```
/*--- Makefile ---*/
/*
#################################################################
# Do not copy and past this file.  If you do, you won't get the
# tab in front of the clang++.  Instead download and save!
#
# Edit the compilation lines to reflect any additional .cpp
# that will need to be compiled for your code.  Example:
#
# make part1
#################################################################
.PHONY: part1 part2 part3 part4 part5 partX

part1:
        clang++ -Wall -o part1 part1.cpp Pos.cpp Node.cpp easycurses.cpp -l ncurses
########
# preceeding the clang++ must be a tab and nothing but a tab!

part2:
        clang++ -Wall -o part2 part2.cpp Pos.cpp Point.cpp Board.cpp easycurses.cpp -l nc
urses
########
# preceeding the clang++ must be a tab and nothing but a tab!

part3:
        clang++ -Wall -o part3 part3.cpp Point.cpp Board.cpp Pos.cpp easycurses.cpp -l nc
urses
########
# preceeding the clang++ must be a tab and nothing but a tab!

part4:
        clang++ -Wall -o part4 part4.cpp Point.cpp Board.cpp Pos.cpp easycurses.cpp -l nc
urses
########
# preceeding the clang++ must be a tab and nothing but a tab!

part5:
        clang++ -Wall -o part5 part5.cpp Pos.cpp Point.cpp Board.cpp easycurses.cpp Node.
cpp -l ncurses
########
# preceeding the clang++ must be a tab and nothing but a tab!

partX:
        clang++ -Wall -o partX partX.cpp easycurses.cpp Point.cpp Board.cpp Node.cpp -l n
curses
########
# preceeding the clang++ must be a tab and nothing but a tab!
\n*/

/*--- Point.h ---*/
#ifndef GEORGEPOINT
#define GEORGEPOINT
#include <cstdlib>

struct point {
  char cVal;
  int x, y, dir;
  int lastX, lastY, lastDir;
};

//draws a point array p of length n at their position. Ignore is a special case to not pr
int out Zs and Ys.
void drawPoints(point* p, int n, bool ignore=false);
//takes in an array of points of n length and 'erases' them from the screen by replacing
them with a space character
void delPoints(point* p, int n);
```

```
//Collision function that takes in a point, an array of entities, int n for the length of
 the entity array, and a starting index. The starting index defaults to 1 and assumes
//that the player is at the first index of the entity array. Override this with 0 if this
 is not the case.
bool collision(point p, point* Players, int n, int stIndex=1);
//Function that takes in a point and flips the direction of the point. This direction is
used in the movePoint function.
void invertDir(point &p);
//rotates the point to the left if leftTurn is true or to the right if it is false
void rotateDir(point &p, bool leftTurn);

//increments the point position based on the direction and the distance specified. Used i
n conjunction with canMove
void movePoint(point &p, bool forward, int dist);

//returns true if the point is within the bounds specified as width and height. Forward i
s just if the point is moving forward or backwards.
bool canMove(point p, bool forward, int dist, int wid, int hei);
//takes in a point, a movement toggle, and the key pressed. Makes the point move in the k
ey direction of WASD and changes mvToggle accordingly.
void handleMove(point &p, char key, bool &mvToggle);

//amogn us!
int outOfBounds(point p, bool forward, int dist, int wid, int hei);

//yoinked from the Pos.h and made to work for my point struct.
int dist(point p, point q);
//overrides the assignment operator for simplicity purposes
// void operator=(point &a);

#endif

/*--- Point.cpp ---*/
#include "Point.h"
#include "easycurses.h"

//draws a point array p of length n at their position. Ignore is a special case to not pr
int out Zs and Ys.
void drawPoints(point* p, int n, bool ignore) {
  for(int i=0; i<n; i++) {
    //if ignore is false, cout everything
    if(!ignore) {
      drawChar(p[i].cVal, p[i].x, p[i].y);
    } else {
      //put a space in these spots to ignore the Z and Y
      if(p[i].cVal == 'Z' || p[i].cVal == 'Y') {
        drawChar(' ', p[i].x, p[i].y);
      } else {
        drawChar(p[i].cVal, p[i].x, p[i].y);
      }
    }
  }
}

//Function that takes in a point and flips the direction of the point. This direction is
used in the movePoint function.
void invertDir(point &p) {
  switch(p.dir) {
    case 0:
      p.dir = 2;
      break;
    case 1:
      p.dir = 3;
      break;
    case 2:
      p.dir = 0;
      break;
    case 3:
      p.dir = 1;
      break;
  }
}

//Collision function that takes in a point, an array of entities, int n for the length of
 the entity array, and a starting index. The starting index defaults to 1 and assumes
//that the player is at the first index of the entity array. Override this with 0 if this
 is not the case.
bool collision(point p, point* entities, int n, int stIndex) {
  for(int i=stIndex; i<n; i++) {
    if(p.x == entities[i].x && p.y == entities[i].y) {
      return true;
    }
    if(p.x == entities[i].lastX && p.y == entities[i].lastY && p.lastX == entities[i].x &
& p.lastY == entities[i].y) {
      return true;
    }
    //now check if P and and entity have passed through each other by having the same pre
vious positions
  //  After each object has made its step for the round, we will say that player P and st
ar S have collided
  //    both P's current position is the same as S's previous position and P's previous p
osition
  //is the same as S's current position.

  }
  return false;
}

//takes in an array of points of n length and 'erases' them from the screen by replacing
them with a space character
void delPoints(point* p, int n) {
  for(int i=0; i<n; i++) {
    drawChar(' ', p[i].x, p[i].y);
  }
}

//rotates the point :)
void rotateDir(point &p, bool leftTurn) {
  if(leftTurn) {
    //left turn
    if(p.dir > 0) {
      p.dir -= 1;
    } else {
      p.dir = 3;
    }
  } else {
    //right turn
    if(p.dir < 3) {
      p.dir += 1;
    } else {
      p.dir = 0;
    }
  }
}

//returns true if the point is within the bounds specified as width and height. Forward i
s just if the point is moving forward or backwards.
bool canMove(point p, bool forward, int dist, int wid, int hei) {
  int neg = 1;
  if(forward) {
    neg = 1;
  } else {
    neg = -1;
  }
  switch(p.dir) {
    case 0:
    //north
      p.x -= dist*neg;
      if(p.x < 0) {
        return false;
      }
      break;
    case 1:
    //east (right)
      p.y += dist*neg;
      if(p.y >= hei) {
        return false;
      }
      break;
    case 2:
    //south
      p.x += dist*neg;
      if(p.x >= wid) {
        return false;
      }
      break;
    case 3:
    //west (left)
      p.y -= dist*neg;
      if(p.y < 0) {
        return false;
      }
      break;
  }
  return true;
}

//sets either xB or yB to true if the point is past the x bounds or the y bounds. Similar
 to can move but for 2d instead of 1d. 0 north, 1 east... so on.
int outOfBounds(point p, bool forward, int dist, int wid, int hei) {
  int neg = 1;
  if(forward) {
    neg = 1;
  } else {
    neg = -1;
  }
  switch(p.dir) {
    case 0:
    //north
      p.x -= dist*neg;
      if(p.x < 0) {
        return 0;
      }
      break;
    case 1:
    //east (right)
      p.y += dist*neg;
      if(p.y >= hei) {
        return 1;
      }
      break;
    case 2:
    //south
      p.x += dist*neg;
      if(p.x >= wid) {
        return 2;
      }
      break;
    case 3:
    //west (left)
      p.y -= dist*neg;
      if(p.y < 0) {
        return 3;
      }
      break;
  }
  return 4;
}

//increments the point position based on the direction and the distance specified. Used in conjunction with canMove
void movePoint(point &p, bool forward, int dist) {
  p.lastX = p.x;
  p.lastY = p.y;
  p.lastDir = p.dir;
  int neg = 1;
  if(forward) {
    neg = 1;
  } else {
    neg = -1;
  }
  switch(p.dir) {
    case 0:
    //north
      p.x -= dist*neg;
      break;
    case 1:
    //east (right)
      p.y += dist*neg;
      break;
    case 2:
    //south
      p.x += dist*neg;
      break;
    case 3:
    //west (left)
      p.y -= dist*neg;
      break;
  }
}

void handleMove(point &p, char key, bool &mvToggle) {
  if(key == 'w') {
    mvToggle = true;
    p.dir = 0;
  }
  if(key == 's') {
    mvToggle = true;
    p.dir = 2;
  }
  if(key == 'a') {
    mvToggle = true;
    p.dir = 3;
  }
  if(key == 'd') {
    mvToggle = true;
    p.dir = 1;
  }
  if(key == 'r') {
    mvToggle = false;
  }
}

int dist(point p, point q) {
  return abs(p.y - q.y) + abs(p.x - q.x);
}

// //overwrites the assignment operator
// point operator=(point &a) {
//   point t;
//   t.x = a.x;
//   t.y = a.y;
//   t.dir = a.dir;
//   return t;
// }

/*--- Board.h ---*/
#ifndef GEORGEBOARD
#define GEORGEBOARD
#include "Point.h"
#include <fstream>
#include <iostream>

using namespace std;


struct Board {
  int height, width, maxSpawn, wallCount;
  point* spawnList;
  point** bArr;
  point playerSpawn;
  point goalSpawn;
};

//reads a board file and returns a board object
Board readFile(ifstream& f);

//print the board
void printBoard(Board b);
//returns true if the point is near the goal
bool isGoal(Board b, point p);

void destroyBoard(Board b);

#endif

/*--- Board.cpp ---*/
#include <fstream>
#include <iostream>
#include "Point.h"
#include "Board.h"

using namespace std;

Board readFile(ifstream& f) {
  //throwaway
  char c;
  //our board
  Board b;
  //read in header of file
  f >> b.height >> c >> b.width >> b.maxSpawn;
  f.get(c); // skip newline
  b.width++;
  //incremented to account for the newlines at the end of the thing
  //sp is spawn counter
  int spCount = 0;
  //initialitzing point arrays based on the data we read from the file
  b.spawnList = new point[b.maxSpawn];
  b.bArr = new point*[b.height];

  for(int r=0; r<b.height; r++) {
    b.bArr[r] = new point[b.width];
    for(int col=0; col<b.width; col++) {
      f.get(c);
      //if c is a hashtag count it as a wall
      if(c==35) {
        b.wallCount++;
      }
      point k;
      k.cVal = c;
      k.y = col;
      k.x = r;
      //add each point to bArr
      b.bArr[r][col] = k;
      if(c == 'Z') {
        //correct positions
        b.spawnList[spCount].y = col;
        b.spawnList[spCount].x = r;
        spCount++;
        //add x and y dadat
      }
      if(c == 'X') {
        b.goalSpawn.y = col;
        b.goalSpawn.x = r;
        //add x and y dadat
      }
      if(c == 'Y') {
        b.playerSpawn.y = col;
        b.playerSpawn.x = r;
        //add x and y dadat
      }
    }
  }
  return b;
}

void printBoard(Board b) {
  for(int r=0; r<b.height; r++) {
    drawPoints(b.bArr[r], b.width, true);
  }
}

bool isGoal(Board b, point p) {
  if(dist(p, b.goalSpawn) == 1) {
    return true;
  }
  return false;
}

void destroyBoard(Board b){
 for(int r=0; r<b.height; r++) {
   delete [] b.bArr[r];
 }
 delete [] b.bArr;
 delete [] b.spawnList;
}

/*--- part5.cpp ---*/
#include "easycurses.h"
#include "Pos.h"
#include "Point.h"
#include "Board.h"
#include <unistd.h>
#include "Node.h"

//~/bin/submit -c=SI204 -p=proj03 Makefile part5.cpp Board.cpp Point.cpp Node.cpp Node.h
easycurses.cpp Pos.cpp Pos.h easycurses.h Board.h Point.h board2Rm.txt boardCenter.txt bo
ardMaze.txt boardTiny.txt board243354.txt


using namespace std;
```

```
bool game(string boardName, int numStar, int numKill, int score, int &totalScore);

int main() {
  sNode* n = NULL;
  cout << "Enter script filename: ";
  string filen, boardName, numStar, numKill, trash, score;
  cin >> filen;
  ifstream f(filen);
  if(!f) {
    cout << "Error! File not found.";
    return 1;
  }
  while(f >> boardName >> numStar >> numKill >> trash >> trash >> score) {
    string* dat = new string[4];
    dat[0] = boardName;
    dat[1] = numStar;
    dat[2] = numKill;
    dat[3] = score;
    addNode(n, dat);
  }
  printLinkListRev(n);

  //traverses linked list in reverse. now get the data
  int maxLevel = getLinkListLen(n);
  int level =0;
  int totalScore = 0;
  string* a;
  for(int i=0; i<maxLevel; i++) {
    //0
    //run 0 times, then 1 times, then 2...
    for(sNode *p = n; level < maxLevel-i; p = p->next) {
      a = p->data;
      level++;
    }
    //right here amog is the string of data for starting the game.
    int loserCount = 0;
    //play game with these parameters and store win output in a boolean
    while((!game(a[0], stoi(a[1]), stoi(a[2]), stoi(a[3]), totalScore)) && loserCount < 3
) {
      //while I lose keep playing until I lose three times or win.
      loserCount++;
    }
    if(loserCount == 3) {
      cout << "3 consecutive deaths. Game Over.\n";
      cout << "You scored: " << totalScore << " points. Try again!";
      return 0;
    }
    level = 0;
  }
  cout << "Victory! You cleared " << maxLevel << " maps and scored a total of " << totalS
core << " points!";

  return 0;
}

bool game(string boardName, int numStar, int numKill, int score, int &totalScore) {
  //setup the game from file
  ifstream f(boardName);
  if(!f) {
    cout << "Error! File not found.";
    return false;
  }

  //shitty name ngl
  Board t;
  //load in the board
  t= readFile(f);
  int wid = 0, hei = 0, turns = 0;
  const int DELAY = 100000;
  const int SEC = 1000000;
  //TODO: MOVE THIS TO THE BOARD H OR CPP TO SIMPLIFY

  //count of "player" entities in the game
  int starCount = numStar;
  int pCount = 1+starCount+numKill;

  //player point object
  point Player;
  Player.cVal = 'P';
  Player.y = t.playerSpawn.y;
  Player.x = t.playerSpawn.x;
  Player.lastX = t.playerSpawn.x;
  Player.lastY = t.playerSpawn.y;
  //is the player moving?
  bool mvToggle = false;

  //setup killers/other and walls
  point* objs = new point[pCount];
  point* walls = new point[t.wallCount];
  //add objects for the player to contend with to the objects list
  objs[0] = Player;
  int ind = 1;
  for(int i=0; i<starCount/5; i++) {
    //for every spawn, ther is one spawn point and 5 stars
    for(int s=0; s<5; s++) {
      t.spawnList[i].cVal = '*';
      objs[ind] = t.spawnList[i];
      objs[ind].x = t.spawnList[i].x;
      objs[ind].y = t.spawnList[i].y;
      //pick random starting direction
      objs[ind].dir = rand() % 4;
      //avoid having these being undefined in collision function
      objs[ind].lastDir = objs[ind].dir;
      objs[ind].lastX = t.spawnList[i].x;
      objs[ind].lastY = t.spawnList[i].y;
      //increment object index
      ind++;
    }
  }
  for(int i=0; i<numKill; i++) {
    t.spawnList[i].cVal = 'K';
    objs[ind] = t.spawnList[i];
    objs[ind].x = t.spawnList[i].x;
    objs[ind].y = t.spawnList[i].y;
    //pick random starting direction
    objs[ind].dir = rand() % 4;
    //avoid having these being undefined in collision function
    objs[ind].lastDir = objs[ind].dir;
    objs[ind].lastX = t.spawnList[i].x;
    objs[ind].lastY = t.spawnList[i].y;
    //increment object index
    ind++;
  }
  //add all points from the board to walls for collision function
  int wc = 0;
  for(int r=0; r<t.height; r++) {
    for(int col=0; col<t.width; col++) {
      if((t.bArr[r][col]).cVal == '#') {
        walls[wc] = t.bArr[r][col];
        wc++;
      }
    }
  }
```

```
  bool win = false;
  startCurses();
  getWindowDimensions(wid, hei);
  //game loop
  do {
    //get user input
    char key = inputChar();


    //mv logic for enemies
    //delete everything to update the positions
    delPoints(objs, pCount);
    for(int i=0; i<pCount; i++) {
      if(i != 0) {
        if(i<(pCount-t.maxSpawn)) {
          //movement logic for the stars
          int rando = rand() % 10 + 1;
          if(rando == 1) {
            int turnRand = rand() % 2 + 1;
            if(turnRand == 1) {
              //turn left
              rotateDir(objs[i], true);
            } else {
              //turn right
              rotateDir(objs[i], false);
            }
          }

          movePoint(objs[i], true, 1);
          if(collision(objs[i], walls, t.wallCount, 0)) {
            invertDir(objs[i]);
            movePoint(objs[i], true, 1);
          }
        } else {
          //killer movement logic
//           1. let dc = Player column position - Killer column position
// 2. let dr = Player row position - Killer row position
// 3. if dc < 0 let cdir = 3 else let cdir = 1
// 4. if dr < 0 let rdir = 0 else let rdir = 2
// 5. with prob 1/2 set Killer's direction to rdir, otherwise set Killer's direction to c
dir
          int dc = Player.y-objs[i].y; int dr = Player.x-objs[i].x;
          int cdir = 1, rdir = 2;
          int rando = rand() % 2 + 1;

          if(dc < 0) {
            cdir = 3;
          } else {
            cdir = 1;
          }

          if(dr < 0) {
            rdir = 0;
          } else {
            rdir = 2;
          }

          if(rando == 1) {
            int turnRand = rand() % 2 + 1;
            if(turnRand == 1) {
              //set killer dir to rdir
              objs[i].dir = rdir;
            } else {
              //set killer dir to cdir
              objs[i].dir = cdir;
            }
          }

          movePoint(objs[i], true, 1);
          if(collision(objs[i], walls, t.wallCount, 0)) {
            invertDir(objs[i]);
            movePoint(objs[i], true, 1);
          }
        }

      } else {

        //player movement logic
        handleMove(objs[i], key, mvToggle);
        if(mvToggle) {
          movePoint(objs[i], true, 1);
        }

        if(collision(objs[i], objs, pCount)) {
          //end game because the player was killed
          key = 'y';
          win = false;
          usleep(SEC*2);
        }
        //check if goalS
        if(isGoal(t, objs[i])) {
          key = 'y';
          win = true;
        }
        //wall collision check
        if(collision(objs[i], walls, t.wallCount, 0)) {
          invertDir(objs[i]);
          movePoint(objs[i], true, 1);
        }
      }
    }
    //display the updated board
    printBoard(t);
    //draw all of the entities
    drawPoints(objs, pCount);
    usleep(DELAY);
    //tick counter for score
    turns++;

    if (key == 'y') {  // game exits with a 'y'
      break;
    }
  } while(true);

  endCurses();
  //ending data'
  cout << " Playing on: " << boardName << ", with killer count: " << numKill << ", with s
tar per Z: " << numStar << ", for a possible " << score << " points.\n";
  if(win) {
    cout << "Victory!\n";
  } else {
    cout << "Defeat...\n";
  }
  cout << "Score: " << 500-turns << endl;
  totalScore += score+500-turns;
  destroyBoard(t);
  delete [] objs;
  delete [] walls;
  return win;
}

/*--- Node.h ---*/
#ifndef GEORGENODE
```

```
#define GEORGENODE
#include <iostream>

using namespace std;

struct Node {
  char data;
  Node* next;
};

struct sNode {
  string* data;
  sNode* next;
};

int getLinkListLen(Node* firstNode);

void printLinkList(Node* firstNode);

void printLinkListRev(Node* firstNode);

void addNode(Node* &lastNode, char data);

//from course notes
void deleteFirstNode(Node* &L);
//from course notes
void deleteList(Node* L);


//same thing for a string* node
int getLinkListLen(sNode* firstNode);

void printLinkList(sNode* firstNode);

void printLinkListRev(sNode* firstNode);

void addNode(sNode* &lastNode, string* data);

//from course notes
void deleteFirstNode(sNode* &L);
//from course notes
void deleteList(sNode* L);

#endif

/*--- Node.cpp ---*/
#include "Node.h"
#include <iostream>

using namespace std;

int getLinkListLen(Node* firstNode) {
  int count = 0;
  for(Node *curr = firstNode; curr != NULL; curr = curr->next)  {
     // record that we've visited the node pointed to by curr
     count++;
  }
  return count;
}

void printLinkListRev(Node* firstNode) {
  if(firstNode == NULL) {
    return;
  }
  printLinkListRev(firstNode->next);
  if(firstNode->data != 'Z') {
    cout << firstNode->data;
  } else {
    cout << ' ';
  }
}

void printLinkList(Node* firstNode) {
  //ignore printing the Zs
  for(Node *p = firstNode; p != NULL; p = p->next) {
    cout << p->data;
  }
  cout << '\n';
}

void addNode(Node* &lastNode, char data) {
  Node* n = new Node;
  n->data = data;
  n->next = lastNode;
  lastNode = n;
}

//from course notes
void deleteFirstNode(Node* &L) {
  Node *T = L;
  L = L->next;
  delete T;
}
//from course notes
void deleteList(Node* L) {
  while(L != NULL) {
    deleteFirstNode(L);
  }
}

//snode overloading
int getLinkListLen(sNode* firstNode) {
  int count = 0;
  for(sNode *curr = firstNode; curr != NULL; curr = curr->next)  {
     // record that we've visited the node pointed to by curr
     count++;
  }
  return count;
}

void printLinkListRev(sNode* firstNode) {
  if(firstNode == NULL) {
    return;
  }
  printLinkListRev(firstNode->next);
    for(int i=0; i<4; i++) {
    cout << (firstNode->data)[i] << ' ';
  }
  cout << '\n';
}

void printLinkList(sNode* firstNode) {
  //ignore printing the Zs
  for(sNode *p = firstNode; p != NULL; p = p->next) {
    cout << p->data;
  }
  cout << '\n';
}

void addNode(sNode* &lastNode, string* data) {
  sNode* n = new sNode;
  n->data = data;
  n->next = lastNode;
  lastNode = n;
}

//from course notes
void deleteFirstNode(sNode* &L) {
  sNode *T = L;
  L = L->next;
  delete T;
}
//from course notes
void deleteList(sNode* L) {
  while(L != NULL) {
    deleteFirstNode(L);
  }
}

/*--- board243354.txt ---*/
/*
23 x 50 5
##################################################
#######-----------A M O G U S !-----------########
##################################################
#    X    #                                      #
#         #             Z                 Z      #
#         #                                      #
#         #         ##########                   #
#         #             #                        #
#        #              #          #####         #
#        #                             #         #
#       #               Z              #         #
#      #                               #         #
#     #                                          #
#     #                                          #
#    #                #######          #         #
#    #                #                #         #
#    #                #                #         #
#                                      #         #
#        Z                             #         #
#                     #                #         #
#        Z            #                #         #
#                     #                #    Y    #
##################################################
\n*/
```