//int navigateSegment(int r, int c, char dir, int maxSteps)

{

/\*char lowerCaseLetter = tolower(dir);

int possibleSteps = 0;

if (r > getRows() || c > getCols()) // if the starting position is outside of the grid size, return -1

return -1;

else

return 0;

//else if (isWall(r, c)) // if the starting position is at a wall, return -1

//return -1;

/\*else if (lowerCaseLetter == 'n') // if the direction is North

{

int RowNorthofPoint = (r - 1); // subract one from the row to test the spot exactly one unit above north

while (validStep(RowNorthofPoint, c)) // while a step is valid (no wall and within boundaries: see validStepNS below)

{

possibleSteps++; // add one to the possible steps that can be taken

RowNorthofPoint--;

}

if (possibleSteps >= maxSteps)

return maxSteps;

else

return possibleSteps;

}

else if (lowerCaseLetter == 's')

{

int RowSouthofPoint = (r + 1);

while (validStep(RowSouthofPoint, c)) // while a step is valid (no wall and within boundaries: see validStepNS below)

{

possibleSteps++; // add one to the possible steps that can be taken

RowSouthofPoint--;

}

if (possibleSteps >= maxSteps)

return maxSteps;

else

return possibleSteps;

}

else if (lowerCaseLetter == 'e')

{

int ColumnEastofPoint = (c + 1);

while (validStep(r, ColumnEastofPoint)) // while a step is valid (no wall and within boundaries: see validStepEW below)

{

possibleSteps++; // add one to the possible steps that can be taken

ColumnEastofPoint--;

}

if (possibleSteps >= maxSteps)

return maxSteps;

else

return possibleSteps;

}

else if (lowerCaseLetter == 'w')

{

int ColumnWestofPoint = (c - 1);

while (validStep(r, ColumnWestofPoint)) // while a step is valid (no wall and within boundaries: see validStepEW below)

{

possibleSteps++; // add one to the possible steps that can be taken

ColumnWestofPoint--;

}

if (possibleSteps >= maxSteps)

return maxSteps;

else

return possibleSteps;

}

else return -1;\*/

}

/\*int nRows = 0;

int nColumns = 0;

int startr = 0;

int startc = 0;

int endr = 0;

int endc = 0;

char Direction;

int NumberofSteps = 0;

// GET DATA FOR TESTING

cout << "Number of rows: ";

cin >> nRows;

cout << "Number of columns: ";

cin >> nColumns;

cout << "Start row: ";

cin >> startr;

cout << "Start column: ";

cin >> startc;

cout << "End row: ";

cin >> endr;

cout << "End column: ";

cin >> endc;

setSize(nRows, nColumns);

draw(startr, startc, endr, endc);

int nWalls = 0;

cout << "How many walls would you like to set ";

cin >> nWalls;

for (int k = nWalls; k > 0; k--)

{

int wallr = 0;

int wallc = 0;

cout << "Wall row: ";

cin >> wallr;

cout << "Wall column: ";

cin >> wallc;

setWall(wallr, wallc);

}

draw(startr, startc, endr, endc);

int checkr;

int checkc;

cout << "Check if steps can be taken from this point: " << endl;

cout << "Row: ";

cin >> checkr;

cout << "Column: ";

cin >> checkc;

cout << "Direction: ";

cin >> Direction;

cout << "Number of Steps: ";

cin >> NumberofSteps;

cout << navigateSegment(checkr, checkc, Direction, NumberofSteps);\*/

/\*if (clearPosition(checkr, checkc))

cout << "This is a valid location to take a step" << endl;

if (!clearPosition(checkr, checkc))

cout << "This is not a valid location to take a step" << endl;\*/

// cout << "Maximum number of steps in this direction: " << navigateSegment(startr, startc, Direction, NumberofSteps); \*/

//if (dirLower == 'n' || dirLower == 's')

//{

//if (dirLower == 'n')

// nextPosition = r + 1;

//if (dirLower == 's')

// nextPosition = r - 1;

//cout << nextPosition;

//while (clearPosition(nextPosition, c))

//{

// nSteps++;

// nextPosition++;

//}

//}

//if (dirLower == 'e' || dirLower == 'w')

//{

//if (dirLower == 'e')

// nextPosition = c + 1;

//if (dirLower == 'w')

// nextPosition = c - 1;

//while (clearPosition(r, nextPosition))

//{

// nSteps++;

// nextPosition++;

//}

//}

//if (nSteps >= maxSteps)

// return maxSteps;

//else

// return nSteps;

#include "grid.h"

#include <string>

#include <cctype>

#include <cassert>

#include <iostream>

using namespace std;

bool isRouteWellFormed(string route);

int navigateSegment(int r, int c, char dir, int maxSteps);

bool clearPosition(int row, int column);

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps);

int main()

{

string route;

getline(cin, route);

if (!isRouteWellFormed(route))

{

cout << "Route is not well formed!" << endl;

}

else

cout << "Route is well formed!" << endl;

setSize(6, 6);

setWall(3, 1);

setWall(2, 2);

setWall(3, 3);

setWall(5, 5);

draw(2, 1, 3, 4);

int stepsyo = 1;

switch (navigateRoute(2, 1, 3, 4, route, stepsyo))

{

case 0:

cout << "valid route. valid start/end points. robot ends at (er, ec). Congrats!" << endl;

break;

case 1:

cout << "valid route. valid start/end points. robot does not end at (er, ec). sorry :(" << endl;

break;

case 2:

cout << "not a valid route or not valid start/end points. #youmessedupyo" << endl;

break;

case 3:

cout << "valid route. valid start/end points. Your path hit a wall or fell of the grid. sorry :(" << endl;

break;

default:

cout << "you did something very long because you aren't even getting a correct digit returned....wooowwwww..." << endl;

}

cout << "nsteps: " << stepsyo << endl;

}

bool isRouteWellFormed(string route)

{

for (int position = 0; position != route.size(); position++)

{

if (isdigit(route[0]))

return false;

if (isalpha(route[position]))

{

char letter = tolower(route[position]); // establish this character to ease following if statement

if (letter == 'n' || letter == 's' || letter == 'e' || letter == 'w') // if the letter is n, s, e, or w, continue (position++)

continue;

else

return false;

}

else if (isdigit(route[position]))

{

int digit1 = route[position]; // simplify calculations by adding integer

int digit2 = position + 1; // digit 2 indicates the character following the digit

if (digit1 < 0) // if the first digit is negative, the string is not correct, therefore return false

return false;

else if (isdigit(route[digit2])) // if the digit is positive, check to see if the next character is a digit

{

if (digit2 < 0) // make sure the next digit is nonnegative

return false;

int digit3 = position + 2; // digit 3 indicates the character two positions away from the first digit

if (isdigit(route[digit3])) // if the 3rd character in this sequence is a digit, then the string is not in the proper form

{

return false;

}

else

continue;

}

else

continue;

}

else if (position != (route.size() - 1)) // if none of the above if statements apply, and we have not reached the end of the string

return false; // then there is a character that is not a letter or a digit in the string, making it invalid

}

return true;

}

int navigateSegment(int r, int c, char dir, int maxSteps)

{

int nSteps = -1; // start at -1 steps because the first step in the while loop is just the starting position, not motion to a new step

if (r > getRows() || c > getCols()) // if the point given is outside of the grid, point not valid

return -1;

else if (isWall(r, c)) // if the point given is a wall, point not valid

return -1;

else

{

while (getRows() >= r && getCols() >= c && r > 0 && c > 0) // while the point is within the grid

{

if (isWall(r,c)) // if the position is a wall, end the path

break;

else

nSteps++; // if the position is not a wall, count it as a step

if (tolower(dir) == 'n') // if dir is north, check the point one row above it

r--;

if (tolower(dir) == 's') // if dir is south, check the point one row below it

r++;

if (tolower(dir) == 'e') // if dir is east, check the point one column to the right

c++;

if (tolower(dir) == 'w') // if dir is east, check the point one column to the left

c--;

}

if (nSteps >= maxSteps) // if the possible steps are greater than or equal to the max steps, return the maxsteps

return maxSteps;

else return nSteps; // if the possible steps are less than max steps, return the possible steps

}

}

bool clearPosition(int row, int column)

{

if (row > getRows() || column > getCols())

return false;

else if (row <= 0 || column <= 0) // make sure row and column are within the grid

return false;

else if (isWall(row, column)) // if it's inside the boundaries, check to see if there is a wall in new location

return false; // if there's a wall in the position you want to step to, return false (step not valid)

else return true; // if the new step passes these tests, return true because the step is valid

}

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps)

{

if (!isRouteWellFormed || !clearPosition(sr, sc) || !clearPosition(er, ec))

return 2;

int k = 0;

int mr = sr; // moving row starts at the row of the start position

int mc = sc; // moving column starts at the column of the start position

int stepsMoved = 0;

int possible\_steps = 0;

while (k != route.size())

{

if (!clearPosition(mr, mc))

{

nsteps = stepsMoved;

return 3;

}

if (isalpha(route[k]))

{

if (isalpha(route[k + 1]) || k == (route.size() - 1)) // if the next character is a letter or the last character of the route

{

possible\_steps = navigateSegment(mr, mc, route[k], 1);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

if (!clearPosition(mr, mc))

{

nsteps = stepsMoved;

return 3;

}

k += 2;

}

else if (isdigit(route[k + 1])) // if the next character is a number

{

int firstDigit = route[k + 1] - '0'; // save the first digit after the letter as

if (isdigit(route[k + 2])) // if the number contains two digit characters

{

int secondDigit = route[k + 2] - '0'; // establish the character as an integer

int twoDigitNumber = (10 \* firstDigit) + secondDigit; // create the two digit number

possible\_steps = navigateSegment(mr, mc, route[k], twoDigitNumber);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

if (!clearPosition(mr, mc))

{

nsteps = stepsMoved;

return 3;

}

k += 3; // have k skip over the letter, and two digits of the number if mr is a clear position

}

else

{ // if first digit is the only digit and k+2 is a letter

possible\_steps = navigateSegment(mr, mc, route[k], firstDigit);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

if (!clearPosition(mr, mc))

{

nsteps = stepsMoved;

return 3;

}

k += 2;

}

}

}

}

cout << "End Row: " << mr << endl;

cout << "End Column: " << mc << endl;

if (mr == er && mc == ec)

{

nsteps = stepsMoved;

return 0;

}

else

{

nsteps = stepsMoved;

return 1;

}

}

#include "grid.h"

#include <string>

#include <cctype>

#include <cassert>

#include <iostream>

using namespace std;

bool isRouteWellFormed(string route);

int navigateSegment(int r, int c, char dir, int maxSteps);

bool clearPosition(int row, int column);

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps);

int main()

{

string route;

getline(cin, route);

if (!isRouteWellFormed(route))

{

cout << "Route is not well formed!" << endl;

}

else

cout << "Route is well formed!" << endl;

setSize(15, 20);

setWall(4, 5);

setWall(3, 8);

setWall(3, 3);

setWall(10, 8);

draw(12, 1, 13, 14);

int stepsyo = 1;

switch (navigateRoute(2, 1, 3, 4, route, stepsyo))

{

case 0:

cout << "valid route. valid start/end points. robot ends at (er, ec). Congrats!" << endl;

break;

case 1:

cout << "valid route. valid start/end points. robot does not end at (er, ec). sorry :(" << endl;

break;

case 2:

cout << "not a valid route or not valid start/end points. #youmessedupyo" << endl;

break;

case 3:

cout << "valid route. valid start/end points. Your path hit a wall or fell of the grid. sorry :(" << endl;

break;

default:

cout << "you did something very long because you aren't even getting a correct digit returned....wooowwwww..." << endl;

}

cout << "nsteps: " << stepsyo << endl;

}

bool isRouteWellFormed(string route)

{

for (int position = 0; position != route.size(); position++)

{

if (isdigit(route[0]))

return false;

if (isalpha(route[position]))

{

char letter = tolower(route[position]); // establish this character to ease following if statement

if (letter == 'n' || letter == 's' || letter == 'e' || letter == 'w') // if the letter is n, s, e, or w, continue (position++)

continue;

else

return false;

}

else if (isdigit(route[position]))

{

int digit1 = route[position]; // simplify calculations by adding integer

int digit2 = position + 1; // digit 2 indicates the character following the digit

if (digit1 < 0) // if the first digit is negative, the string is not correct, therefore return false

return false;

else if (isdigit(route[digit2])) // if the digit is positive, check to see if the next character is a digit

{

if (digit2 < 0) // make sure the next digit is nonnegative

return false;

int digit3 = position + 2; // digit 3 indicates the character two positions away from the first digit

if (isdigit(route[digit3])) // if the 3rd character in this sequence is a digit, then the string is not in the proper form

{

return false;

}

else

continue;

}

else

continue;

}

else if (position != (route.size() - 1)) // if none of the above if statements apply, and we have not reached the end of the string

return false; // then there is a character that is not a letter or a digit in the string, making it invalid

}

return true;

}

int navigateSegment(int r, int c, char dir, int maxSteps)

{

int nSteps = -1; // start at -1 steps because the first step in the while loop is just the starting position, not motion to a new step

if (r > getRows() || c > getCols()) // if the point given is outside of the grid, point not valid

return -1;

else if (isWall(r, c)) // if the point given is a wall, point not valid

return -1;

else

{

while (getRows() >= r && getCols() >= c && r > 0 && c > 0) // while the point is within the grid

{

if (isWall(r,c)) // if the position is a wall, end the path

break;

else

nSteps++; // if the position is not a wall, count it as a step

if (tolower(dir) == 'n') // if dir is north, check the point one row above it

r--;

if (tolower(dir) == 's') // if dir is south, check the point one row below it

r++;

if (tolower(dir) == 'e') // if dir is east, check the point one column to the right

c++;

if (tolower(dir) == 'w') // if dir is east, check the point one column to the left

c--;

}

if (nSteps >= maxSteps) // if the possible steps are greater than or equal to the max steps, return the maxsteps

return maxSteps;

else return nSteps; // if the possible steps are less than max steps, return the possible steps

}

}

bool clearPosition(int row, int column)

{

if (row > getRows() || column > getCols())

return false;

else if (row <= 0 || column <= 0) // make sure row and column are within the grid

return false;

else if (isWall(row, column)) // if it's inside the boundaries, check to see if there is a wall in new location

return false; // if there's a wall in the position you want to step to, return false (step not valid)

else return true; // if the new step passes these tests, return true because the step is valid

}

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps)

{

if (!isRouteWellFormed || !clearPosition(sr, sc) || !clearPosition(er, ec))

return 2;

int k = 0;

int mr = sr; // moving row starts at the row of the start position

int mc = sc; // moving column starts at the column of the start position

int stepsMoved = 0;

int possible\_steps = 0;

if (route.size() == 1 && navigateSegment(mr, mc, route[0], 1) == 0)

{

nsteps = 0;

return 3;

}

if (route.size() == 1 && navigateSegment(mr,mc,route[0],1) == 1)

{

nsteps = 1;

return 1;

}

while (k != route.size())

{

if (isalpha(route[k]))

{

if (k == route.size() - 1) // the last letter will not have a digit nor a letter after it, so this must be addressed

{

possible\_steps = navigateSegment(mr, mc, route[k], 1);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (possible\_steps < 1)

{

nsteps = stepsMoved;

return 3;

}

else

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

break;

}

if (!isdigit(route[k + 1])) // if the next character is a letter or the last character of the route

{

possible\_steps = navigateSegment(mr, mc, route[k], 1);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (possible\_steps < 1)

{

nsteps = stepsMoved;

return 3;

}

if ((k + 1) >= route.size())

break;

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

k++;

}

else // if the next character is a number

{

int firstDigit = route[k + 1] - '0'; // save the first digit after the letter as

if (isdigit(route[k + 2])) // if the number contains two digit characters

{

int secondDigit = route[k + 2] - '0'; // establish the character as an integer

int twoDigitNumber = (10 \* firstDigit) + secondDigit; // create the two digit number

possible\_steps = navigateSegment(mr, mc, route[k], twoDigitNumber);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (possible\_steps < twoDigitNumber)

{

nsteps = stepsMoved;

return 3;

}

if ((k + 3) >= route.size())

break;

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

k += 3; // have k skip over the letter, and two digits of the number if mr is a clear position

}

else

{ // if first digit is the only digit and k+2 is a letter

possible\_steps = navigateSegment(mr, mc, route[k], firstDigit);

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (possible\_steps < firstDigit)

{

nsteps = stepsMoved;

return 3;

}

if ((k + 2) >= route.size())

break;

if (tolower(route[k]) == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (tolower(route[k]) == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (tolower(route[k]) == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (tolower(route[k]) == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

k += 2;

}

}

}

}

cout << "End Row: " << mr << endl;

cout << "End Column: " << mc << endl;

if (mr == er && mc == ec)

{

nsteps = stepsMoved;

return 0;

}

else

{

nsteps = stepsMoved;

return 1;

}

}

if (route.size() == 1 && navigateSegment(mr, mc, route[0], 1) == 0)

{

nsteps = 0;

return 3;

}

if (route.size() == 1 && navigateSegment(mr,mc,route[0],1) == 1)

{

nsteps = 1;

return 1;

}

FINAL CODE!!!!! YAYYYYY!!!!

#include "grid.h"

#include <string>

#include <cctype>

#include <cassert>

#include <iostream>

using namespace std;

bool isRouteWellFormed(string route);

int navigateSegment(int r, int c, char dir, int maxSteps);

bool clearPosition(int row, int column);

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps);

int main()

{

string route;

getline(cin, route);

if (!isRouteWellFormed(route))

{

cout << "Route is not well formed!" << endl;

}

else

cout << "Route is well formed!" << endl;

setSize(15, 20);

setWall(4, 5);

setWall(3, 8);

setWall(3, 3);

setWall(10, 8);

draw(12, 1, 13, 14);

int stepsyo = 1;

switch (navigateRoute(12, 1, 13, 14, route, stepsyo))

{

case 0:

cout << "valid route. valid start/end points. robot ends at (er, ec). Congrats!" << endl;

break;

case 1:

cout << "valid route. valid start/end points. robot does not end at (er, ec). sorry :(" << endl;

break;

case 2:

cout << "not a valid route or not valid start/end points. #youmessedupyo" << endl;

break;

case 3:

cout << "valid route. valid start/end points. Your path hit a wall or fell of the grid. sorry :(" << endl;

break;

default:

cout << "you did something very long because you aren't even getting a correct digit returned....wooowwwww..." << endl;

}

cout << "nsteps: " << stepsyo << endl;

}

bool isRouteWellFormed(string route)

{

for (int position = 0; position != route.size(); position++)

{

if (isdigit(route[0]))

return false;

if (isalpha(route[position]))

{

char letter = tolower(route[position]); // establish this character to ease following if statement

if (letter == 'n' || letter == 's' || letter == 'e' || letter == 'w') // if the letter is n, s, e, or w, continue (position++)

continue;

else

return false;

}

else if (isdigit(route[position]))

{

int digit1 = route[position]; // simplify calculations by adding integer

int digit2 = position + 1; // digit 2 indicates the character following the digit

if (digit1 < 0) // if the first digit is negative, the string is not correct, therefore return false

return false;

else if (isdigit(route[digit2])) // if the digit is positive, check to see if the next character is a digit

{

if (digit2 < 0) // make sure the next digit is nonnegative

return false;

int digit3 = position + 2; // digit 3 indicates the character two positions away from the first digit

if (isdigit(route[digit3])) // if the 3rd character in this sequence is a digit, then the string is not in the proper form

{

return false;

}

else

continue;

}

else

continue;

}

else if (position != (route.size() - 1)) // if none of the above if statements apply, and we have not reached the end of the string

return false; // then there is a character that is not a letter or a digit in the string, making it invalid

}

return true;

}

int navigateSegment(int r, int c, char dir, int maxSteps)

{

int nSteps = -1; // start at -1 steps because the first step in the while loop is just the starting position, not motion to a new step

if (r > getRows() || c > getCols()) // if the point given is outside of the grid, point not valid

return -1;

else if (isWall(r, c)) // if the point given is a wall, point not valid

return -1;

else

{

while (getRows() >= r && getCols() >= c && r > 0 && c > 0) // while the point is within the grid

{

if (isWall(r,c)) // if the position is a wall, end the path

break;

else

nSteps++; // if the position is not a wall, count it as a step

if (tolower(dir) == 'n') // if dir is north, check the point one row above it

r--;

if (tolower(dir) == 's') // if dir is south, check the point one row below it

r++;

if (tolower(dir) == 'e') // if dir is east, check the point one column to the right

c++;

if (tolower(dir) == 'w') // if dir is east, check the point one column to the left

c--;

}

if (nSteps >= maxSteps) // if the possible steps are greater than or equal to the max steps, return the maxsteps

return maxSteps;

else return nSteps; // if the possible steps are less than max steps, return the possible steps

}

}

bool clearPosition(int row, int column)

{

if (row > getRows() || column > getCols())

return false;

else if (row <= 0 || column <= 0) // make sure row and column are within the grid

return false;

else if (isWall(row, column)) // if it's inside the boundaries, check to see if there is a wall in new location

return false; // if there's a wall in the position you want to step to, return false (step not valid)

else return true; // if the new step passes these tests, return true because the step is valid

}

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps)

{

if (!isRouteWellFormed || !clearPosition(sr, sc) || !clearPosition(er, ec))

return 2;

int k = 0;

string newRoute; // modified route where all lower case letters

while (k != route.size()) // go through every character of the route string

{

if (isdigit(route[0]))

{

return 2;

}

if (isalpha(route[k] && route[k + 1] == '0'))

{

continue;

}

if (isalpha(route[k]))

{

newRoute += tolower(route[k]); // add the lowercase equivalent of that letter to newRoute

k++; // increase k to move onto next character in string

}

if (isdigit(route[k])) // if the character is a digit

{

int number = route[k] - '0'; // change the character to a number

if (isalpha(route[k - 1])) // if there is a letter before the digit

{

if (number == 0)

{

continue;

}

for (int j = (number - 1); j > 0; j--)

{

newRoute += tolower(route[k - 1]); // output that letter the digit number of times

}

k++; // increase k to move onto next character in string

}

else if (isdigit(route[k - 1])) // if the character before the digit is another digit

{

int twoDigitNumber = (10 \* (route[k - 1] - '0')) + (route[k] - '0'); // convert the number to the value of a two digit number

for (int n = (twoDigitNumber - 1); n > 0; n--)

{

newRoute += tolower(route[k - 2]); // output that letter the digit number of times

}

k++; // increase k to move onto the next character in string

}

}

}

if (!isRouteWellFormed(newRoute))

{

return 2;

}

int m = 0; // set me to zero, m is the position in the newRoute

int mr = sr; // moving row starts at the row of the start position

int mc = sc; // moving column starts at the column of the start position

int stepsMoved = 0; // stepsMoved will count up the possible number of steps

int possible\_steps = 0; // an integer set to the value of navigateSegment (makes later computations to move point (mr, mc) simpler)

while (m != newRoute.size()) // go through every character in the newRoute

{

possible\_steps = navigateSegment(mr, mc, newRoute[m], 1);

if (possible\_steps < 1) // if navigateSegment returns a number less than one

{

nsteps = stepsMoved; // return the number of steps you have taken so far

return 3; // return 3 because you hit a wall or fell off the grid

}

else

stepsMoved += possible\_steps; // add the number of steps possible in that direction

if (newRoute[m] == 'n') // add the change in position to moving position

mr -= possible\_steps; // if north, moving position goes up one row

else if (newRoute[m] == 's')

mr += possible\_steps; // if south, moving position goes down one row

else if (newRoute[m] == 'e')

mc += possible\_steps; // if east, moving position goes right one column

else if (newRoute[m] == 'w')

mc -= possible\_steps; // if west, moving position goes left one column

m++;

}

if (mr == er && mc == ec) // if the final point is the end point

{

nsteps = stepsMoved;

return 0;

}

else

{

nsteps = stepsMoved;

return 1;

}

}