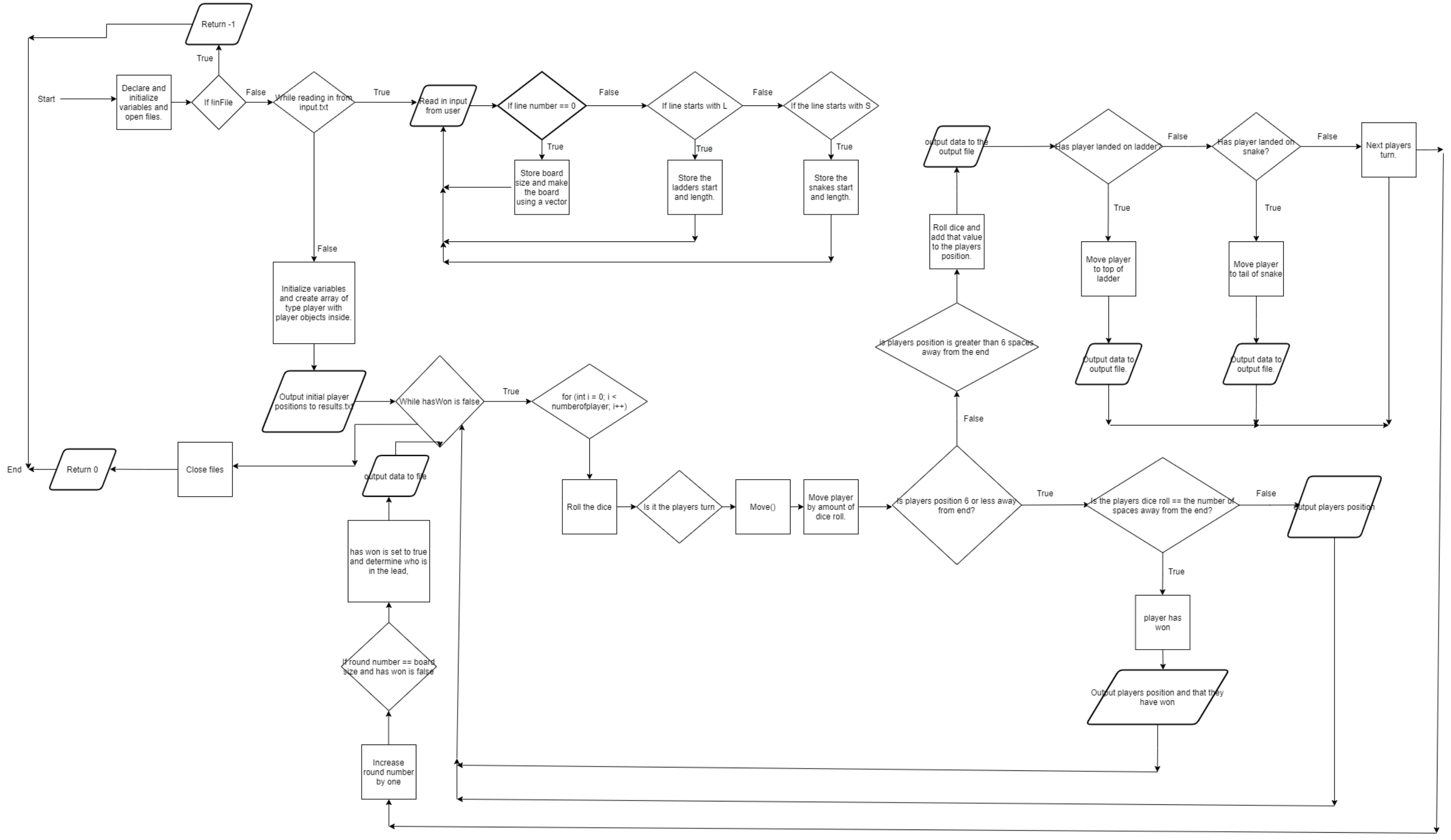
Sunday - 23 May	Monday - 24 May	Tuesday - 25 May	Wednesday - 26 May	Thursday - 27 May	Friday - 28 May	Saturday - 29 May	Total Hours (h)
	18:00 - 21:00 Added player class to simplify code	23:00 - 23:30 Adding functions into header files so that there is minimal code in main.cpp	15:00 - 16:00 Bug where it would show who won but then moves were still being made. Debugged that.				4.5

Sunday - 30 May	Monday - 31 May	Tuesday - 1 June	Wednesday - 2 June	Thursday - 3 June	Friday - 4 June	Saturday - 5 June	Total Hours (h)
17:00 - 19:00 Tested different board sizes and debugging.	12:30 - 14:00 Report Writing	09:00 - 11:30 Report Writing 14:00 - 20:00 More report writing					12

The expected time taken to complete the project was estimated to be 35 hours. 25 hours was estimated to be used for the design and implementation and then 10 hours for report writing. This was nearly accurate. The total time taken to finish the design and implementation was approximately 23.5 hours which was 1.5 hours off the estimation. The time taken to do the report writing was approximately 10 hours which is the time that I had dedicated to the report writing. The total time taken is 33.5 hours. This is 1.5 hours off the estimated time taken.

There was one issue with my time management though. The implementation of the design was executed quicker than I thought so I thought I had enough time to implement the multiple games feature. The way I implemented the reading in of the user input was messy and not the best in terms of coding practices. I struggled to implement this feature and so decided to leave it out so there is at least one game that can be played correctly with no bugs.

Appendix B: Flow Chart


Start
Declare and initialize variables and open files.
If !inFile
True
Return -1
False
While reading in from input.txt
True
Read in input from user
If line number == 0
True
Store board size and make the board using a vector
False
If line starts with L
True
Store the ladders start and length.
False
If the line starts with S
True
Store the snakes start and length.
False
Initialize variables and create array of type player with player objects inside.
Output initial player positions to results.txt
While hasWon is false
True
for (int i = 0; i < numberofplayer; i++)
Roll the dice
Is it the players turn
Move()
Move player by amount of dice roll.
Is players position 6 or less away from end?
False
is players position is greater than 6 spaces away from the end
Roll dice and add that value to the players position.
output data to the output file
Has player landed on ladder?
True
Move player to top of ladder
Output data to output file.
False
Has player landed on snake?
True
Move player to tail of snake
Output data to output file.
False
Next players turn.
True
Is the players dice roll == the number of spaces away from the end?
False
Output players position
True
player has won
Output players position and that they have won
Increase round number by one
If round number == board size and has won is false
has won is set to true and determine who is in the lead,
output data to file
Close files
Return 0
End