Лабораторна робота №1

# **Виконав:**

# Студент 2 курсу групи ІП-71

# Мокроусов Владислав

**Перевірив:**

Пономаренко М. Н.

# ЗМІСТ

1. Ціль лабораторної роботи…………………………………………………….3

2. Постановка задачі..............................................................................................4

3. Алгоритм вирішення задачі…………………………………………………..5

4. UML-діаграмма………………………………………………………………..6

5. Код програми………………………………………………………………….7

6. Приклад роботи програми……………………………………………………19

7. Висновок………………………………………………………………………20

# 1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мети роботи – вивчити основні концепції об’єктно-оріентованого програмуванняю Вивчити особливости користування класів та об’єктів, а також особливості застосування конструкторів і деструкторів.

# 2 ПОСТАНОВКА ЗАДАЧІ

Визначити клас «Ферзь», з закритим елементом: масив клітинок – А (8х8). Визначити такі конструкори:

- конструктор ініціалізації, який розміщує у 8 клітинок 8 ферзів.

Визначити метод пошуку такого розміщення ферзів, щоб вони не били один одного. Ферзь може бити інші фігури по вертиклі, горизонтаі та діагоналі.

# 3 АЛГОРИТМ ВИРІШЕННЯ ЗАДАЧІ

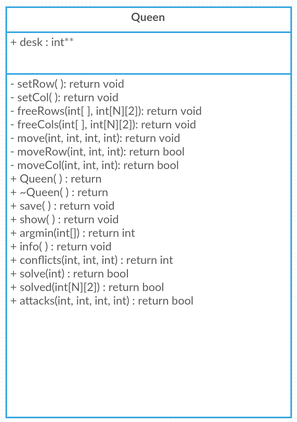
Конструктор випадково ініціалізує 8 королев на дошці.

Метод знаходження потрібного розміщення фігур проводиться у 2 кроки. На перший крок алгоритм розтавляє фігури так, щоб у кожній колонці та у кожному рядку було рівно по однії королеві. За це відповідають методи setCol() та setRow(), вони ітерують по колонкам та рядкам матриці та знаходять колонки/рядки в яких більше однієї фігури та переміщують їх до вільних колонок/рядків поки у кожному рядку/колонці не залишиться по одній фігурі.

Далі використовується алгоритм min-conflicts для знаходження потрібного розміщення. Алгоритм випадково обирає одну колонку та для кожної клітинки рахує кількість конфліктів (кількість інших фігур, що б’ють її). Потім він переміщує королеву з цієї колонки на цю клітинку. Він повторює ці дії доки розміщення не буде знайдене або доки не закінчиться кількість заданих ітерації.

# 4 UML ДІАГРАМА

Програма містить 1 клас – Queen. Нижче представлена діаграма класів. N – кількість ферзів на дошці.



# 5 КОД ПРОГРАМИ

Код програми міститься у файлах Queem.cpp, Queen.hpp, main.cpp, Makefile. Вивід відбувається до файлу output.txt.

// Файл Queen.hpp

#define N 8

class Queen {

int \*\*desk;

// arrange queens (by row/col)

void setRow();

void setCol();

// get free rows/cols

void freeRows(int[], int[N][2]);

void freeCols(int[], int[N][2]);

// move functions

void move(int, int, int, int);

bool moveRow(int, int, int);

bool moveCol(int, int, int);

public:

// constructors

Queen();

~Queen();

// helper functions

void save();

void show();

int argmin(int[]);

void info();

// main algorithm

int conflicts(int, int, int); // (ROW, COL, X), queen in (X, COL)

bool solve(int); // num iterations as parameter

bool solved(int[N][2]);

bool attacks(int, int, int, int);// check if queen (x1, y1) attacks (x2, y2)

};

// Файл main.cpp

#include <iostream>

#include "Queen.hpp"

using namespace std;

int main() {

cout << "Start" << endl;

Queen queen;

queen.show();

if (queen.solve(250)) { // try 100/250/500/1000/2000

cout << "Solved." << endl;

queen.show();

} else {

cout << "Failed. Try more iterations." << endl;

}

return 0;

}

// Файл Queen.cpp

#include "Queen.hpp"

#include <stdlib.h>

#include <fstream>

#include <iostream>

// constructor

Queen::Queen() {

srand(time(NULL));

// dynamically allocate memory

desk = new int\*[N];

for (int i = 0; i < N; i++)

desk[i] = new int[N];

// initialize desk

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

desk[i][j] = 0;

}

}

// initialize queen position

int i = N;

while (i != 0) {

int x = rand()%N, y = rand()%N;

if (desk[x][y] == 1) continue;

desk[x][y] = 1;

i--;

}

}

Queen::~Queen() {

for (int i = 0; i < N; ++i)

delete [] desk[i];

delete [] desk;

}

// helper functions

void Queen::save() {

// save to file;

std::ofstream f;

f.open("output.txt", std::ios::app);

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

f << desk[i][j] << " ";

}

f << "\n";

}

f << "\n";

f.close();

}

void Queen::show() {

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++) {

std::cout << desk[i][j] << " ";

}

std::cout << "\n";

}

}

void Queen::info() {

std::cout << "Мокроусов Владислав ІП-71 ФІОТ" << std::endl;

}

int Queen::argmin(int arr[]) {

int min = arr[0], j = 0;

for (int i = 0; i < N; i++) if (arr[i] < min) min = arr[i];

for (int i = 0; i < N; i++) if (arr[i] == min) j++;

int arrRand[j], k = 0;

for (int i = 0; i < N; i++) {

if (arr[i] == min) {

arrRand[k] = i;

k++;

}

}

return arrRand[rand() % j];

}

// arrange functions (by row/col)

void Queen::setRow() {

// set one queen for each row

int fRows[N] = {0}, qpos[N][2];

freeRows(fRows, qpos);

// iterate rows with more than 1 piece

for (int i = 0; i < N; i++) {

if (fRows[i] != 2) continue;

int j = 0;

while (fRows[i] != 1) {

if (qpos[j][0] == i) {

for (int k = 0; k < N; k++)

if (fRows[k] == 0 && moveRow(k, i, qpos[j][1])) break;

}

freeRows(fRows, qpos);

if (fRows[i] == 1) break;

j++;

if (j == N) j = 0;

}

}

}

void Queen::setCol() {

// set one queen for each row

int fCols[N] = {0}, qpos[N][2];

freeCols(fCols, qpos);

// iterate cols with more than 1 piece

for (int i = 0; i < N; i++) {

if (fCols[i] != 2) continue;

int j = 0;

while (fCols[i] != 1) {

if (qpos[j][1] == i) {

for (int k = 0; k < N; k++)

if (fCols[k] == 0 && moveCol(k, qpos[j][0], i)) break;

}

freeCols(fCols, qpos);

if (fCols[i] == 1) break;

j++;

if (j == N) j = 0;

}

}

}

// get free rows/cols

void Queen::freeRows(int rows[], int qpos[N][2]) {

int k = 0, m; // free rows; k,m - counters

for (int i = 0; i < N; i++) {

m = 0; // nums pieces in row

for (int j = 0; j < N; j++) {

if (desk[i][j] == 1) {

qpos[k][0] = i; // row

qpos[k][1] = j; // col

m++;

k++;

}

}

switch (m) {

case 0:

rows[i] = 0; // row is clear

break;

case 1:

rows[i] = 1; // 1 piece in a row

break;

default:

rows[i] = 2;

break;

}

}

}

void Queen::freeCols(int cols[], int qpos[N][2]) {

int k = 0, m; // free cols; k,m - counters

for (int i = 0; i < N; i++) {

m = 0;

for (int j = 0; j < N; j++) {

if (desk[j][i] == 1) {

qpos[k][0] = j; // row

qpos[k][1] = i; // col

m++;

k++;

}

}

switch (m) {

case 0:

cols[i] = 0; // col is clear

break;

case 1:

cols[i] = 1; // 1 piece in a col

break;

default:

cols[i] = 2;

break;

}

}

}

// main algorithm

int Queen::conflicts(int x1, int y1, int qrow) {

// return number of conflicts at position (x1, y1), queen at (qrow, y1)

int conf = 0;

int fRows[N] = {0}, qpos[N][2];

freeRows(fRows, qpos);

desk[qrow][y1] = 0;

for (int i = 0; i < N; i++) {

if (qpos[i][1] == y1) continue; // skip queen at that col

if (attacks(qpos[i][0], qpos[i][1], x1, y1)) conf++;

}

desk[qrow][y1] = 1;

return conf;

}

bool Queen::solve(int iter = 100) {

this->setRow();

this->setCol();

int fRows[N] = {0}, qpos[N][2], x, y, rcol;

freeRows(fRows, qpos); // get queen positions

for (int i = 0; i < iter; i++) {

rcol = rand()%N; // random col

int colConflicts[N];

// get queen coords in random col

for (int j = 0; j < N; j++) {

if (qpos[j][1] == rcol) {

x = qpos[j][0];

break;

}

}

for (int j = 0; j < N; j++)

colConflicts[j] = conflicts(j, rcol, x);

int min = argmin(colConflicts);

if (x != min)

move(x, rcol, min, rcol);

freeRows(fRows, qpos); // get new queen positions

if (solved(qpos)) return true; // check if solved

}

return false;

}

bool Queen::solved(int qpos[N][2]) {

for (int i = 0; i < N; i++)

if (conflicts(qpos[i][0], qpos[i][1], qpos[i][0]) != 0) return false;

return true;

}

// attack function

bool Queen::attacks(int x, int y, int xpos, int ypos) {

int n = x, m = y - 1;

// left

if (y >= 1) {

while (m >= 0) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

m--;

}

if (x >= 1) {

// top-left diagonal

for (n = x - 1, m = y - 1; m >= 0 && n >= 0; n--, m--) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

}

}

if (x <= 6) {

// bottom-left diagonal

for (n = x + 1, m = y - 1; m >= 0 && n < N; n++, m--) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

}

}

}

// right

if (y <= 6) {

n = x, m = y + 1;

while (m < N) {

//std::cout << n << " " << m << std::endl;

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

m++;

}

if (x >= 1) {

// top-right diagonal

for (n = x - 1, m = y + 1; m < N && n >= 0; n--, m++) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

}

}

if (x <= 6) {

// bottom-right diagonal

for (n = x + 1, m = y + 1; m < N && n < N; n++, m++) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

}

}

}

if (x <= 6) {

// bottom

n = x + 1, m = y;

while (n < N) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

n++;

}

}

if (x >= 1) {

// top

n = x - 1, m = y;

while (n >= 0) {

if (desk[n][m] == 1) break;

if (n == xpos && m == ypos) return true;

n--;

}

}

return false;

}

// move functions

void Queen::move(int x1, int y1, int x2, int y2) {

desk[x1][y1] = 0;

desk[x2][y2] = 1;

save();

}

bool Queen::moveRow(int k, int i, int j) {

// k = row, (i, j) - position

int m, n; // counters

if (k == i) {

return true;

} else if (k < i) {

n = i - 1, m = j;

// top

while (n >= 0) {

if (desk[n][m] == 1) break;

if (n == k) {

move(i, j, n, m);

return true;

}

n--;

}

// top-left diagonal

for (n = i - 1, m = j - 1; m >= 0 && n >= 0; n--, m--) {

if (n == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

// top-right diagonal

for (n = i - 1, m = j + 1; m < N && n >= 0; n--, m++) {

if (n == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

} else {

n = i + 1, m = j;

// bottom

while (n < N) {

if (desk[n][m] == 1) break;

if (n == k) {

move(i, j, n, m);

return true;

}

n++;

}

// bottom-left diagonal

for (n = i + 1, m = j - 1; m >= 0 && n < N; n++, m--) {

if (n == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

// bottom-right diagonal

for (n = i + 1, m = j + 1; m < N && n < N; n++, m++) {

if (n == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

}

return false;

}

bool Queen::moveCol(int k, int i, int j) {

// k = col, (i, j) - position

int m, n; // counters

if (k == j) {

return true;

} else if (k < j) {

n = i, m = j - 1;

// left

while (m >= 0) {

if (desk[n][m] == 1) break;

if (m == k) {

move(i, j, n, m);

return true;

}

m--;

}

// top-left diagonal

for (n = i - 1, m = j - 1; m >= 0 && n >= 0; n--, m--) {

if (m == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

// bottom-left diagonal

for (n = i + 1, m = j - 1; m >= 0 && n < N; n++, m--) {

if (m == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

} else {

n = i, m = j + 1;

// right

while (m < N) {

if (desk[n][m] == 1) break;

if (m == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

m++;

}

// top-right diagonal

for (n = i - 1, m = j + 1; m < N && n >= 0; n--, m++) {

if (m == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

// bottom-right diagonal

for (n = i + 1, m = j + 1; m < N && n < N; n++, m++) {

if (m == k && desk[n][m] == 0) {

move(i, j, n, m);

return true;

}

}

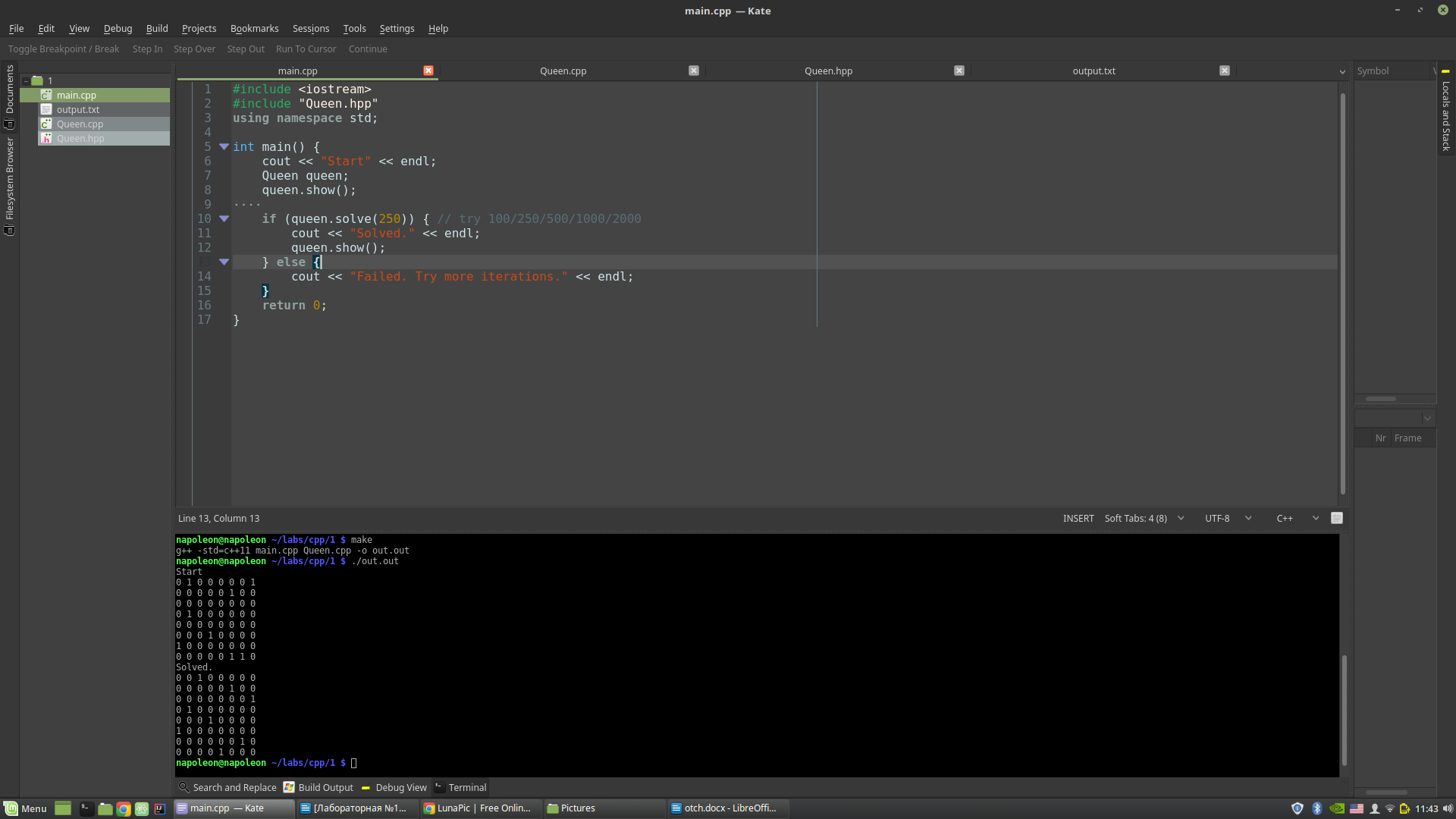
}

return false;

}

# 6 ПРИКЛАД РОБОТИ ПРОГРАМИ

Для запуску програми потрібно ввести команду “make” та запустити її “./out.out”. Нижче представлено скріншот роботи програми.



# 7 ВИСНОВОК

Під час роботи на лабораторною було створено та реалізовано алгоритм для вирішення задачі про розміщення ферзів на дошці. Слід зауважити, що реалізований алгоритм працює для випадку N ферзів на дошці розміром (NxN), хоча робота програми буде довшою. При реалізації був застосований об’єктно-орієнтований підхід.

У результаті отримано ефективну програму для вирішення заданої задачі.