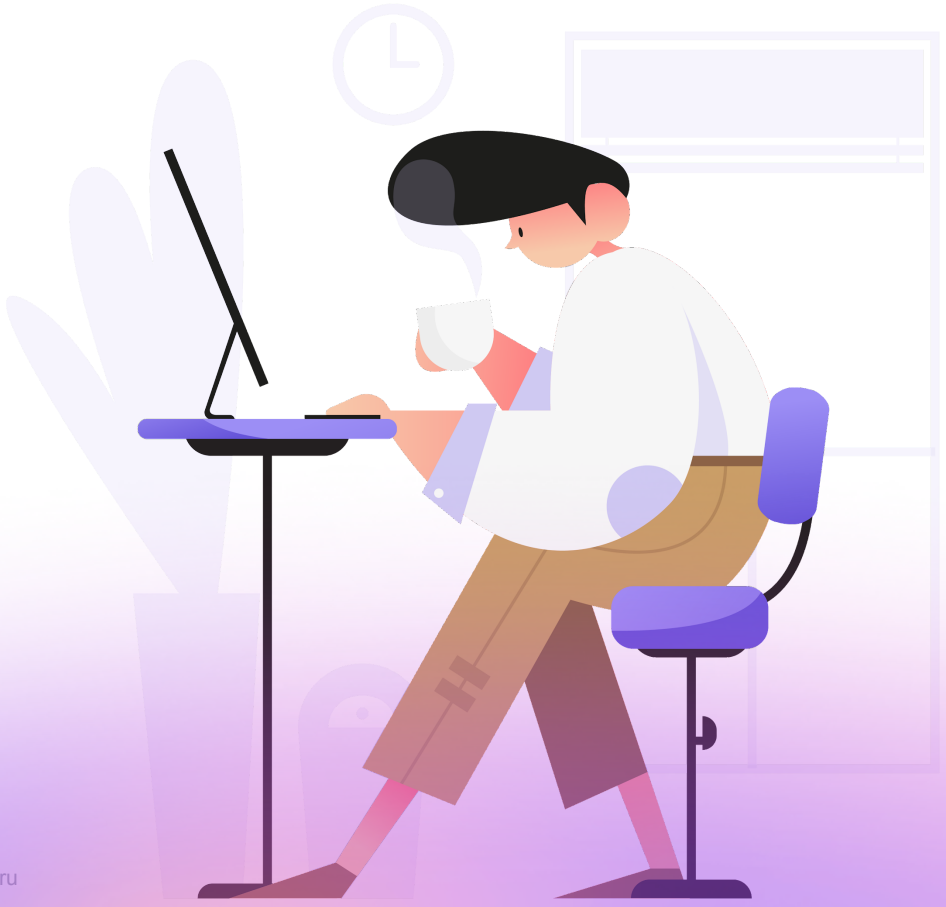
GeekBrains

Основы C++

Итоги.

Игра в крестики-нолики



На этом уроке

1. Научимся базовой декомпозиции задач
2. Применим все (или почти все) полученные ранее знания на практике

Оглавление

Предисловие

Практическое применение языка и технологий всегда разительно отличается от теоретического изучения. Академические примеры, демонстрирующие поведение тех или иных механизмов важны, но имеют свойство быстро забываться, именно поэтому данный курс, как и многие другие, ориентирован на практику программирования, применение изученных технологий и формирование причинно-следственных связей в части использования тех или иных конструкций. То есть, не просто демонстрирует один из множества вариантов решения, но и показывает почему в той или иной части программы нужно использовать тот или иной механизм.

В практическом применении языка есть одна важная особенность: нужно отталкиваться от поставленной задачи, а не от изученных технологий, поскольку во всех случаях без исключения именно задача диктует выбор технологии, а не наоборот (так, кстати, рождаются новые языки программирования, когда разработчики приходят к выводу, что существующие не справляются с задачей).

Задача

Нашей задачей будет научить компьютер играть с пользователем в игру крестики-нолики, и научиться делать это с использованием только терминала операционной системы. Начинают решать любые задачи обычно с анализа того, что необходимо. В нашем случае, для игры в крестики-нолики. Для нас

это будут: поле для игры, символ крестика, символ нолика и пустой символ, он нужен будет для более-менее симпатичного отображения поля в терминале.

```
enum PLAYER {HUMAN='X', AI='O', EMPTY='_'};

typedef struct {
   int szY;
   int szX;
   PLAYER** map;
   int towin;
} Field;
```

Действия

Опишем действия, необходимые для игры в крестики-нолики. Некоторые из этих действий потребуют описания вспомогательных функций, но об этом поговорим в конце, когда будем описывать сам процесс игры (игровой цикл).

Начать новую игру

```
void init(Field &field) {
   field.towin = 3;
   field.szY = 3;
   field.szX = 3;
   field.map = (PLAYER **) calloc(sizeof(PLAYER *), field.szY);
   for (int y = 0; y < field.szY; ++y) {
       *(field.map + y) = (PLAYER *) calloc(sizeof(PLAYER),
field.szX);
       for (int x = 0; x < field.szX; ++x) {
           setval(field.map, y, x, EMPTY);
       }
   }
}
```

Здесь, очевидно, нам понадобится функция, устанавливающая значение “пустая ячейка” по определённым координатам поля. Эта функция обеспечивает создание объекта поля с определёнными размерами, обратите внимание, что размеры поля мы можем также вынести в параметры функции, и научиться играть на полях, отличающихся размерами от 3x3.

Вывести текущее состояние поля

```
void print(Field &field) {
   //#include <windows.h>
   //system("cls")
   std::system("clear");
   printf("-------\n");
   for (int i = 0; i < field.szY; ++i) {
       cout << "|";
       for (int j = 0; j < field.szX; ++j) {
           printf("%c|", getval(field.map, i, j));
       }
       cout << endl;
   }
```

```
}
```

В этой функции мы полностью описываем то, как хотим, чтобы выглядело наше поле для играющего. Сюда можно добавлять любое количество украшений, на вкус программиста, главное, чтобы играющему было выведено текущее состояние каждой отдельной клетки поля. Здесь в комментарии указано необходимое включение и простейшая команда, очищающая командную строку Windows. Здесь, очевидно, нам понадобится функция, получающая значение ячейки поля для демонстрации пользователю

Ход игрока

```
void human(Field &field) {
    int x;
    int y;
    do {
        printf("Введите координаты хода X и Y (от 1 до %d) через пробел >>",
field.szY);
        cin >> x >> y;
        // C-style
        // scanf("%d %d", &x, &y);

        // need to check if numbers are entered
        x--; y--;
    } while (!isvalid(field, x, y) || !isempty(field, x, y));
    setval(field.map, y, x, HUMAN);
}
```

При описании этой функции важно учесть несколько моментов: пользователь может несколько раз ввести неверные координаты, пользователь может ввести не числа, а символы, пользователь может попытаться поставить крестик на уже занятую клетку. Для этого внутри функции организован цикл с постусловием (сначала спросим пользователя, потом проверим, что он ввёл). Здесь мы видим, что нужны ещё две вспомогательные функции, проверяющие попадание внутрь поля и попадание в незанятую клетку. Также комментарием указан способ попытки автоматизированного приведения типов в С-стиле, если же придерживаться работы с потоком ввода, то необходимо осуществить дополнительную проверку ввода.

Ход компьютера

```
void ai(Field &field) {
    int x, y;
    //C++11 <random>
    random_device rd;
    mt19937 mt(rd());
    uniform_real_distribution<double> dist(0, 3);
    do {
        // C-style
        // x = rand() % field->szX;
        // y = rand() % field->szY;
        x = dist(mt);
        y = dist(mt);
    } while (!isempty(field, x, y));
    setval(field.map, y, x, AI);
}
```

Для начала научим компьютер хотя бы как то делать ход, самое простое - делать ход в случайную клетку. Далее мы научим наш компьютер действовать немного умнее, но пока что нам достаточно

того, что он в случайном порядке попадает в поле и не попадает в те клетки, на которых уже стоит какой то символ.

Проверка победы делающего ход

```
int winchk(Field &field, PLAYER c) {
    for (int y = 0; y < field.szX; y++) {
        for (int x = 0; x < field.szY; x++) {
            if (linechk(field, y, x, 1, 0, field.towin, c)) return 1;
            if (linechk(field, y, x, 1, 1, field.towin, c)) return 1;
            if (linechk(field, y, x, 0, 1, field.towin, c)) return 1;
            if (linechk(field, y, x, 1, -1, field.towin, c)) return 1;
        }
    }
    return 0;
}
```

Возможно, самая сложная функция нашей программы. Мы сразу хотим спроектировать её таким образом, чтобы она могла работать на полях любого размера и любой формы, а также поддерживать игру с любым количеством символов подряд для победы (в простейшем случае, для поля 3x3 условием победы будут три символа (крестика или нолика) подряд по любой горизонтали, любой вертикали или любой диагонали). Для осуществления такой проверки необходимо пройти по каждой клетке поля и проверить вертикаль, горизонталь и две диагонали, которые начинаются в этой клетке. Для этого описываем вспомогательную функцию “проверка одной линии”, в аргументы которой продублируем ссылку на поле, проверяемый символ, передадим координаты начала линии, множители “направления проверки” и выигрышную длину.

```
int linechk(Field &field, int y, int x, int vx, int vy, int len, PLAYER c) {
    const int endx = x + (len - 1) * vx;
    const int endy = y + (len - 1) * vy;
    if (!isvalid(field, endx, endy))
        return 0;
    for (int i = 0; i < len; i++)
        if (getval(field.map, (y + i * vy), (x + i * vx)) != c)
            return 0;
    return 1;
}
```

Таким образом функция завершит свою работу, если проверка выходит за пределы поля по одному из направлений, и последовательно пройдёт все ячейки поля от начальных координат в нужном направлении, проверяя, находится ли там переданный ей для проверки символ. Как только найдётся несовпадение - вернётся ложь, истина вернётся только если мы прошли линию нужной длины полностью и на этой линии встречали только нужные символы.

Проверка на ничью

```
int isdraw(Field &field) {
    for (int y = 0; y < field.szY; y++)
        for (int x = 0; x < field.szX; x++)
            if (isempty(field, x, y))
                return 0;
    return 1;
}
```

Возможно, самая простая функция в нашей программе, просто пробегается по всему полю и возвращает ложь, если найдена пустая ячейка. Согласно правил игры в крестики-нолики, если ещё есть куда ставить символ - игра продолжается, если делать ход некуда и никто не победил - ничья.

Игровой цикл

```
void tictactoe() {
    Field field;
    // C-style
    // while (1) {
    while (true) {
        init(field);
        print(field);
        while (true) {
            human(field);
            print(field);
            if (gamechk(field, HUMAN, "Human win!")) break;
            ai(field);
            print(field);
            if (gamechk(field, AI, "AI win!")) break;
        }
        // C-style
        // char answer[1];
        // scanf("%s", answer);
        // if (strcasecmp(answer, "y") != 0)
        //    break;

        string answer;
        cout << "Play again? ";
        // one word to separator
        cin >> answer;

        // STL <algorithm>
        transform(answer.begin(), answer.end(), answer.begin(), ::tolower);
        // only "y"
        // if (answer != "y")

        if (answer.find('y') != 0) // consider y, yo, ya, yes, yeah, yep, yay,
etc
            break;
    }
}
```

Игровой цикл - это два бесконечных цикла. Внутренний - это один раунд игры, а внешний - это постоянное переспрашивание пользователя, хочет ли он сыграть снова. Внутри одного раунда мы инициализируем поле, показываем его пользователю и бесконечно ходим по очереди, постоянно проверяя, не закончилась ли игра на каком-то из ходов. Для такой проверки организована дополнительная функция “игровые проверки”, внутри которой осуществляется не только проверка победы, но и проверка на ничью и вывод сообщений о результатах игры, в случае, если какая-то из проверок вернула истину. Сама же вспомогательная функция также возвращает маркер своей успешности, чтобы иметь возможность по результатам её работы прервать выполнение основного игрового цикла.

```
int gamechk(Field &field, PLAYER dot, const string &winString) {
    if (winchk(field, dot)) {
```

```
        cout << winString << endl;
        return 1;
    }
    if (isdraw(field)) {
        cout << "draw" << endl;
        return 1;
    }
    return 0;
}
```

Вспомогательные функции

Раз всё остальное готово, мы можем приступать к заполнению “белых пятен”. Вспомогательные функции всегда желательно писать максимально абстрактно и атомарно (одна функция - одно действие), чтобы иметь возможность переиспользования их в других проектах, создания библиотек таких функций или вовсе создания собственного игрового движка. Поэтому мы выделили написание всех вспомогательных функций в отдельную секцию. С предыдущих занятий у нас было описана пара полезных макросов, вспомним их и применим в соответствующих функциях получения значения и проверки попадания символа в поле, сразу дополним эти функции установщиком значений и проверкой на наличие символа в клетке:

```
#define CHK_DOT(x, sz) ((x) >= 0 && (x) < (sz))
#define POINT_ITEM(a, r, c) (*((*(a + r)) + c))

char getval(PLAYER** array, const int row, const int col) {
    return POINT_ITEM(array, row, col);
}
void setval(PLAYER** array, const int row, const int col, PLAYER value) {
    POINT_ITEM(array, row, col) = value;
}
int isvalid(Field &field, int x, int y) {
    return CHK_DOT(x, field.szX) && CHK_DOT(y, field.szY);
}
int isempty(Field &field, int x, int y) {
    return getval(field.map, y, x) == EMPTY;
}
```

Помимо этих простых и довольно общих функций нам нужно будет научить наш компьютер действовать немного умнее, чем просто ставить нолик в случайную клетку. Для этого опишем две простейшие функции “попытка выиграть” и “помешать игроку”. Вызывать их будем внутри функции хода компьютера, а делать они будут следующее:

- первая функция будет пытаться поставить нолик в каждую пустую клетку поля, и возвращать истину, если такая установка дала выигрышную комбинацию, или возвращать обратно пустоту в клетку. Возврат истины нужен для корректной реакции вызывающей функции, чтобы показать, что символ уже установлен и необходимо передать управление игре.
- вторая будет пытаться поставить крестик в каждую пустую клетку поля, и заменять этот крестик ноликом, если полученная комбинация является выигрышной для игрока. Если же комбинация не выигрышная, заменить установленный крестик обратно на пустую клетку. Вернём истину, если успешно противодействуем игроку (нужно для корректной реакции вызывающей функции, иначе мы не только противодействуем, но и ставим лишний нолик в случайную клетку поля)

```
int aiwinchk(Field &field) {
    for (int y = 0; y < field.szY; y++) {
        for (int x = 0; x < field.szX; x++) {
```

```
            if (isempty(field, x, y)) {
                setval(field.map, y, x, AI);
                if (winchk(field, AI))
                    return 1;
                setval(field.map, y, x, EMPTY);
            }
        }
    }
    return 0;
}

int humwinchk(Field &field) {
    for (int y = 0; y < field.szY; y++) {
        for (int x = 0; x < field.szX; x++) {
            if (isempty(field, x, y)) {
                setval(field.map, y, x, HUMAN);
                if (winchk(field, HUMAN)) {
                    setval(field.map, y, x, AI);
                    return 1;
                }
                setval(field.map, y, x, EMPTY);
            }
        }
    }
    return 0;
}
```

Заключение

Как вы можете заметить, практика программирования состоит не только и не столько из знаний о синтаксисе того или иного языка программирования, но из решений задач, составления алгоритмов действий и применения изученных конструкций в уместном контексте. При решении практических задач важно помнить две вещи:

- что компьютер не имеет никакого жизненного опыта, поэтому не сможет выполнить действие, которое “ну тут и так понятно, что надо”;
- что работа компьютера не является магией, и компьютер всегда в точности делает именно то, что написал в своём коде программист.

Ключ к решению любой сложной задачи - это её декомпозиция до понятных конструкций. В нашем случае, до понятных компьютеру конструкций.

Практическое задание

1. Описать недостающие проверки в коде или доработать проект, чтобы пользователь мог выбирать размеры поля до начала игры.

Используемые источники

1. *Брайан Керниган, Деннис Ритчи.* Язык программирования C. — Москва: Вильямс, 2015. — 304 с. — ISBN 978-5-8459-1975-5.

2. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)