به نام خدا

گزارش کار پروژه میانی

پرتو اشراقی 9623403

https://github.com/Parto-E/PartoEshraghi

تو ضیحات کامل Maze generator به صورت کامنت نوشته شده:

این کد در یک Nested classes تعریف شده به این صورت که یک کلاس درونی Cell داریم و کلاس اصلی Maze

در کلاس Cell یه سرس از تابع های مورد نیاز برای کار کرد هر کدام از مربع های کوچک که در شکل قبل هم دیدیم نوشته شده.

در کلاس Maze هم به کمک کلاس Cell خود Maze طراحی و نمایش داده میشود.

کلاس Cell:

```
enum WALL { WALL_NORTH = 0x0008, WALL_EAST = 0x0004,
            WALL_SOUTH  = 0x0002, WALL_WEST = 0x0001,
            WALL_ALL = 0x000f, WALL_NONE = 0x0000 };
```

اول از همه متغیرهای بالا را تعریف میکنیم که با آن ها تعداد دیوارها و اینکه هر سلول کدام دیوار را دارد مشخص میکنیم.

یک تابع init داریم که در واقع به کمک کانستراکتورها مقداردهی متغیرها را برای ما انجام میدهد.

```
bool visited() const;
void setVisited(const bool v = true);
```

این دو تابع مشخص میکنند آیا یک سلول بازدید شده یا نه.

```
void removeWall(const int w);
int getWalls() const;
void setWalls(const int w);
```

با استفاده از این توابع در صورت نیاز دیوار یک سلول را حذف کرده یا از اینکه سلول کدام دیوارها را دارد یا برای اضافه کردن دیوار استفاده میکنیم.

```
void setPosition(const int r, const int c);
```

این تابع موقعیت ما در بین سلول ها را مشخص میکند.

```
std::ostream& operator<<(std::ostream& strm, const Maze::Cell& c) {
        if ((c.getWalls() & Maze::Cell::WALL_WEST) != 0) strm << '|';
        else strm << ' ';
        if ((c.getWalls() & Maze::Cell::WALL_SOUTH) != 0) strm << '_';
        else strm << ' ';

    return strm;
}
```

آخرین تابع هم برای نمایش خود مارپیچ هست به این گونه که اگر دیوار غربی داشته باشد سلولی یک | میگذارد و اگر دیواره جنوبی داشته باشد یک _ میگذارد اگر هم نداشته باشد صرفا یک جای خالی میگذارد.

کلاس Maze:

- در این کلاس تنها یک کانستراکتور داریم که برای مشخص کردن ابعاد و نوع حل است، و یک تابع استارت داریم که در واقع با فراخوانی آن کل Maze طراحی شده و نمایش داده میشود.
- در تابع استارت در واقع با یک آرایه دو بعدی و دو وکتور داریم پیش میرویم.
- آرایه دو بعدی ما از کلاس Cell هست و درابتدا دیوارهای تمام سلول های آن را کامل میکنیم.
- در وکتور trail مسیر حرکت را ذخیره میکنیم و در وکتور live همسایه هایی که هر سلول میتواند به آنها برود را به صورت enum ذخیره میکنیم.
- در این وکتور برای اینکه بفهمیم کدام همسایه ها قابل دسترسی هستند به صورت زیر عمل میکنیم:

```
    if(curY)
        if(maze[curX][curY-
1].getWalls()==Cell::WALL_ALL) // West has all walls
            live.push_back(WEST);
    if(curY<COLS-1)
```

```
if(maze[curX][curY+1].getWalls()==Cell::WALL_ALL) // east has all walls
    live.push_back(EAST);
if(curX)
    if(maze[curX-1][curY].getWalls()==Cell::WALL_ALL) // North has all walls
    live.push_back(NORTH);
if(curX<ROWS-1)
    if(maze[curX+1][curY].getWalls()==Cell::WALL_ALL) // South has all walls
    live.push_back(SOUTH);
```

بعد از آنکه همسایه های قابل دسترسی مشخص شدند به صورت تصادفی یکی از آنها رو مشخص میکنیم و وارد آن میشویم اما باید حواسمان به پاک کردن دیوارهایی که از آنها عبور کردیم باشد:

```
switch(live[rand() % live.size()]) {
    case 0: //NORTH
        maze[curX][curY].removeWall(Cell::WALL_NORTH);
        maze[--curX][curY].removeWall(Cell::WALL_SOUTH);
        break;
```

مثلا برای جهت شمال به این صورت عمل میکنیم.

مهمترین قسمت هم زمانی که به جایی رسیدیم که آخر خط بودم دیگه راهی نداشتیم باید به عقب برگردیم پس کافیه آخرین المان وکتور trail را حذف کنیم تا جایی که به دو راه خورده بودیم و حالا راه جدید انتخاب میشود راه قدیم هم چون یه سری از دیواره هاش پاک شده دیگر در وکتور live قرار نمیگیرد و تکرار نمیشود.

و بعد هم موقعیت باید به آخرین مورد در وکتور trail باز گردد.

```
    trail.pop_back();
    if(trail.empty()==false) {
    int n = trail.size();
    curX=trail[n-1].getRow();
```

curY=trail[n-1].getColumn();

در نهایت انقدر trail را pop میکنیم تا خالی شود و دیگر مارپیچ بدست آمده و فقط باید نمایش داده شود.

Maze.h:

```
#ifndef __MAZE_H_
#define __MAZE_H_

#include <string>
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <vector>

class Maze{
    public:
        class Cell {
        private:
            int row;
            int col;
            bool visit;
            int walls;
            void init(const int r, const int c, const int walls, const bool v = f
alse);
        public:
            enum WALL { WALL_NORTH = 0x0008, WALL_EAST = 0x0004,
                WALL_SOUTH  = 0x0002, WALL_WEST = 0x0001,
                WALL_ALL = 0x000f, WALL_NONE = 0x0000 };
            Cell();
            Cell(const int r, const int c);
            Cell(const int r, const int c, const int stat);
            bool visited() const;
            void setVisited(const bool v = true);
            int getRow() const;
            int getColumn() const;
            void removeWall(const int w);
            int getWalls() const;
            void setWalls(const int w);
            void setPosition(const int r, const int c);
            friend std::ostream& operator<<(std::ostream& strm, const Cell& c);
        };
```

```
        Maze(int ROWS, int COLS, int solvation);
        void start();

    private:
        int ROWS;
        int COLS;
        int solvation;
        enum DIR { NORTH, SOUTH, EAST, WEST };
};

#endif
```

Maze.cpp:

```
#include "maze.h"
#include <unistd.h>

using std::cin;
using std::cout;
using std::vector;
using std::stack;
#define underline "\033[4;37m"

void Maze::Cell::init(const int r, const int c, const int walls, const bool v) {
    setPosition(r, c);
    setWalls(walls);
    setVisited(v);
}
Maze::Cell::Cell() { init(0, 0, 0); }
Maze::Cell::Cell(const int r, const int c) { init(r, c, 0); }
Maze::Cell::Cell(const int r, const int c, const int walls) { init(r, c, walls);
}
bool Maze::Cell::visited() const { return visit; }
void Maze::Cell::setVisited(const bool v) { visit = v; }
int Maze::Cell::getRow() const { return row; }
int Maze::Cell::getColumn() const { return col; }
void Maze::Cell::removeWall(const int w) {
    if (w!=WALL_NORTH && w!=WALL_EAST && w!=WALL_SOUTH && w!=WALL_WEST)
        throw std::string("Illegal wall argument");
    walls &= ~w;
```

```
}
int Maze::Cell::getWalls() const { return walls & WALL_ALL; }
void Maze::Cell::setWalls(const int w) { walls = w & WALL_ALL; }
void Maze::Cell::setPosition(const int r, const int c) { row = r; col = c; }

std::ostream& operator<<(std::ostream& strm, const Maze::Cell& c) {
            if ((c.getWalls() & Maze::Cell::WALL_WEST) != 0) strm << '|';
            else strm << ' ';
            if ((c.getWalls() & Maze::Cell::WALL_SOUTH) != 0) strm << '_';
            else strm << ' ';

    return strm;
}

Maze::Maze(int _ROWS, int _COLS, int _solvation)
    : ROWS{ _ROWS }
    , COLS{ _COLS }
    , solvation{ _solvation }
{}

void Maze::start()
{
    // Randomize the random number function.
    srand(time(NULL));

    // Create a 2-D array ([ROWS][COLS]) of Cell objects.
    Cell maze[ROWS][COLS];

    // For each Cell in the maze:
    for(int row = 0; row < ROWS; row++)
       for(int col = 0; col < COLS; col++) {
          // set visited to false
          maze[row][col].setVisited(false);
          // set its position to its row and column in the maze
          maze[row][col].setPosition(row, col);
          // set the Cell's walls to Cell::WALL_ALL
          maze[row][col].setWalls(Cell::WALL_ALL);
    }

    //Create curX and curY variables and set them to a random position in the maz
e.
    int curX = 0;
    int curY = 0;

    // Create a vector of Cell objects named trail which will be used as a stack.
```

```
    vector<Cell> trail;

    // Create a vector of DIR values named live.
    vector<DIR> live;

    // Grab the Cell at the curX, curY position and push it on the trail stack.
    trail.push_back(maze[curX][curY]);
    ++visited;
    // While the trail stack is not empty do the following:
    while(trail.empty()==false) { // stay in here till display
        // Empty the live vector.

        live.clear();
        // Check the neighbors of the current cell to the north, east, south, and
 west.
        // If any of the neighbors have all four walls, add the direction to that

        // neighbor to the live vector.
        if(curY)
            if(maze[curX][curY-
1].getWalls()==Cell::WALL_ALL) // West has all walls
                live.push_back(WEST);
        if(curY<COLS-1)
            if(maze[curX][curY+1].getWalls()==Cell::WALL_ALL) // east has all wal
ls
                live.push_back(EAST);
        if(curX)
            if(maze[curX-
1][curY].getWalls()==Cell::WALL_ALL) // North has all walls
                live.push_back(NORTH);
        if(curX<ROWS-1)
            if(maze[curX+1][curY].getWalls()==Cell::WALL_ALL) // South has all wa
lls
                live.push_back(SOUTH);
        // If the live vector is not empty:
        if(live.empty()==false) {
        // Choose one of the directions in the live vector at random
        // Remove the walls between the current cell and the neighbor in that dir
ection
        // and Change curX and curY to refer to the neighbor
            maze[curX][curY].setVisited(true);
            switch(live[rand() % live.size()]) {
                case 0: //NORTH
                    maze[curX][curY].removeWall(Cell::WALL_NORTH);
                    maze[--curX][curY].removeWall(Cell::WALL_SOUTH);
```

```
                visited++;
                break;
            case 1: //SOUTH
                maze[curX][curY].removeWall(Cell::WALL_SOUTH);
                maze[++curX][curY].removeWall(Cell::WALL_NORTH);
                visited++;
                break;
            case 2: //EAST
                maze[curX][curY].removeWall(Cell::WALL_EAST);
                maze[curX][++curY].removeWall(Cell::WALL_WEST);
                visited++;
                break;
            case 3: //WEST
                maze[curX][curY].removeWall(Cell::WALL_WEST);
                maze[curX][--curY].removeWall(Cell::WALL_EAST);
                visited++;
                break;
        }
        // Push the new current cell onto the trail stack
        trail.push_back(maze[curX][curY]);
    }     //If the live vector was emtpy:
    else {
    // Pop the top item from the trail stack
    if(visited == ROWS*COLS){
        x_target = curX;
        y_target = curY;
        visited = 0;
    }
        trail.pop_back();
    // position of the top item in the trail stack.
        if(trail.empty()==false) {
           int n = trail.size();
           curX=trail[n-1].getRow();
           curY=trail[n-1].getColumn();
        }

    }
}

cout << y_target << "  " << x_target << std::endl;

// Do the following to display the maze:
int r, c;
for (c=0; c<COLS; c++) {
    if (c == 0) cout << "\033[35m _";
```

```
        else cout << "__";
    }
    cout << '\n';
    for (r=0; r<ROWS; r++) {
        for (c=0; c<COLS; c++) {
            if(r == 0 && c == 0){
                if ((maze[r][c].getWalls() & Maze::Cell::WALL_WEST) != 0)
                cout << '|';
                else cout << ' ';
                if ((maze[r][c].getWalls() & Maze::Cell::WALL_SOUTH) != 0) cout <
< underline << "\033[35m#\033[0m" << "\033[35m";
                else cout << "#";
            }
            else if(r == x_target && c == y_target){
                if ((maze[r][c].getWalls() & Maze::Cell::WALL_WEST) != 0)
                cout << '|';
                else cout << ' ';
                if ((maze[r][c].getWalls() & Maze::Cell::WALL_SOUTH) != 0) cout <
< underline << "\033[35m*\033[0m" << "\033[35m";
                else cout << "*";
            }
            else
                cout << maze[r][c];
        }
        cout << "|\n";
    }
    cout << "\033[0m\n";

    usleep(5000000);
```

- الگوریتم DFS هم بسیار شبیه به الگوریتم اصلی هست با این تفاوت که اینجا به جای پاک و جا به جا کردن دیوارها آنجاهایی که دیوار ندارند را تشخیص داده و به آنجا میرویم و برای اینکه هرجا را چندبار نرویم به عنوان visited مشخصشان میکنیم. برای سلول‌هایی که می‌توانییم به آن‌ها برویم دو حالت در نظر می‌گیریم یگر فقط یه سلول باشد و اگر بیش از دو تا باشد اگر دو تا باشد باید یکی را آنقدر ادامه دهد تا به بنبست بخورد بعد شروع به پاک کردن می‌کند تا دوباره به جایی که دو انتخاب داشته باز گردد و آنجا راه جدیدی را طی می‌کند.

```
    //DFS Algorithm

    if(solvation == 1){
        cout << "\033[1;32mNow you'll see the DFS algorithm for solving this maze
\033[0m\n";
```

```
        cout << "\033[1;32mBut first remember:\033[0m\n";
        usleep(3000000);
        cout << "\033[1;36mFirst argument shows your horizontal movement\033[0m\n
";
        cout << "\033[1;36mSecond argument shows your vertical movement\033[0m\n\
n";
        usleep(3000000);
        for(int row = 0; row < ROWS; row++)
        for(int col = 0; col < COLS; col++) {
            maze[row][col].setVisited(false);
            maze[row][col].setPosition(row, col);
        }

        int curX1 = 0;
        int curY1 = 0;

        vector<Cell> trail1;

        vector<DIR> live1;

        trail1.push_back(maze[curX1][curY1]);

        maze[curX1][curY1].setVisited(true);

        while(maze[x_target][y_target].visited() == false) {
            live1.clear();
            if(curY1)
                if((maze[curX1][curY1-
1].getWalls() & Maze::Cell::WALL_EAST) == 0 && maze[curX1][curY1-
1].visited() == false)
                    live1.push_back(WEST);
            if(curY1<COLS-1)
                if((maze[curX1][curY1+1].getWalls() & Maze::Cell::WALL_WEST) == 0
 && maze[curX1][curY1+1].visited() == false)
                    live1.push_back(EAST);
            if(curX1)
                if((maze[curX1-
1][curY1].getWalls() & Maze::Cell::WALL_SOUTH) == 0 && maze[curX1-
1][curY1].visited() == false)
                    live1.push_back(NORTH);
            if(curX1<ROWS-1)
                if((maze[curX1+1][curY1].getWalls() & Maze::Cell::WALL_NORTH) ==
0 && maze[curX1+1][curY1].visited() == false)
                    live1.push_back(SOUTH);
```

```
            if(live1.size() == 1) {
                switch(live1[0]) {
                    case 0: //NORTH
                        maze[--curX1][curY1].setVisited(true);
                        visited++;
                        break;
                    case 1: //SOUTH
                        maze[++curX1][curY1].setVisited(true);
                        visited++;
                        break;
                    case 2: //EAST
                        maze[curX1][++curY1].setVisited(true);
                        visited++;
                        break;
                    case 3: //WEST
                        maze[curX1][--curY1].setVisited(true);
                        visited++;
                        break;
                    }
                trail1.push_back(maze[curX1][curY1]);
            }
            else if(live1.size() > 1){
                for (size_t I = 0; I < live1.size(); i++)
                {
                        switch(live1[i]) {
                            case 0: //NORTH
                                maze[--curX1][curY1].setVisited(true);
                                break;
                            case 1: //SOUTH
                                maze[++curX1][curY1].setVisited(true);
                                break;
                            case 2: //EAST
                                maze[curX1][++curY1].setVisited(true);
                                break;
                            case 3: //WEST
                                maze[curX1][--curY1].setVisited(true);
                                break;
                            }
                        trail1.push_back(maze[curX1][curY1]);
                        break;
                    }

            }
            else if(live1.empty() == true){
                    trail1.pop_back();
```

```
                    if(trail1.empty()==false) {
                    int n = trail1.size();
                    curX1=trail1[n-1].getRow();
                    curY1=trail1[n-1].getColumn();
                    }
                }
            }
            for (size_t I = 0; I < trail1.size(); i++)
            {
                cout << trail1[i].getColumn() << " " << trail1[i].getRow() << std::en
dl;
                usleep(100000);
            }
```

- در الگوریتم BFS همانطور که بیشتر استفاده می‌شود از queue استفاده شده که هر سلول را در خود ذخیره می‌کند بعد سلول‌های قابل دستیابی آن را پیدا کرده و در queue ذخیره می‌کند ولی سلول اولیه ره پاک می‌کند و در حلقه for به ترتیب برای تمام سلول‌های ذخیره شده در queue اینکار را تکرار می‌کند. برای مسیرهایی که می‌تواند برود هم در حلقه for دو حالت در نظر گرفته شده اگر یک مسیر داشت مانند الگوریتم‌های بالا عمل کند ولی اگر بیش از دو مسیر داشت ابتدا به یک مسیر می‌رود آن را در queue و trail(آنکه مسیر کلی را ذخیره می‌کند) ذخیره کرد و بعد دوباره به سلول بعد باز می‌گردد و از آن سلول مسیرهای دیگر را هم به همین ترتیب چک می‌کند.

```
    //BFS Algorithm

    if(solvation == 2){
        cout << "\033[1;32mNow you'll see the BFS algorithm for solving this maze
\033[0m\n";
        cout << "\033[1;32mBut first remember:\033[0m\n";
        usleep(3000000);
        cout << "\033[1;36mFirst argument shows your horizontal movement\033[0m\n
";
        cout << "\033[1;36mSecond argument shows your vertical movement\033[0m\n\
n";
        usleep(3000000);
        for(int row = 0; row < ROWS; row++)
        for(int col = 0; col < COLS; col++) {
            maze[row][col].setVisited(false);
            maze[row][col].setPosition(row, col);
        }
```

```
        int curX1 = 0;
        int curY1 = 0;

        vector<Cell> trail1;

        vector<DIR> live1;

        queue<Cell> queue;

        trail1.push_back(maze[curX1][curY1]);

        queue.push(maze[curX1][curY1]);

        maze[curX1][curY1].setVisited(true);

        while(maze[x_target][y_target].visited() == false) {

            for(size_t j = 0; j < queue.size(); j++){
                curX1 = queue.front().getRow();
                curY1 = queue.front().getColumn();
                live1.clear();
                if(curY1)
                    if((maze[curX1][curY1-
1].getWalls() & Maze::Cell::WALL_EAST) == 0 && maze[curX1][curY1-
1].visited() == false)
                        live1.push_back(WEST);
                if(curY1<COLS-1)
                    if((maze[curX1][curY1+1].getWalls() & Maze::Cell::WALL_WEST)
== 0 && maze[curX1][curY1+1].visited() == false)
                        live1.push_back(EAST);
                if(curX1)
                    if((maze[curX1-
1][curY1].getWalls() & Maze::Cell::WALL_SOUTH) == 0 && maze[curX1-
1][curY1].visited() == false)
                        live1.push_back(NORTH);
                if(curX1<ROWS-1)
                    if((maze[curX1+1][curY1].getWalls() & Maze::Cell::WALL_NORTH)
 == 0 && maze[curX1+1][curY1].visited() == false)
                        live1.push_back(SOUTH);

                for (size_t I = 0; I < live1.size(); i++)
                {
                    if(live1.size() == 1){
                        switch(live1[i]) {
```

```
                    case 0: //NORTH
                        maze[--curX1][curY1].setVisited(true);
                        break;
                    case 1: //SOUTH
                        maze[++curX1][curY1].setVisited(true);
                        break;
                    case 2: //EAST
                        maze[curX1][++curY1].setVisited(true);
                        break;
                    case 3: //WEST
                        maze[curX1][--curY1].setVisited(true);
                        break;
                }
                trail1.push_back(maze[curX1][curY1]);
                queue.push(maze[curX1][curY1]);
            }
            else if(live1.size() > 1){
                switch(live1[i]){
                    case 0: //NORTH
                        maze[--curX1][curY1].setVisited(true);
                        trail1.push_back(maze[curX1][curY1]);
                        queue.push(maze[curX1][curY1]);
                        curX1++;
                        break;
                    case 1: //SOUTH
                        maze[++curX1][curY1].setVisited(true);
                        trail1.push_back(maze[curX1][curY1]);
                        queue.push(maze[curX1][curY1]);
                        curX1--;
                        break;
                    case 2: //EAST
                        maze[curX1][++curY1].setVisited(true);
                        trail1.push_back(maze[curX1][curY1]);
                        queue.push(maze[curX1][curY1]);
                        curY1--;
                        break;
                    case 3: //WEST
                        maze[curX1][--curY1].setVisited(true);
                        trail1.push_back(maze[curX1][curY1]);
                        queue.push(maze[curX1][curY1]);
                        curY1++;
                        break;
                }
            }
        }
```

```
                queue.pop();
            }
        }
        for (size_t I = 0; I < trail1.size(); i++)
        {
            cout << trail1[i].getColumn() << " " << trail1[i].getRow() << std::en
dl;
            usleep(100000);
        }

    }

}
```

```
Please enter the number of ROWS and COLOUMNS respectively
4
4
Now choose the solvation algorithm:1)DFS 2)BFS
1
Now you're going to see the maze you wanted :)
And remember you must get from # to *
2  2
 _______
|#  |   |
| | | | |
| | |*| |
|_ _ _ _|

Now you'll see the DFS algorithm for solving this maze
But first remember:
First argument shows your horizontal movement
Second argument shows your vertical movement

0 0
1 0
1 1
1 2
1 3
2 3
3 3
3 2
3 1
3 0
2 0
2 1
2 2
```

```
Please enter the number of ROWS and COLOUMNS respectively
4
4
Now choose the solvation algorithm:1)DFS 2)BFS
2
Now you're going to see the maze you wanted :)
And remember you must get from # to *
1  0
 _______
|#|* _   |
| |   | |
| | |_  |
|_ _|_ _|

Now you'll see the BFS algorithm for solving this maze
But first remember:
First argument shows your horizontal movement
Second argument shows your vertical movement

0 0
0 1
0 2
0 3
1 3
1 2
1 1
2 1
2 2
3 2
3 1
3 3
3 0
2 3
2 0
1 0
```