Оглавление

[Введение 1](#_Toc91011148)

[Глава 1 2](#_Toc91011149)

[**1.1.** **Теория игр** 3](#_Toc91011150)

[**1.2.** **Антагонистическая игра** 3](#_Toc91011151)

[**1.3.** **Алгоритм Альфа-бета отсечение** 4](#_Toc91011152)

[**1.4.** **Интерфейс** 6](#_Toc91011153)

[Глава 2. Реализация 7](#_Toc91011154)

[**2.1. Input System и ViewPort** 7](#_Toc91011155)

[**2.2. Перечисления** 8](#_Toc91011156)

[**2.3. Класс Game** 9](#_Toc91011157)

[**2.4. Функция Game::Logic()** 9](#_Toc91011158)

[**2.5. Сущность AI** 11](#_Toc91011159)

[**2.6. AI::GetBestMove()** 11](#_Toc91011160)

[**2.7. AI::GetMove()** 12](#_Toc91011161)

[Глава 3. Оптимизация алгоритмов 13](#_Toc91011162)

[3.1. Оптимизация циклов 13](#_Toc91011163)

[3.2 Оптимизация вызовов функций 14](#_Toc91011164)

[Заключение 15](#_Toc91011165)

[Cписок используемых источников 16](#_Toc91011166)

[Приложение 17](#_Toc91011167)

# Введение

Целью курсовой работы является моделирование игры «Крестики-нолики». Для реализации был выбран язык программирования С++ в среде MS Visual Studio 2019. Вкратце, в данной курсовой работе будет реализована полноценная игра. Также игроку будет доступен выбор размеров игрового поля.

# Глава 1

## **Теория игр**

**Теория игр** — это раздел математической экономики, изучающий решение конфликтов между игроками и оптимальность их стратегий. Конфликт может относиться к разным областям человеческого интереса: чаще всего это экономика, социология, политология, реже биология, кибернетика и даже военное дело.

**Конфликтом** является любая ситуация, в которой затронуты интересу двух и более участников, традиционно называемых игроками. Для каждого игрока существует определенный набор стратегий, которые он может применить. Пересекаясь, стратегии нескольких игроков создают определенную ситуацию, в которой каждый игрок получает определенный результат, называемый выигрышем, положительным или отрицательным.

Оптимальные решения или стратегии в математическом моделировании предлагались ещё в XVIII в. Задачи производства и ценообразования в условиях олигополии, которые стали позже хрестоматийными примерами теории игр, рассматривались в XIX в. А. Курно и Ж. Бертраном. В начале XX в. Эмануил Ласкер, Эрнст Цермело и Эмиль Борель выдвигают идею математической теории конфликта интересов.

Математическая теория игр берёт своё начало из неоклассической экономики. Впервые математические аспекты и приложения теории были изложены в классической книге 1944 года Джона фон Неймана и Оскара Моргенштерна «Теория игр и экономическое поведение».

## **Антагонистическая игра**

**Антагонистическая игра** — термин теории игр. **Нулевая сумма** — это ситуация в теории игр, в которой выигрыш одного игрока эквивалентен проигрышу другого, поэтому чистое изменение богатства или выгоды равно нулю.

В игре с нулевой суммой может участвовать как минимум два игрока, так и миллионы участников. На финансовых рынках опционы и фьючерсы являются примерами игр с нулевой суммой, за исключением транзакционных издержек. На каждого человека, который выигрывает по контракту, есть контрагент, который проигрывает.

Игры с нулевой суммой можно найти в теории игр, но они менее распространены, чем игры с ненулевой суммой. Покер и азартные игры являются популярными примерами игр с нулевой суммой, поскольку сумма выигрышей одних игроков равна сумме проигрышей других. Такие игры, как шахматы и теннис, где есть один победитель и один проигравший, также являются играми с нулевой суммой.

Игры с нулевой суммой — это противоположность беспроигрышным ситуациям — таким как торговое соглашение, которое значительно увеличивает торговлю между двумя странами — или проигрышным ситуациям, таким как война, например.

**Игра Пенни** — это игра с нулевой суммой, потому что выигрыш одного игрока — это проигрыш другого. Результаты для игроков A и B показаны на рисунке 1. Как видно, объединенный результат игры для X и Y во всех четырех ситуациях равен нулю.

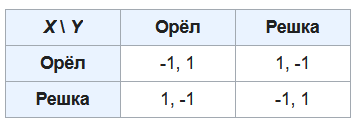


Рис.1 Все возможные исходы

## **Алгоритм Альфа-бета отсечение**

**Альфа-бета-отсечение**— алгоритм поиска, стремящийся сократить количество узлов, оцениваемых в дереве поиска алгоритмом минимакса. Предназначен для **антагонистических игр** и используется для машинной игры. В основе алгоритма лежит идея, что оценивание ветви дерева поиска может быть досрочно прекращено (без вычисления всех значений оценивающей функции), если было найдено, что для этой ветви значение оценивающей функции в любом случае хуже, чем вычисленное для предыдущей ветви. Альфа-бета-отсечение является оптимизацией, так как не влияет на корректность работы алгоритма.

**Минимакс** — правило принятия решений, используемое в теории игр, теории принятия решений, исследовании операций, статистике и философии для минимизации возможных потерь из тех, которые лицу, принимающему решение, нельзя предотвратить при развитии событий по наихудшему для него сценарию.

Критерий минимакса первоначально был сформулирован в теории игр для игры двух лиц с нулевой суммой Джеймсом Уолдгрейвом в 1713 году, в случаях последовательных и одновременных ходов, впоследствии получил развитие в более сложных играх и при принятии решений в условиях неопределённости.

Преимущество альфа-бета-отсечения фактически заключается в том, что некоторые из ветвей подуровней дерева поиска могут быть исключены после того, как хотя бы одна из ветвей уровня рассмотрена полностью. Так как отсечения происходят на каждом уровне вложенности (кроме последнего), эффект может быть весьма значительным. На эффективность метода существенно влияет предварительная сортировка вариантов (без перебора или с перебором на меньшую глубину) — при сортировке чем больше в начале рассмотрено «хороших» вариантов, тем больше «плохих» ветвей может быть отсечено без исчерпывающего анализа.

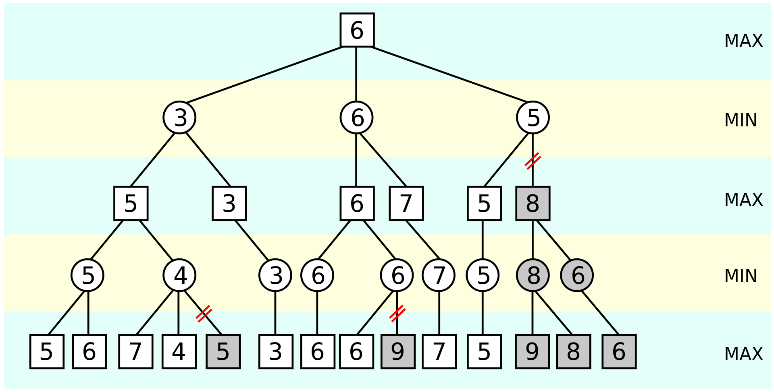


Рис.2 Пример использования алгоритма.

## **Интерфейс**

Для данной работы был выбран консольный интерфейс. Сначала игра попросит пользователя ввести размеры игрового поля и количество символов для победы. После чего программа выведет поле и начнётся игра.

Так как был реализован консольный интерфейс, символами для игры стали две переменные типа **char**.

Во время игры пользователь будет вводить координаты ходов, программа будет ходить в ответ.

Кому будет принадлежать первый ход решает случай. После этого игрок и программа ходят по очереди. После окончания игры, приложение выведет результат и даёт пользователю выбор: сыграть ещё одну партию или закрыть игру.

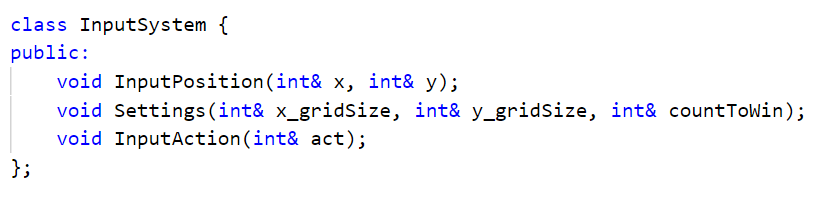
Если пользователь введёт большие размеры поля, в консоли появятся ползунки, что дает **преимущество** перед реализацией игры с графической оболочкой.

Также выбор консольного интерфейса повлияет на **скорость работы** игры и позволит запустить её не опасаясь провести вечер в ожидании хода программы.

# 

# Глава 2. Реализация

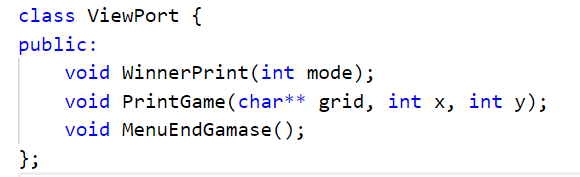
## **2.1. Input System и ViewPort**



Листинг 1. Структура системы ввода

Эта структура отвечает за взаимодействие пользователя с программой. Даёт игроку возможность ввода своих правил игры, таких как размеры игрового поля и условие победы. У данной структуры есть три функции:

1. InputPosition() – Функция, которая спрашивает игрока о его ходе
2. Settings() – Функция, которая считывает настройки, введённые пользователем
3. InputAction() – Функция, которая просит игрока выбрать из нескольких вариантов



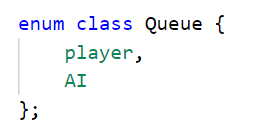
Листинг 1. Структура системы вывода

Структура ViewPort нужна для вывода в консоль данных игры, и общения с пользователем. ViewPort имеет также имеет три функции:

1. WinnerPrint() – Функция вывода результата партии
2. PrintGame() – Функция вывода игрового поля
3. MenuEndGamase() – Функция вывода выбора в конце игры (сыграть ещё партию или закрыть приложение)

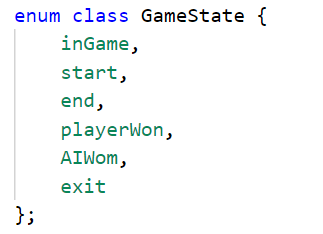
## **2.2. Перечисления**

**Queue** отвечает за то, чья очередь ходить. Так как крестики-нолики – игра для двоих, в очереди будут всего два игрока – пользователь и программа.



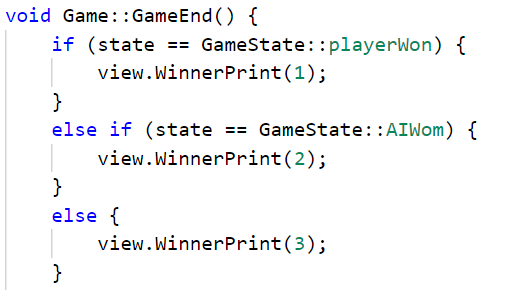
Листинг 3. Очередь игроков

**GameState** нужен для отслеживания стадий игры. С помощью этого можно проверят на какой стадии сейчас находится игра и изменять её.



Листинг 4. GameState

Ниже представлена часть функции **GameEnd**(), в которой проводится проверка стадии игры для корректного вывода результата партии.



Листинг 5. Пример использования GameState

## **2.3. Класс Game**

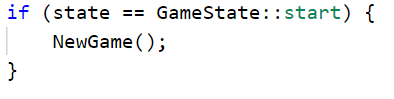
Класс **Game** – структура, которая представляет из себя всю игру. В ней есть функции начла и конца игры, инициализации и отчистки игрового поля, проверки статуса игры, поля, отвечающие за игровое поле и его размеры, условие победы и пр. Большинство методов класса **Game** – закрытые, в связи с ненадобностью работы с ними. Также раз это класс, у него должны быть конструктор и деструктор.

|  |  |
| --- | --- |
| https://i.imgur.com/3ajWRiU.png | https://i.imgur.com/ZHt84b7.png |

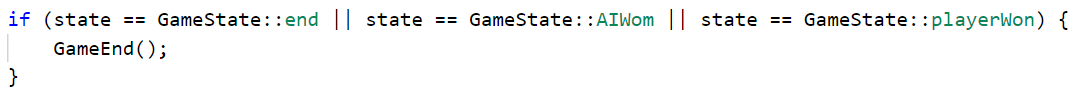
Листинг 6. Все поля и функции структуры Game

## **2.4. Функция Game::Logic()**

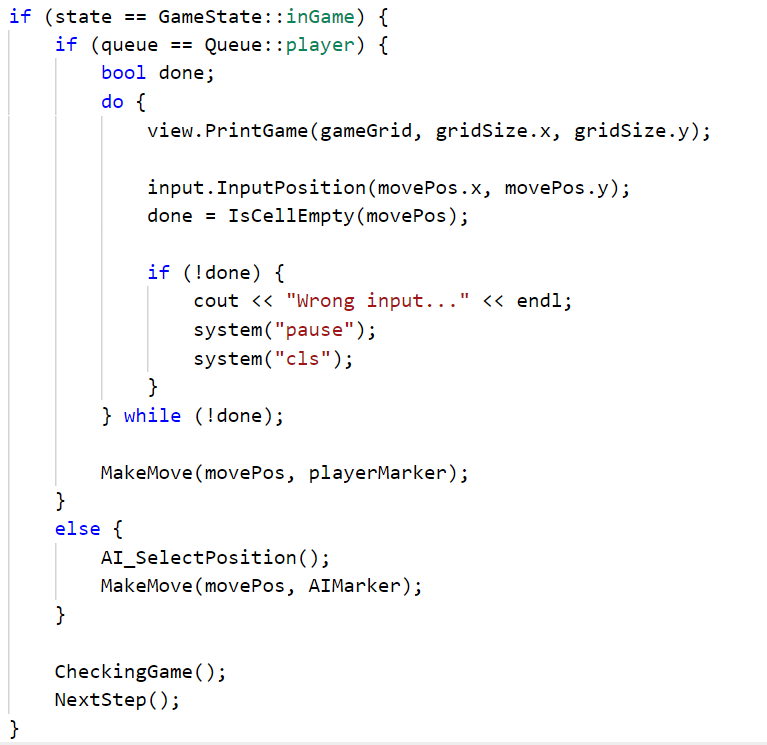
**Game::Logic** – Функция, в которой и происходит игра. Она проверяет на какой стадии в данный момент игра, и в связи с этими данными, вызывает другие функции, такие как инициализация игры, вывод результата партии, и пр.



Листинг 7. Инициализация игры на этапе “start”



Листинг 8. Конец игры на этапах “end”,“AIWon” и “playerWon”

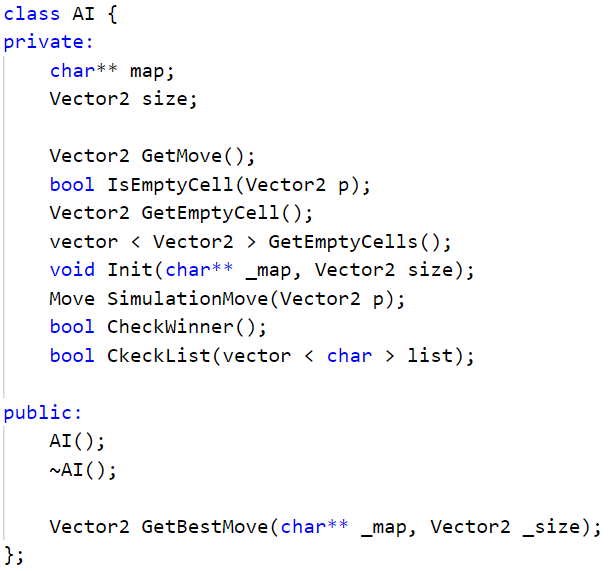


Листинг 8. Проверка на стадию “inGame”

Если стадия игры – **“inGame”**, функция проверит чей сейчас ход, и если сейчас настало время ходить пользователя, игрок должен будет ввести свой ход. Дальше следует проверка на правильность ввода, и затем, если ввод правильный, функция сделает ход. Если на этом этапе время хода программы, то функция просто попросит сущность AI выбрать позицию, и сделает ход. После этого функция проверит выиграл ли кто-то, и игра перейдёт на следующую стадию.

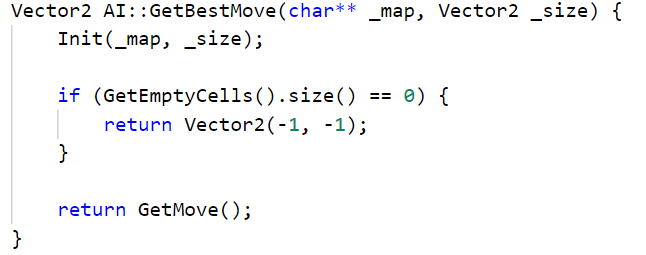
## **2.5. Сущность AI**

Для реализации игры против компьютера была реализована сущность **AI**. Сама структура имеет несколько полей и закрытых методов. Так как нет нужды в получении доступа к некоторым методам и полям, они были перенесены в спецификатор доступа **“private”**. Всего один метод класса имеет спецификатор доступа **“public”** – это функция **GetBestMove**().



Листинг 9. Структура Искусственного Интеллекта

## **2.6. AI::GetBestMove()**



Листинг 10. Функция GetBestMove()

На вход данному методу поступает текущий вариант игрового поля и его размеры. После чего он инициализирует текущие данные внутри себя, для работы с ними без опасений изменить изначальные данные игры. После этого метод проверяет наличие свободных клеток для хода.

Если таких ячеек нет, функция возвращает ход (-1,-1). Так как такого хода не может существовать, программа поймет, что игровое поле полностью заполнено, и перейдёт на следующий этап – анализ и вывод результата партии.

Если же есть минимум одна клетка для хода, метод вернёт результат выполнения функции **GetMove**().

## **2.7. AI::GetMove()**

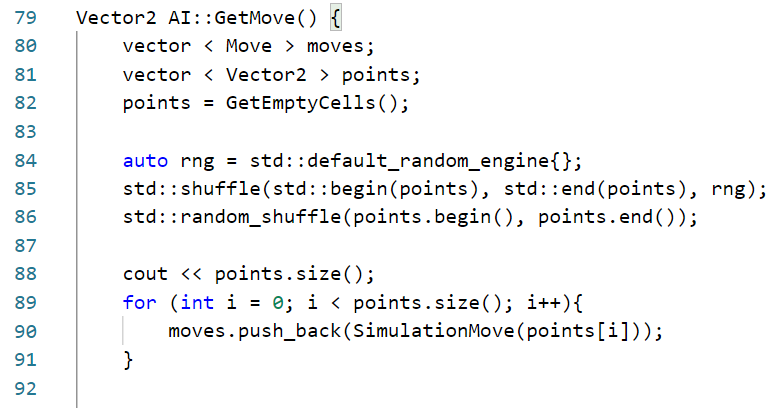
Функция **GetMove** отвечает за логику виртуального оппонента. Сначала функция собирает все доступный ячейки. Перемешивает их, что добавляет вариативности. После этого функция проходится по всем записанным клеткам. Если был найден лучший ход, функция вернёт его координаты.

|  |  |
| --- | --- |
| https://i.imgur.com/c2XNRys.png | https://i.imgur.com/T8DIuZr.png |

Листинг 11. Функция GetMove

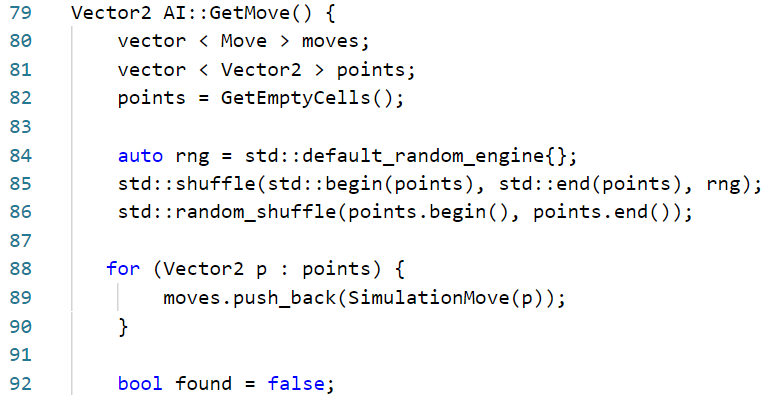
# Глава 3. Оптимизация алгоритмов

# 3.1. Оптимизация циклов



Листинг 12. До оптимизации цикла

При выполнении функции GetMove(), программа записывает ценность тех или иных ходов. До оптимизации запись была реализована с помощью цикла for. Существует намного более эффективный цикл под названием for\_each (цикл, основанный на диапазоне), который предоставляет более простой и безопасный способ итерации по массиву (или по любой другой структуре типа списка).



Листинг 13. После оптимизации цикла

Оптимизировав код, переписав цикл for на цикл for\_each мы получили более быстрый вариант, который работает в разы лучше. Продемонстрируем это на примере обработки и записи ценности ходов в 100 партиях.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Замер 1 | Замер 2 | Замер 3 | Замер 4 |
| Цикл for | 4,6 сек | 4,7 сек | 4,9 сек | 4,1 сек |
| Цикл for\_each | 0,007 сек | 0,005 сек | 0,005 сек | 0,003 сек |

Таблица 1. Результаты оптимизации

# 3.2 Оптимизация вызовов функций

Данная оптимизация не имеет места в данной работе, т.к. не предусматривает вызов целого алгоритма действий по отношению ко входным данным. Ни одна функция не может быть реализована макросом [#define](https://vk.com/im?sel=225844612&st=%23define).

# Заключение

При реализации этого приложения, я много узнал о теории игр, и оптимизации игр с нулевой суммой. Также во время работы над игрой я реализовал версию на микроконтроллерах, и в будущем собираюсь сделать маленькую портативную консоль, но это уже выходит за рамки работы.

Данное приложение достаточно оптимизированно относительно задач, которое оно выполняет. Первые версии думали над ходом около 10 секунд, но в итоге я оптимизировал алгоритм до приемлемой скорости работы.

# Cписок используемых источников

1. Страуструп Б. «Язык программирования C++. Специальное издание. Пер. с англ». — М.: Издательство Бином, 2011 г. — 1136 с: ил.
2. Конова Е. "Алгоритмы и программы. Язык С++. Учебное пособие. Гриф УМО вузов РФ", издательство: Лань, год издания:2020.
3. Шилдт Г. "C++ для начинающих", издательство: ЭКОМ, год издания: 2007.
4. Барбара Э. "Язык программирования C++. Базовый курс. Руководство", издательство: Диалектика/Вильямс, год издания: 2014.

# Приложение

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#include <random>

using namespace std;

class ViewPort {

public:

void WinnerPrint(int mode);

void PrintGame(char\*\* grid, int x, int y);

void MenuEndGamase();

};

struct Vector2 {

int x,

y;

Vector2(int \_x, int \_y) : x(\_x), y(\_y) {}

Vector2() : x(0), y(0) {}

Vector2(const Vector2& p) : x(p.x), y(p.y) {}

friend bool operator == (const Vector2& p1,

const Vector2& p2);

};

class Move {

public:

Move(Vector2 \_p);

Move();

Vector2 p;

float estimation;

int CompareTo(Move other);

};

class InputSystem {

public:

void InputPosition(int& x, int& y);

void Settings(int& x\_gridSize, int& y\_gridSize, int& countToWin);

void InputAction(int& act);

};

enum class Queue {

player,

AI

};

enum class GameState {

inGame,

start,

end,

playerWon,

AIWom,

exit

};

class AI {

private:

char\*\* map;

Vector2 size;

Vector2 GetMove();

bool IsEmptyCell(Vector2 p);

Vector2 GetEmptyCell();

vector < Vector2 > GetEmptyCells();

void Init(char\*\* \_map, Vector2 size);

Move SimulationMove(Vector2 p);

bool CheckWinner();

bool CkeckList(vector < char > list);

public:

AI();

~AI();

Vector2 GetBestMove(char\*\* \_map, Vector2 \_size);

};

Vector2 AI::GetMove() {

vector < Move > moves;

vector < Vector2 > points;

points = GetEmptyCells();

auto rng = std::default\_random\_engine{};

std::shuffle(std::begin(points), std::end(points), rng);

std::random\_shuffle(points.begin(), points.end());

for (Vector2 p : points) {

moves.push\_back(SimulationMove(p));

}

bool found = false;

Move move;

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

if (moves[i].estimation > 0) {

move = moves[i];

found = true;

break;

}

}

if (found) {

return move.p;

}

else {

if (points.size() != 0)

return GetEmptyCell();

return Vector2(-1, -1);

}

}

bool AI::IsEmptyCell(Vector2 p) {

if (map[p.x][p.y] != '\_')

return false;

return true;

}

Vector2 AI::GetEmptyCell() {

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

for (int j = 0; j < size.y; j++)

if (map[i][j] == '\_')

return Vector2(i, j);

return Vector2(-1, -1);

}

vector < Vector2 > AI::GetEmptyCells() {

vector < Vector2 > points;

Vector2 point;

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

for (int j = 0; j < size.y; j++)

if (map[i][j] == '\_') {

point.x = i;

point.y = j;

points.push\_back(point);

}

return points;

}

void AI::Init(char\*\* \_map, Vector2 \_size) {

size.x = \_size.x;

size.y = \_size.y;

map = new char\* [size.x];

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

map[i] = new char[size.y];

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

for (int j = 0; j < size.y; j++)

map[i][j] = \_map[i][j];

}

Move AI::SimulationMove(Vector2 p) {

map[p.x][p.y] = '0';

Move m(p);

if (CheckWinner())

m.estimation = 1;

else

m.estimation = 0;

map[p.x][p.y] = '\_';

return m;

}

bool AI::CheckWinner() {

vector < char > list;

int stateIA = 0;

int statePL = 0;

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

for (int j = 0; j < size.y; j++) {

list.push\_back(map[i][j]);

}

if (CkeckList(list)) {

stateIA++;

}

else {

statePL++;

}

list.clear();

}

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

for (int j = 0; j < size.y; j++) {

list.push\_back(map[j][i]);

}

if (CkeckList(list)) {

stateIA++;

}

else {

statePL++;

}

list.clear();

}

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

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

list.push\_back(map[i + j][j]);

}

if (CkeckList(list)) {

stateIA++;

}

else {

statePL++;

}

list.clear();

}

for (int i = 1; i < size.x; ++i) {

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

list.push\_back(map[i + j][j]);

}

if (CkeckList(list)) {

stateIA++;

}

else {

statePL++;

}

list.clear();

}

for (int j = size.y; j > 0; --j) {

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

if (size.x - j - i >= 0) {

list.push\_back(map[i][size.x - j - i]);

}

}

if (CkeckList(list)) {

stateIA++;

}

else {

statePL++;

}

list.clear();

}

for (int j = 0; j < size.y; ++j) {

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

if (size.x + j - i < size.x) {

if (size.x - j - i >= 0) {

list.push\_back(map[i][size.x + j - i]);

}

}

}

if (CkeckList(list)) {

stateIA++;

}

else {

statePL++;

}

list.clear();

}

if (stateIA > statePL)

return true;

else

return false;

}

bool AI::CkeckList(vector < char > list) {

int countPl = 0;

int countAI = 0;

if (list.size() == 0)

return false;

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

if (list[i] == 'X') {

countPl++;

if (i < list.size() - 1) {

if (list[i + 1] != 'X') {

countPl = 0;

}

else {

if (i == list.size() - 1) {

countPl++;

}

}

}

}

if (list[i] == '0') {

countAI++;

if (i < list.size() - 1) {

if (list[i + 1] != '0') {

countAI = 0;

}

else {

if (i == list.size() - 1) {

countAI++;

}

}

}

}

}

return countAI >= countPl;

}

AI::AI() {

map = new char\* [0];

size.x = 0;

size.y = 0;

}

AI::~AI() {

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

delete[] map[i];

delete[] map;

}

Vector2 AI::GetBestMove(char\*\* \_map, Vector2 \_size) {

Init(\_map, \_size);

if (GetEmptyCells().size() == 0) {

return Vector2(-1, -1);

}

return GetMove();

}

class Game {

private:

Vector2 gridSize;

char\*\* gameGrid;

int countToWin;

Queue queue;

GameState state;

char AIMarker;

AI ai;

char playerMarker;

Vector2 movePos;

InputSystem input;

ViewPort view;

bool fisrtLounch;

void NewGame();

void GameEnd();

void InitGameGrid(Vector2 size);

void DefineQueue();

void ClearGameGrid();

bool CkeckList(vector < char > list);

void CheckingGame();

void NextStep();

bool MakeMove(Vector2 pos, char marker);

bool IsCellEmpty(Vector2 pos);

void AI\_SelectPosition();

public:

Game();

~Game();

void Logic();

};

void Game::NewGame() {

if (!fisrtLounch) {

ClearGameGrid();

}

fisrtLounch = false;

DefineQueue();

input.Settings(gridSize.x, gridSize.y, countToWin);

InitGameGrid(gridSize);

state = GameState::inGame;

Logic();

}

void Game::GameEnd() {

if (state == GameState::playerWon) {

view.WinnerPrint(1);

}

else if (state == GameState::AIWom) {

view.WinnerPrint(2);

}

else {

view.WinnerPrint(3);

}

int act;

cout << endl;

view.MenuEndGamase();

input.InputAction(act);

if (act == 1) {

state = GameState::start;

Logic();

}

else {

state = GameState::exit;

return;

}

}

void Game::InitGameGrid(Vector2 size) {

gridSize = size;

gameGrid = new char\* [gridSize.x];

for (int i = 0; i < gridSize.x; i++)

gameGrid[i] = new char[gridSize.y];

for (int i = 0; i < gridSize.x; i++) {

for (int j = 0; j < gridSize.y; j++) {

gameGrid[i][j] = '\_';

}

}

}

void Game::DefineQueue() {

int randVal = 1 + rand() % 10;

if (randVal <= 5) {

queue = Queue::player;

}

else {

queue = Queue::AI;

}

}

void Game::ClearGameGrid() {

for (int i = 0; i < gridSize.x; i++)

delete[] gameGrid[i];

delete[] gameGrid;

}

bool Game::CkeckList(vector < char > list) {

int countPl = 0;

int countAI = 0;

if (list.size() == 0)

return false;

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

if (list[i] == playerMarker) {

countPl++;

if (countPl >= countToWin) {

state = GameState::playerWon;

return true;

}

if (i < list.size() - 1) {

if (list[i + 1] != playerMarker) {

countPl = 0;

}

else {

if (i == list.size() - 1) {

countPl++;

}

}

}

}

if (list[i] == AIMarker) {

countAI++;

if (countAI >= countToWin) {

state = GameState::AIWom;

return true;

}

if (i < list.size() - 1) {

if (list[i + 1] != AIMarker) {

countAI = 0;

}

else {

if (i == list.size() - 1) {

countAI++;

}

}

}

}

}

return false;

}

void Game::CheckingGame() {

vector < char > list;

for (int i = 0; i < gridSize.x; i++) {

for (int j = 0; j < gridSize.y; j++) {

list.push\_back(gameGrid[i][j]);

}

if (CkeckList(list)) {

return;

}

else {

list.clear();

}

}

for (int i = 0; i < gridSize.x; i++) {

for (int j = 0; j < gridSize.y; j++) {

list.push\_back(gameGrid[j][i]);

}

if (CkeckList(list)) {

return;

}

else {

list.clear();

}

}

for (int i = 0; i < gridSize.x; ++i) {

for (int j = 0; i + j < gridSize.y; ++j) {

list.push\_back(gameGrid[i + j][j]);

}

if (CkeckList(list)) {

return;

}

else {

list.clear();

}

}

for (int i = 1; i < gridSize.x; ++i) {

for (int j = 0; i + j < gridSize.y; ++j) {

list.push\_back(gameGrid[i + j][j]);

}

if (CkeckList(list)) {

return;

}

else {

list.clear();

}

}

for (int j = gridSize.y; j > 0; --j) {

for (int i = 0; i < gridSize.x; ++i) {

if (gridSize.x - j - i >= 0) {

list.push\_back(gameGrid[i][gridSize.x - j - i]);

}

}

if (CkeckList(list)) {

return;

}

else {

list.clear();

}

}

for (int j = 0; j < gridSize.y; ++j) {

for (int i = 0; i < gridSize.x; ++i) {

if (gridSize.x + j - i < gridSize.x) {

if (gridSize.x - j - i >= 0) {

list.push\_back(gameGrid[i][gridSize.x + j - i]);

}

}

}

if (CkeckList(list)) {

return;

}

else {

list.clear();

}

}

int countStep = 0;

for (int i = 0; i < gridSize.x; i++) {

for (int j = 0; j < gridSize.y; j++) {

if (gameGrid[i][j] == '\_')

countStep++;

}

}

if (countStep == 0)

state == GameState::end;

}

void Game::NextStep() {

if (queue == Queue::AI) {

queue = Queue::player;

}

else {

queue = Queue::AI;

}

}

bool Game::MakeMove(Vector2 pos, char marker) {

if (!IsCellEmpty(pos))

return false;

gameGrid[pos.x][pos.y] = marker;

return true;

}

bool Game::IsCellEmpty(Vector2 pos) {

if (pos.x < 0 || pos.x >= gridSize.x) {

return false;

}

if (pos.y < 0 || pos.y >= gridSize.y) {

return false;

}

if (gameGrid[pos.x][pos.y] != '\_')

return false;

return true;

}

void Game::AI\_SelectPosition() {

movePos = ai.GetBestMove(gameGrid, gridSize);

if (!IsCellEmpty(movePos))

AI\_SelectPosition();

}

Game::Game() {

fisrtLounch = true;

gridSize.x = 3;

gridSize.y = 3;

countToWin = 3;

queue = Queue::player;

state = GameState::start;

playerMarker = 'X';

AIMarker = '0';

}

Game::~Game() {

ClearGameGrid();

}

void Game::Logic() {

do {

if (state == GameState::start) {

NewGame();

}

if (state == GameState::inGame) {

if (queue == Queue::player) {

bool done;

do {

view.PrintGame(gameGrid, gridSize.x, gridSize.y);

input.InputPosition(movePos.x, movePos.y);

done = IsCellEmpty(movePos);

if (!done) {

cout << "Wrong input..." << endl;

system("pause");

system("cls");

}

} while (!done);

MakeMove(movePos, playerMarker);

}

else {

AI\_SelectPosition();

MakeMove(movePos, AIMarker);

}

CheckingGame();

NextStep();

}

if (state == GameState::end || state == GameState::AIWom || state == GameState::playerWon) {

GameEnd();

}

if (state != GameState::exit) {

break;

}

system("cls");

} while (true);

}

void InputSystem::InputPosition(int& x, int& y) {

cout << "Your turn (x, y): ";

cin >> x >> y;

x--;

y--;

system("cls");

}

void InputSystem::Settings(int& x\_gridSize, int& y\_gridSize, int& countToWin) {

cout << "Game size (x, y): ";

cin >> x\_gridSize >> y\_gridSize;

cout << "Win size: ";

cin >> countToWin;

system("cls");

}

void InputSystem::InputAction(int& act) {

cout << "Your choise: ";

cin >> act;

system("cls");

}

Move::Move(Vector2 \_p) {

p.x = \_p.x;

p.y = \_p.y;

estimation = 0;

}

Move::Move() {

p.x = -1;

p.y = -1;

estimation = 0;

}

int Move::CompareTo(Move other) {

auto res = CompareTo(other);

if (res == 0 && p.x == 1 && p.y == 1) return 1;

return res;

}

bool operator == (const Vector2& p1,

const Vector2& p2) {

if (p1.x != p2.x || p1.y != p2.y)

return false;

return true;

}

void ViewPort::WinnerPrint(int mode) {

cout << "Game over" << endl;

if (mode == 1) {

cout << "You win!!!" << endl;

}

else if (mode == 2) {

cout << "You lose..." << endl;

}

}

void ViewPort::PrintGame(char\*\* grid, int x, int y) {

cout << " === TicTacToe === " << endl;

cout << " ";

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

cout << " " << i + 1;

}

cout << endl;

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

cout << i + 1 << " ";

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

cout << grid[i][j] << " ";

}

cout << endl;

}

cout << "===================================" << endl;

cout << "\_ - empty cell, X - Player, 0 - Bot" << endl;

cout << "===================================" << endl;

}

void ViewPort::MenuEndGamase() {

cout << "Chose: " << endl <<

"(0) Exit" << endl <<

"(1) One more time" << endl;

}

int main() {

//setlocale(LC\_ALL, "rus");

srand(time(NULL));

Game game;

game.Logic();

}