**ДОДАТОК 2**

Код програми

НТУУ «КПІ ім. Ігоря Сікорського» ТЕФ АПЕПС ТІ-01

Листів 12

Київ – 2021

**main.cpp**

#include "chess.h"

class Chess {//основной класс игры

private:

enum Figures {//перечисление фигур, гранц, пустой клетки

EMPTY,

PAWN, QUEEN, KING,

KNIGHT, BISHOP, ROOK,

BARRIER

};

enum Angles {//перечисление углов направления

ANGLE0,

ANGLE45,

ANGLE90,

ANGLE135

};//enumeration векторов направления

typedef struct Cell {//структура, хранящая данные о состоянии клетки

char figure;//переменная для хранения значения фигуры

char mobility;//направление передвижения

char color;//цвет клетки

char attacked;//индикатор атакованых клеток

char special;//ходы под связкой

} cell\_t;

cell\_t \*\*desk;

char \*\*move\_table;//доска ходов

int moveptr\_list[6] = { 0, 0, 0, 0, 0, 0 };//масив размеров ходов

struct Move {//структура хранящая координаты текущего хода

int fromX, fromY, toX, toY, figure;

} current\_move;

int side;//сторона игрока

int castling\_active[2][3];//масив проверки рокировок

struct Extra {

int exist, X, Y;

} extra;//структура для взятия на проходе

int atacks\_on\_king = 0;

typedef void (Chess::\*line\_handler\_t)(int, int, int, int);

typedef void(Chess::\*cell\_handler\_t)(int, int);

//установка начальных позиций доски, состояний рокировок и первого хода

const char \*starting =

"rgbqkbgr"//белые фигуры

"pppppppp"

"........"

"........"

"........"

"........"

"PPPPPPPP"

"RGBQKBGR"//чёрные фигуры

"000000e8e8k";//состояние рокировок, ход по умолчанию

vector<tuple<int, int, int, int>> moves;//вектор из кортежей

int white\_moves = 0, black\_moves = 0;//счётчики шагов игроков

clock\_t start;//начало таймера игри

public:

void clear\_desk() {//очистка доски

for (int i = 1; i <= 8; ++i) {

for (int j = 1; j <= 8; ++j) {

desk[i][j].figure = EMPTY; //во всех клетках установка пустого состояния

}

}

}

void clear\_desk\_state() {//состояние доски анулируется

for (int i = 1; i <= 8; ++i) {

for (int j = 1; j <= 8; ++j) {

desk[i][j].attacked = 0;//нет атак на короля

desk[i][j].special = 0;//нет специальных ходов

}

}

}

void init\_desk() {//инициализация доски

desk = new cell\_t\* [10];

for (int i = 0; i < 10; ++i) {

desk[i] = new cell\_t [10];

}

for (int i = 0; i < 10; ++i) {//назначение границ доски

desk[i][0].figure = BARRIER;

desk[i][9].figure = BARRIER;

desk[0][i].figure = BARRIER;

desk[9][i].figure = BARRIER;

}

clear\_desk();//очистка доски

clear\_desk\_state();//очистка состояния доски

}

void clear\_move\_table() {//очистка строк ходов

for (int i = 0; i < 6; ++i) {

moveptr\_list[i] = 0;

}

}

void init\_move\_table() {//инициализация доски ходов

move\_table = new char\* [6];//выделение памяти под 6 строк

for (int i = 0; i < 6; ++i)

move\_table[i] = new char[1024];//выделение памяти под масив с ходами

clear\_move\_table();//очистка доски ходов

}

void destroy\_move\_table() {//очистить доски ходов

for (int i = 0; i < 6; ++i) {

delete move\_table[i];//очистка масива с ходами

}

delete move\_table;

}

void add\_move(int fromX, int fromY, int toX, int toY) {//функция добавления ходов

char \*move\_list = move\_table[desk[fromX][fromY].figure - 1];//запись ходов в список

int moveptr = moveptr\_list[desk[fromX][fromY].figure - 1];

move\_list[moveptr++] = fromX + 'a' - 1;//запись в список ходов с конвертацией в char

move\_list[moveptr++] = fromY + '0';

move\_list[moveptr++] = toX + 'a' - 1;

move\_list[moveptr++] = toY + '0';

moveptr\_list[desk[fromX][fromY].figure - 1] = moveptr;

}

//установка текущего хода

void set\_current\_move(int fromX, int fromY, int toX, int toY, int figure) {

current\_move.fromX = fromX;

current\_move.fromY = fromY;

current\_move.toX = toX;

current\_move.toY = toY;

current\_move.figure = figure;

}

void release\_current\_move() {//осуществление выбраного хода

int fromX = current\_move.fromX,//запись координат текущей позиции в переменые

fromY = current\_move.fromY,

toX = current\_move.toX,

toY = current\_move.toY,

figure = current\_move.figure;//запись текущей фигуры

side = !desk[fromX][fromY].color;//переход на другую сторону

if (fromX == toX && fromY == toY) {//если выбран ход на текущую позицию

return;

}

update\_castling();//рокировка

extra\_check();//взятие на проходе

if (desk[fromX][fromY].figure == KING && abs(fromX - toX) > 1) {//при рокировке

int \_fromX = fromX < toX ? 8 : 1;//опредиление рокировки

int \_toX = (fromX + toX) / 2;

desk[\_toX][toY] = desk[\_fromX][fromY];//передвижение ладьи

desk[\_fromX][fromY].figure = EMPTY;

}

if (desk[fromX][fromY].figure == PAWN) {//если ходит пешка

if (toY == (side ? 1 : 8)) {

desk[fromX][fromY].figure = figure - 1;

}

if (fromX != toX && desk[toX][toY].figure == EMPTY) {//замена при взятии на проходе

desk[toX][fromY].figure = EMPTY;

}

}

desk[toX][toY] = desk[fromX][fromY];//передвижение

desk[fromX][fromY].figure = EMPTY;//клетка, с которой походила становиться пустой

side ? (++white\_moves) : (++black\_moves);

}

void update\_castling() {//рокировка

int fromX = current\_move.fromX,//запись координат текущей позиции в переменные

fromY = current\_move.fromY,

toX = current\_move.toX,

toY = current\_move.toY;

for (int i = 1; i <= 8; i += 7) {//проход по 1 и 8 ряду

for (int j = 1; j <= 8; j += 3 + (j == 1)) {

castling\_active[i < 8][j / 4] |= (fromX == j && fromY == i);//

castling\_active[i < 8][j / 4] |= (toX == j && toY == i);

}

}

}

int extra\_figure(const cell\_t \*cell) {

if (cell->figure == EMPTY) {

return 0;

}//проверяем дальше

if ((cell->color == side && cell->figure == KING) ||

(cell->color != side && (cell->figure == ROOK || cell->figure == QUEEN))) {

return cell->figure;

} else {

return -1;

}

}

//проверка можно ли осуществить взятие на проходе

void extra\_check() {

int fromX = current\_move.fromX,//запись координат

fromY = current\_move.fromY,

toX = current\_move.toX,

toY = current\_move.toY;

extra.exist = 0;

//проверка правильности хода

if (desk[fromX][fromY].figure != PAWN || abs(fromY - toY) < 2) {

return;

}

int X, Y;//координаты пешки, которая может осуществить взятие на проходе

//проверка влево на наличие вражеской пешки(ход белых - белой), которая может взять на проходе

if (desk[toX - 1][toY].figure == PAWN && desk[toX - 1][toY].color == side) {

X = toX - 1;

Y = toY;

//проверка вправо на наличие вражеской пешки, которая может взять на проходе

} else if (desk[toX + 1][toY].figure == PAWN && desk[toX + 1][toY].color == side) {

X = toX + 1;

Y = toY;

} else {

return;

}

extra.exist = 1;//взятие на проходе может быть осуществлено

extra.X = toX;//координаты на которые станет пешка при взятии

extra.Y = toY + (side ? 1 : -1);//опредиление направления хода

int dec = 0, inc = 0;

for (int i = X - 1; i > 0; --i) {

int t = extra\_figure(&desk[i][toY]);

//если t = 0 проверяем дальше

if (t > 0) {

dec = t;

break;

} else if (t < 0) {

return;

}

}

for (int i = X + 1; i <= 8; ++i) {

int t = extra\_figure(&desk[i][toY]);

if(t > 0) {//если есть фигура

inc = t;//запоминание фигуры

break;

} else if (t < 0) {

return;

}

}

//если стоит король, то невозможно осуществить из-за шаха

if (inc && dec && (dec == KING || inc == KING)) {

extra.exist = 0;

}

}

char get\_figure\_symbol(int figure) {//конвертация enum в char

return (figure == PAWN) \* 'p' + (figure == QUEEN) \* 'q' + (figure == KING) \* 'k'

+ (figure == KNIGHT) \* 'g' + (figure == BISHOP) \* 'b' + (figure == ROOK) \* 'r'

+ (figure == EMPTY) \* '.';

}

int get\_figure\_number(char figure) {//конвертация char в enum

figure = tolower(figure);

return (figure == 'p') \* PAWN + (figure == 'q') \* QUEEN + (figure == 'k') \* KING

+ (figure == 'g') \* KNIGHT + (figure == 'b') \* BISHOP + (figure == 'r') \* ROOK

+ (figure == '.') \* EMPTY;

}

void unpack\_desk(const char \*position) {//распаковка доски

init\_desk();

for (int i = 0; i < 8; ++i) {

for (int j = 0; j < 8; ++j) {

const char c = position[i \* 8 + j];

desk[j + 1][i + 1] = (cell\_t) {

static\_cast<char>(get\_figure\_number(c)), 0, (c >= 'a' && c <= 'z'), 0, 0

};

}

}

}

//задание начального состояни рокировок

void unpack\_castling(const char \*position) {

for (int i = 0; i < 2; ++i) {

for (int j = 0; j < 3; ++j) {

castling\_active[i][j] = position[i \* 3 + j] - '0';

}

}

}

void unpack\_move(const char \*position) {//установка первого хода

set\_current\_move(position[0] - 'a' + 1, position[1] - '0',

position[2] - 'a' + 1, position[3] - '0',

get\_figure\_number(position[4]));

}

//установка начальных позиций доски

void unpack\_position(const char \*position) {

unpack\_desk(position);

unpack\_move(position + 70);

unpack\_castling(position + 64);

}

int get\_mobility(int dx, int dy) {//определяем направление хода

return abs(dx) == abs(dy) ? (dx == dy ? ANGLE45 : ANGLE135) : (dx ? ANGLE0 : ANGLE90);

}

int is\_check() {//проверка на шах

return atacks\_on\_king == 1;

}

int is\_multicheck() {//форк

return atacks\_on\_king > 1;

}

//проверка на выход за границы доски

int is\_correct(int x, int y) {

return x > 0 && x < 9 && y > 0 && y < 9;

}

//проверка существуует ли фугура

int is\_figure(const cell\_t \*cell) {

return cell->figure != EMPTY && cell->figure != BARRIER;

}

//есть ли союзной фигурой

int is\_ally(const cell\_t \*cell) {

return is\_figure(cell) && cell->color == side;

}

//вражеская фигура

int is\_enemy(const cell\_t \*cell) {

return is\_figure(cell) && cell->color != side;//фигуры которая ходит

}

//проверка на правильность направления ходаесли фигура на клетке не король

int is\_direction\_correct(const cell\_t \*cell, int dx, int dy) {

return (cell->figure != KNIGHT && get\_mobility(dx, dy) == cell->mobility) || !cell->special;

}

//проверка на правильность хода, можно ли походить

int is\_moveable(int x, int y, int dx, int dy) {

if (!is\_correct(x + dx, y + dy) || is\_ally(&desk[x + dx][y + dy]) || !is\_direction\_correct(&desk[x][y], dx, dy))

return 0;

return !is\_check() || desk[x + dx][y + dy].special;

}

//генерация всех ходов, проверка каждой клетки, на которую может походить вражеская фигура

void atack\_handler(int x, int y, int dx, int dy) {

if (is\_correct(x + dx, y + dy)) {

cell\_t \*cell = &desk[x + dx][y + dy];

cell->attacked = 1;//клетка на которой вражеская фигура может быть атакована

if (is\_ally(cell) && cell->figure == KING) {//король атакован

desk[x][y].special = 1;

++atacks\_on\_king;

}

}

}

//блокировка ходов для клеток под связкой

void pierce\_block(int x, int y, int dx, int dy) {

desk[x][y].special = 1;

desk[x][y].mobility = get\_mobility(dx, dy);//вектор,в котором фигура под связкой

}

//генерация ходов, которые могут быть побиты для линии союзных фигур и клеток под их прикрытием

void ally\_line\_handler(int x, int y, int dx, int dy) {

if(!is\_direction\_correct(&desk[x][y], dx, dy))

return;

for(int i = dx, j = dy; is\_correct(x + i, y + j); i += dx, j += dy) {

if(is\_moveable(x, y, i, j)) {

add\_move(x, y, x + i, y + j);//добавить ход

}

if(is\_figure(&desk[x + i][y + j])) {

break;

}

}

}

//функция блокировки невозможных ходов

int pierce\_detector(int x, int y, int dx, int dy, int cnt) {

cell\_t \*cell = &desk[x + dx][y + dy];

if (!cnt) {

atack\_handler(x, y, dx, dy);

}

//если клетка пустая продолжаем проверять в этом направлении

if (cell->figure == EMPTY) {

return cnt;//сnt = 0

}

++cnt;

//если atack\_handler обнаружил шах

cnt \*= (cell->figure == KING ? -1 : 1);

if (cell->figure == BARRIER || is\_enemy(cell) || cnt == 2) {

return -3;

} else { //могут взаимно побить друг друга

return cnt;

}

}

//обработчик возможных ходов вражеских фигур

void enemy\_line\_handler(int x, int y, int dx, int dy) {

int cnt = 0, i = 0, j = 0;

do {

i += dx;

j += dy;;

cnt = pierce\_detector(x, y, i, j, cnt);

} while (cnt >= 0);

if (cnt == -1) {//при шахе

atack\_handler(x, y, i + dx, j + dy);

//проверка на вражеского короля

for (i = x, j = y; desk[i][j].figure != KING; i += dx, j += dy) {

desk[i][j].special = 1;

}

} else if (cnt == -2) {//двойная связка

for (i = x, j = y; !is\_ally(&desk[i][j]); i += dx, j += dy);

pierce\_block(i, j, dx, dy);

}

}

//направление по горизонтали/вертикали

void X\_handler(int x, int y, int vh, int d, line\_handler\_t handl) {

if (vh) {//если ход по горизонтали или вертикали

(this->\*handl)(x, y, 1, 0);

(this->\*handl)(x, y, -1, 0);

(this->\*handl)(x, y, 0, 1);

(this->\*handl)(x, y, 0, -1);

}

if (d) {//если ход по горизонтали

(this->\*handl)(x, y, 1, 1);

(this->\*handl)(x, y, 1, -1);

(this->\*handl)(x, y, -1, 1);

(this->\*handl)(x, y, -1, -1);

}

}

//генерация ходов союзного коня

void ally\_knight\_handler(int x, int y) {

for(int i = 1; i <= 2; ++i) {

int j = 3 - i;

if(is\_moveable(x, y, i, j)) {

add\_move(x, y, x + i, y + j);

}

if(is\_moveable(x, y, i, -j)) {

add\_move(x, y, x + i, y - j);

}

if(is\_moveable(x, y, -i, j)) {

add\_move(x, y, x - i, y + j);

}

if(is\_moveable(x, y, -i, -j)) {

add\_move(x, y, x - i, y - j);

}

}

}

void enemy\_knight\_handler(int x, int y) {//обработчик вражеского короля

for (int i = 1; i <= 2; ++i) {

int j = 3 - i;

atack\_handler(x, y, i, j);

atack\_handler(x, y, i, -j);

atack\_handler(x, y, -i, j);

atack\_handler(x, y, -i, -j);

}

}

//генерация ходов для союзного короля

void ally\_king\_handler(int x, int y) {

for (int i = -1; i <= 1; ++i) {

for (int j = -1; j <= 1; ++j) {

if (is\_correct(x + i, y + j) && !desk[x + i][y + j].attacked &&

!is\_ally(&desk[x + i][y + j])) {

add\_move(x, y, x + i, y + j);

}

}

}

}

//генерация возможных ходов для вражеского короля

void enemy\_king\_handler(int x, int y) {

for (int i = -1; i <= 1; ++i) {

for (int j = -1; j <= 1; ++j) {

if (desk[x + i][y + j].figure != KING) {

atack\_handler(x, y, i, j);

}

}

}

}

//генератор ходов, при которых пешка может побить вражескую фигуру

void pawn\_kill\_move(int x, int y, int dx, int dy) {

if ((is\_enemy(&desk[x + dx][y + dy]) && is\_moveable(x, y, dx, dy))

|| (extra.exist && x + dx == extra.X && y + dy == extra.Y

&& (is\_moveable(x, y, dx, dy) || is\_moveable(x, y, dx, 0)))) {

add\_move(x, y, x + dx, y + dy);

}

}

void ally\_pawn\_handler(int x, int y) {//генерация ходов для союзной пешки

int t = (side ? 1 : -1);

//t определяет направление

pawn\_kill\_move(x, y, 1, t);

pawn\_kill\_move(x, y, -1, t);

if (is\_figure(&desk[x][y + t])) {

return;

}

//генерация хода на одну клетку вверх/вниз

if (is\_moveable(x, y, 0, t)) {

add\_move(x, y, x, y + t);

}

//генерация хода на 2 клетки вверх/вниз для первого хода пешки

if (y == 7 - 5 \* side && !is\_figure(&desk[x][y + 2 \* t]) && is\_moveable(x, y, 0, 2 \* t)) {

add\_move(x, y, x, y + 2 \* t);

}

}

//генерация ходов, которые может сделать вражеская пешка, клеток, которые она может побить

void enemy\_pawn\_handler(int x, int y) {

atack\_handler(x, y, 1, -(side ? 1 : -1));

atack\_handler(x, y, -1, -(side ? 1 : -1));

}

void ally\_rook\_handler(int x, int y) {//проверка союзной ладьи

X\_handler(x, y, 1, 0, ally\_line\_handler);

}

void enemy\_rook\_handler(int x, int y) {//проверка вражеской ладьи

X\_handler(x, y, 1, 0, enemy\_line\_handler);

}

void ally\_bishop\_handler(int x, int y) {//проверка союзного слона

X\_handler(x, y, 0, 1, ally\_line\_handler);

}

void enemy\_bishop\_handler(int x, int y) {//проверка вражеского слона

X\_handler(x, y, 0, 1, enemy\_line\_handler);

}

void ally\_queen\_handler(int x, int y) {//проверка союзной королевы

X\_handler(x, y, 1, 1, ally\_line\_handler);

}

void enemy\_queen\_handler(int x, int y) {//проверка вражеской королевы

X\_handler(x, y, 1, 1, enemy\_line\_handler);

}

//поиск фигуры и опредиление генератора ходов для фигур

void ally\_figure\_handler(int x, int y) {

//статичиский массив с функциями генерации ходов фигур

const static cell\_handler\_t handlers[] = {

ally\_pawn\_handler, ally\_queen\_handler,

ally\_king\_handler, ally\_knight\_handler,

ally\_bishop\_handler, ally\_rook\_handler

};

if(!is\_figure(&desk[x][y]) || is\_enemy(&desk[x][y])

|| (is\_multicheck() && desk[x][y].figure != KING)) {//

return;

}

(this->\*handlers[desk[x][y].figure - 1])(x, y);

}

void enemy\_figure\_handler(int x, int y) {//функция вражеских обработчиков

const static cell\_handler\_t handlers[] = {

enemy\_pawn\_handler, enemy\_queen\_handler,

enemy\_king\_handler, enemy\_knight\_handler,

enemy\_bishop\_handler, enemy\_rook\_handler

};

if (!is\_figure(&desk[x][y]) || is\_ally(&desk[x][y])) {

return;

}

(this->\*handlers[desk[x][y].figure - 1])(x, y);

}

void desk\_handler(cell\_handler\_t handl) {//обработчик доски

for (int i = 1; i <= 8; ++i) {

for (int j = 1; j <= 8; ++j) {

(this->\*handl)(i, j);

}

}

}

void castle\_handler() {//обработчик рокировок

if (is\_check() || is\_multicheck() || castling\_active[side][1]) {

return;

}

int sideline = 8 - side \* 7;

//длинная рокировка

if (!castling\_active[side][0]

&& !desk[4][sideline].attacked && !is\_figure(&desk[4][sideline])

&& !desk[3][sideline].attacked && !is\_figure(&desk[3][sideline])) {

add\_move(5, sideline, 3, sideline);

}

//короткая рокировка

if (!castling\_active[side][2] && !desk[6][sideline].attacked && !is\_figure(&desk[6][sideline])

&& !desk[7][sideline].attacked && !is\_figure(&desk[7][sideline])) {

add\_move(5, sideline, 7, sideline);

}

}

void position\_handler() {//обработчик состояний

atacks\_on\_king = 0;

clear\_move\_table();

clear\_desk\_state();

release\_current\_move();

desk\_handler(enemy\_figure\_handler);

desk\_handler(ally\_figure\_handler);

castle\_handler();

}

void UpdateMoves() {

moves.clear();

for (int i = 0; i < 6; ++i) {

for (int j = 0; j < moveptr\_list[i]; j += 4) {

moves.push\_back(make\_tuple(

move\_table[i][j] - 'a' + 1,

move\_table[i][j + 1] - '0',

move\_table[i][j + 2] - 'a' + 1,

move\_table[i][j + 3] - '0'

));

}

}

destroy\_move\_table();

init\_move\_table();

}

void start\_timer() {//начало отсчета времени

start = clock();

}

void game\_info() {

double time = (clock() - start) / CLOCKS\_PER\_SEC / 60.;

cout << "\n\t\t\t\t\t" << (side ? "Black won\n" : "White won\n");

cout << "\t\t\tTime of the game: " << (int)time << " minutes, " << setprecision(2)<< (time - (int)time)\*100 << " seconds\n";

cout << "\t\t\t\tTotal moves: " << white\_moves + black\_moves << endl;

cout << "\t\t\tWhite\_moves -- " << white\_moves << ", black\_moves -- " << black\_moves << endl;

}

void output\_desk() {//функция вывода доски

system("cls");

cout << "\n\t\t\t\t Chess game\n\n";

cout << "\t\t\t ----------------------------------------" << endl;

for (int i = 8; i >= 1; --i) {

cout << "\t\t\t" << i << " | ";

for (int j = 1; j <= 8; ++j) {

cout << static\_cast<char>(get\_figure\_symbol(desk[j][i].figure) +

(desk[j][i].color || desk[j][i].figure == EMPTY ? 0 : 'A' - 'a'));

if (j < 8)

cout << " || ";

}

cout << " |\n\t\t\t ----------------------------------------\n";

}

cout << " \t\t\t a b c d e f g h \n\n";

}

Chess() {

init\_move\_table();

unpack\_position(starting);

position\_handler();

UpdateMoves();

}

bool Turn(int number) {//функция осуществления хода

//проверка выбора правильного хода, вызов обработчика

if (number < 0 || number >= moves.size()) {//проверка корректности введеного номера

return false;

}

//распаковка, возвращение значений кортежа, лежащего за номером в векторе и запись в значения ходов

tie(current\_move.fromX, current\_move.fromY, current\_move.toX, current\_move.toY) = moves[number];

position\_handler();//генерация для измененной доски и позиций

UpdateMoves();

return true;

}

void Play() {//функция начала игры, вывод доски ивозможных ходов

start\_timer();

while (!moves.empty()) {//пока не будут выведены все хвозможные ходы

output\_desk();//вывод доски

sort(moves.begin(), moves.end());

cout << setw(52) << (side ? "White's turn\n" : "Black's turn\n");

cout << setw(54) << "Possible moves\n\n";

for (int i = 0; i < moves.size(); ++i) {

if (i % 2 == 0) cout << "\t\t\t\t";

cout << setw(2) << i + 1 << ". " << char(get<0>(moves[i]) + 96) << get<1>(moves[i]) << "->"

<< char(get<2>(moves[i]) + 96) << get<3>(moves[i]) << " ";

if(i % 2 != 0) cout << endl;

}

cout << "\n\t\t\t\t\t ";

int number;

try {

cin >> number;

if (cin) {

Turn(number - 1);

} else {

throw -1;

}

} catch(int n) {

game\_info();

cout << "\n\n\t\t\t\tThank you for the game!\n\t\t\t\tHope to see you later.\n\n";

exit(0);

}

}

game\_info();

}

void Setup() {//меню игры

string answer;//ответ пользователя

cout << "\n\t\t\t\t\tWelcome to Chess game!\n\n";

cout << setw(64) << "Choose option and enter:\n";

cout << setw(69) << "<play>, <rules>, <exit>\n\t\t\t\t\t\t";

cin >> answer;

if (answer == "play") {

system("cls");

Play();

} else if(answer == "rules") {

system("cls");

Rules();

} else if (answer == "exit") {

system("cls");

cout << "\n\n\t\t\t\tThank you for the game!\n\t\t\t\tHope to see you later.\n\n";

exit(0);

} else {

cout << "\n\n\t\t\t\tYour answer is incorrect.\n";

}

}

};

//дополнительный клас

class ChessGame {

public:

void StartGame() {//зацикливание программы

while(true) {

Chess game;

game.Setup();

}

}

};

int main() {

ChessGame game;

game.StartGame();

return 0;

}