CSC 362 Programming Assignment #3
Due date: Friday, October 18

This assignment will test your ability to use pointers to access a string (array of chars). The program itself will implement a variation of the game *Chutes and Ladders*. In the game, players take turns rolling a 6-sided die, moving their piece on the game board. If they land on a chute, they slide "down" to the end of the chute (thus moving backward) and if they land on a ladder, they climb up the ladder (thus moving forward). Your implementation will have a few differences as follows.

1. There will only be two players in this came to simplify its implementation.
2. After a player lands on a chute or ladder, the chute/ladder is removed so that it will not be used again.
3. Some squares are denoted "havens". Some squares cause the player to move backward to the nearest previous haven or forward to the next haven. Once a haven has been landed on, that haven is removed from the board. For instance, player 1 lands on a square to move back to the previous haven and then player 2 lands on the same square to move backward to the previous haven. Player 2 moves further back because the nearest haven behind player 2 was removed when player 1 landed on it. If a player is to move backward to a haven and there are none left on the board, the player moves back to square 0. If a player is to move forward to the next haven and there are none left, the player does not move.
4. If a player lands on a square that the other player is currently on, the player who just moved is moved back 1 square. Note that in moving back 1 square, if the player lands on a chute, ladder or forward/backward move, that the additional move is ignored. On the other hand, if a player moves forward/backward because of a chute or ladder or haven and as a result of that move, lands on the other player, this player moves back 1 additional square. Only test for collision after the entire move takes place.
5. NOTE: the collision rule is not in effect when both players are on square 0 either at the beginning of the game or if both wind up moving back to 0 at some point.

The game board is set up as an array of characters. The array should be declared with the characters already in place by using notation like the following. There are 99 characters (the \0 is the 100th character, which is not shown below). Do not copy and paste from this pdf as the extra blanks do not appear. A version of this is posted with this assignment for you to copy.

```
char board[100]="  mHk nH l B He Flq p H  hByHlho H B  jr HFB ir j H
F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB ";
```

The characters of the board have the following meaning:

- ' ': a normal square, a player who lands here does not move again in this turn (unless the other player is on the same square)
- 'B': move backward to the nearest preceding haven, stop at 0 if no more havens exist
- 'F': move forward to the next haven, stay here if no more havens exist
- 'H': a haven which a player might move to when landing on a 'B' or 'F'; once moved to because of a 'B' or 'F', change the 'H' to a '*' (do not change the 'H' if the player lands on this square normally, that is, without reaching here because of 'B' or 'F')
- 'a' – 'm': a chute, move backward (see below), change this to '-' after the play moves
- 'o' – 'z': a ladder, move forward (see below), change this to '-' after the play moves
- 'n' – is not used

To determine the distance to move on a chute or ladder (‘a’ – ‘m’, ‘o’ – ‘z’), take the lower case letter and subtract ‘n’ (ASCII 110) from it. For instance, ‘m’ is stored in ASCII as 109. 109-110 = -1 so ‘m’ means “move back 1 square”. Landing on ‘p’ would move you forward 2 squares (112 – 110). ‘n’ does not appear on the board because it would result in no movement.

Now for the tricky part of this program. You *cannot use array accessing* at all to access into the board array. You *must* use pointers to access into the board array to determine what a player lands on and to change the board array when you are to remove a chute/ladder (‘a’ – ‘m’, ‘o’ – ‘z’) or haven (‘H’). Use two char pointers, say *p1 and *p2, to store the location of where the two players currently reside on the board. They will start at board[0], which you can accomplish by using either `p1 = board;` or `p1 = &board[0];` Use pointer arithmetic to move the player. For instance if `move` is the amount to move forward, then reset the player’s location using `p1=p1+move`. To test the square where the player has landed, use *p1/*p2 as in `if(*p1=='B')…` Note that to avoid a run-time error, make sure that your pointer (p1 and p2) is within bounds. A pointer is in bounds `if (p1 >= board && p1 < board + SIZE)` where SIZE is 100 (but in this case, make SIZE a constant as you might change the size of the board at some point). If you attempt to dereference an out of bounds pointer, you *may* get a run-time error.

You are required to break this program into functions. The following are the functions to write. You may have other functions if you find them needed or useful.

- main – Declare and initialize the players (the char pointers, say p1, p2), the board and a FILE pointer. Seed the random number generator. Open the FILE as an output file. Loop until a player wins:
  - In the loop, call the function move (see below) to move p1 and then to move p2. Move will output the result of each player’s move. Call the output function to output the current board to disk file (see below).
  - After the loop (after the game ends), determine and output who won the game (the winner will not necessarily be the player who reached board+100 first as both players may reach the end of the board during the same turn and a player could go farther than board+100 (for instance, the player might be at board+98 and roll a 4), players do not need to stop right at the end of the board.
- output – Output the current game board to disk file on a single line. This function receives the board, both player pointers, and the file pointer. The function will use a pointer to iterate through each character of the board and output either ‘1’, ‘2’ or the character at that board position depending on whether the local pointer is p1 (then output ‘1’), p2 (output ‘2’) or neither (output the value at the current location); use a loop to iterate through the string character-by-character and putc to output the current board character, ‘1’ or ‘2’ to the file. **Do not** output the board using fprintf(fp, “%s”, board);). To iterate down the array, use a loop which exits once the board pointer reaches \0.
- move – This function receives both player pointers, the board, the player number (1 or 2) and the size of the board. Output the player number. Randomly generate a move from 1-6. Move the player by that amount and determine if the player has landed on a chute/ladder or ‘B’/’F’, if so, call the appropriate function (see below); if not or if the player has moved because of it, see if the player has landed on the other player (p1==p2) and if so, move this player back 1 additional square indicating a collision occurred. Output the amount the player moved and where the player is now (the current location is p1 – board or p2 - board). NOTE: All output from this function

goes to the console window using printf. main will call move twice, once for each player, using notation like this:

```
      p1 = move(p1, p2, 1, board, SIZE); // player 1's turn
      p2 = move(p2, p1, 2, board, SIZE); // player 2's turn
```

SIZE is a constant, passed as a parameter. It is needed before printing *p1 or *p2 because you do not want to print *p1 or *p2 if p1/p2 >= board+SIZE (may cause an error). NOTE: as this function returns the new location of p1/p2, it is a char * (it returns a char pointer).

- findHaven – This function is called if the player lands on a 'B' or 'F'. Sarch forward/backward for the next 'H'. If you reach board moving backward, stop searching and move the player here and if you reach board+SIZE moving forward, stop searching and do not move the player; reset the square landed on from 'H' to '*'. This function returns the location of the player once moved (or the current location if not moved). This function should not output anything, that should be handled in the move function.
- chuteLadder – This function returns the new location of the player moving from landing on a chute or ladder. The distance moved is `p1+(int)(*p1-'n');` (or p2 for player 2). Change the original square (the chute/ladder square) to '-' and return the new computed location (this is char pointer). The (int) cast in the above expression ensures that (*p-'n') is cast as an int rather than a char. For instance, if p1 is at board+2 then p1 is on an 'm'. This operation computes p1 + (109-110) which is p1 – 1. This function should be called as `p1=chuteLadder(p1, board);` (or p2) This function is called from move, not main. This function should not output anything (again, all output is handled in move).

Although this program requires multiple functions, it can be written in one file with or without a separate header file. However, make sure you comment your prototypes as if they were placed in a header file to get into a good habit.

The following is a partial output from my version of the program of the console window output. We see players landing on a chutes and 'B's and a couple of collisions. Notice that a couple of 'B's cause players to move to square 0 because the previous 'H' had already been removed from an earlier 'B'.

```
Player 1 rolled 1 and moved to square 1
Player 2 rolled 3 and moved to square 3
Player 1 rolled 3 and moved to square 4 which is a chute and is moving
back to square 1
Player 2 rolled 5 and moved to square 8
Player 1 rolled 3 and moved to square 4
Player 2 rolled 3 and moved to square 11 which is a 'B' so is moving back
and lands at 7
Player 1 rolled 5 and moved to square 9 which is a chute and is moving
back to square 7 -- Collision!  1 is moving back 1 square to 6
Player 2 rolled 4 and moved to square 11 which is a 'B' so is moving back
and lands at 3
Player 1 rolled 1 and moved to square 7
Player 2 rolled 4 and moved to square 7 -- Collision!  2 is moving back 1
square to 6
Player 1 rolled 3 and moved to square 10
Player 2 rolled 5 and moved to square 11 which is a 'B' so is moving back
and with no more havens, lands at 0
```

The following is an excerpt of the text file created for the moves shown above. Notice how chutes and ladders are replaced with ‘-’ after having been landed on, and havens replaced with ‘*’. We see player 2 being moved back to 0 twice because of landing on B’s with no ‘H’ available.

```
 1m2k nH l B He Flq p H  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
 1mH- nH2l B He Flq p H  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  mH1 n2 l B He Flq p H  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m2- 1* - B He Flq p H  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m*- 21 - B He Flq p H  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
2 m*- n* -1B He Flq p H  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m*- 2* - B He Flq p 1  hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m*- -* 2 B He Flq p * 1hByHlho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
2 m*- -* - B He Flq p *  hB1H-ho H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m*-2-* - B He Flq p *  hByH-h-1H B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m*- -* 2 B He Flq p *  hByH-h- 1 B  jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
  m*- -* - B2He Flq p *  hByH-h- * B1 jr HFB ir j H  F ku gd  H pjB mH x  BF i H  m oB HlHFBhoH BB
```

After making sure your program is running correctly, run it as many times as necessary until you have a version in which players land on chutes, ladders, Bs/Fs, at least one player moves back to square 0 because of a ‘B’ or does not move because of an ‘F’, and at least one collision. Hand in your program and for the selected run, the entire console window output and output file created from that run.

NOTE: as the console window output and the output file may be lengthy, to save paper you can email the two sets of output to the instructor before you submit the program, but hand in the program itself.