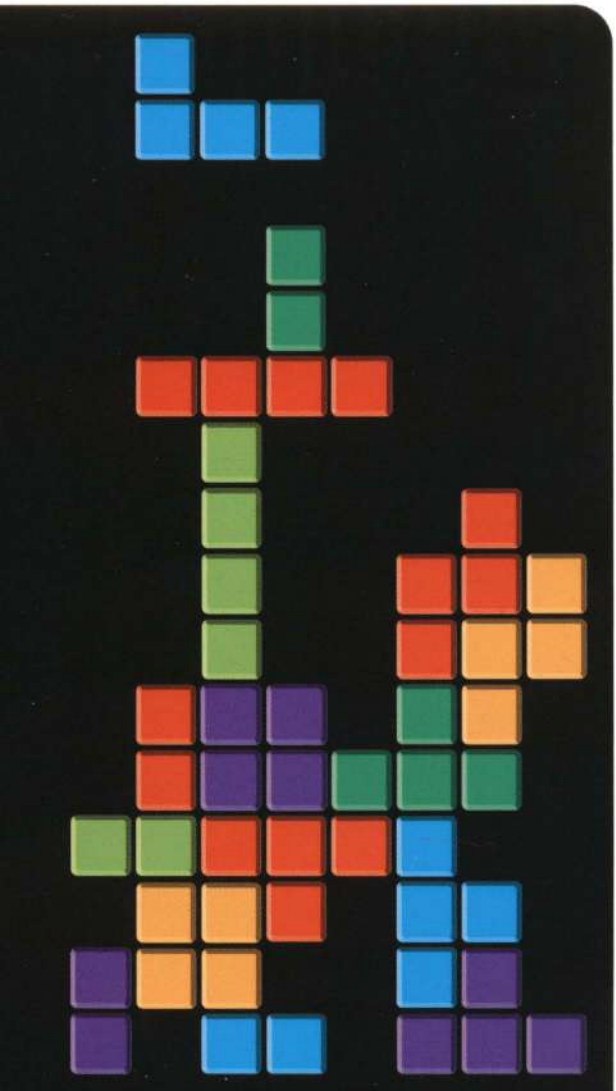
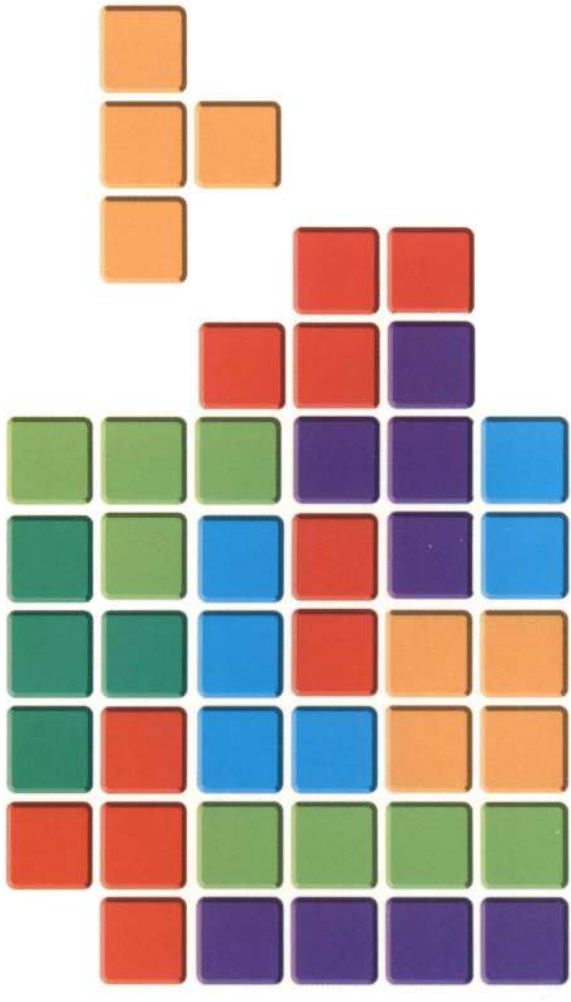
Павел Довгалюк

bhv®

# БАЗОВЫЕ АЛГОРИТМЫ

## Реализация на Python и C++ на примере классических игр





Павел Довгалюк

# БАЗОВЫЕ АЛГОРИТМЫ

## Реализация на Python и C++ на примере классических игр

Санкт-Петербург
«БХВ-Петербург»
2025

УДК 004.43
ББК 32.973.26-018.2
Д58

**Довгалюк П. М.**

Д58 Базовые алгоритмы. Реализация на Python и C++ на примере классических игр. — СПб.: БХВ-Петербург, 2025. — 400 с.: ил.

ISBN 978-5-9775-2100-0

**В книге дается базовая алгоритмическая подготовка, классические алгоритмы разобраны одновременно на двух языках — C++ и Python на примере широко известных и интуитивно понятных компьютерных игр. Наглядно излагается реализация циклов, перебора, рекурсии, эвристики, рассматривается работа с деревьями решений на примере шахмат, предлагается знакомство с прямоугольными координатами, дается введение в самообучающиеся алгоритмы. Книга ориентирована как на начинающих программистов, так и на читателей, планирующих перейти на C++ или Python как на второй язык. Примеры, рассмотренные в книге, помогут сделать первые шаги на пути к профессиональному программированию игр.**

*Для программистов*

УДК 004.43
ББК 32.973.26-018.2

**Группа подготовки издания:**

| | |
|---|---|
| Руководитель проекта | *Олег Сивченко* |
| Зав. редакцией | *Людмила Гауль* |
| Редактор | *Анна Брезман* |
| Компьютерная верстка | *Павла Довгалюка* |
| Корректор | *Анна Брезман* |
| Дизайн обложки | *Зои Канторович* |

Подписано в печать 04.07.25.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 32,25.
Тираж 1200 экз. Заказ № 14510.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-2100-0

# Все игры и головоломки в книге

**Сложность программы**

★ Можно сразу начинать набирать программу на компьютере.

★★★ Нужно сначала прочитать главу, где описана игра.

★★★★★ Потребуется изучить большую часть книги.

**Тип игры**

 Программа решает головоломку.

 Игра для одного человека.

 Человек играет против компьютера.


1 Игра «Бросание кубика» ..... 26

★ 

Программа-имитатор бросков игрального кубика

2 Игра «Отгадай число» ..... 29

★ 

Программа загадывает число и предлагает игроку отгадать его

3 Игра «23 спички» ..... 33

★ 

Кто возьмёт последнюю спичку со стола, тот проиграл

4 Игра «Хаммурапи» ..... 40

★★ 

Игра-симулятор жизни правителя аграрной страны

5 Игра «Blackjack» ..... 59

★★★ 

Или «двадцать одно», одна из самых популярных карточных игр

6 Игра «Ферзя в угол» ..... 74
★
♟♚ Кто заведёт в угол ферзя, тот выиграл

7 Игра «Чудовища» ..... 89
★ Игрок пытается найти чудовищ, спрятавшихся на
♟ болоте

8 Игра «Крестики-нолики» ..... 104
★★ Всем известная детская игра, но теперь в неё играет
♟♚ компьютер

9 Числовые ребусы с буквами ..... 124
★ Программа разгадывает арифметический ребус, где
♚ цифры заменены буквами

10 Игра «Быки и коровы» ..... 130
★★ Компьютер отгадывает задуманное число, получая
♟♚ подсказки о совпадении цифр

11 Игра «Морской бой» ..... 154
★★ Нужно первым потопить спрятанные на поле корабли
♟♚ противника

12 Игра «Реверси» ..... 190
★★★★ Настольная игра, где нужно захватить игровое поле,
♟♚ перекрашивая фишки соперника

13 Головоломка «8 ферзей» ..... 212
★★ Как расставить 8 ферзей на шахматной доске, чтобы
♚ они не били друг друга

14 Игра «Крестики-нолики» 4×4 ..... 227
★★★ Усложнённый вариант классических
♟♚ крестиков-ноликов

15 Головоломка «Лабиринт» ..... 248
★★★
♚ Компьютер ищет выход из лабиринта

16 Игра «8» ..... 264
★★★ Поиск порядка перемещения фишек в модификации
♚ знаменитой головоломки «Пятнадцать»

17 Игра «Солитер» ..... 288
★★★★ Фишки прыгают друг через друга, чтобы съесть.
♚ Нужно, чтобы осталась только одна

18 Игра «Калах» ..... 306
★★ Игра с камушками на доске, где нужно захватить их
♟♚ больше, чем соперник

19 Игра «Мини-шахматы» ..... 334
★★★★★ Шахматы на поле 5×5
♟♚

20 Игра «Угадай животное» ..... 366
★★ Программа пытается отгадать задуманное животное,
♟ задавая вопросы. Если не получилось отгадать, она
добавляет правильный ответ в свою базу знаний

21 Игра «6 пешек» ..... 378
★★★ Программа знает только правила игры, но не знает,
♟♚ как выигрывать. Чем больше она проигрывает, тем
лучше играет потом

# Оглавление

**1 Введение** **13**
1.1. Для кого эта книга . . . . . 14
1.2. C++ или Python? . . . . . 14
1.3. Как читать эту книгу . . . . . 14

**2 Обзор языков C++ и Python** **16**
2.1. Структура программы . . . . . 17
2.2. Вывод на экран . . . . . 17
2.3. Ввод данных . . . . . 18
2.4. Ветвления . . . . . 18
2.5. Циклы . . . . . 19
2.6. Функции . . . . . 20
2.7. Изменение переменных-параметров функции . . . . . 21
2.8. Массивы . . . . . 22
2.9. Словари . . . . . 23
2.10. Генерация случайных чисел . . . . . 23
2.11. Другие отличия C++ и Python . . . . . 24

**3 Начнём с простых игр** **25**
3.1. Игра «Бросание кубика» . . . . . 26
3.1.1. Основа программы . . . . . 26
3.1.2. Имитируем игральный кубик . . . . . 27
3.1.3. Задания для самостоятельной работы . . . . . 27
3.2. Игра «Отгадай число» . . . . . 29
3.2.1. Переделываем «Бросание кубика» . . . . . 29
3.2.2. Подсказки для игрока . . . . . 30
3.3. Игра «23 спички» . . . . . 33
3.3.1. Почти готовая игра . . . . . 34
3.3.2. Учим компьютер играть . . . . . 35
3.3.3. Доделываем выбор хода . . . . . 36
3.3.4. Задания для самостоятельной работы . . . . . 36

**4 Изучаем структуру игровой программы 39**
4.1. Игра «Хаммурапи» . . . 40
4.1.1. Функция `main` . . . 51
4.1.2. Инициализация переменных . . . 51
4.1.3. Главный игровой цикл . . . 52
4.1.4. Вывод итогов игры . . . 53
4.1.5. Покупка и продажа земли . . . 54
4.1.6. Игровая позиция . . . 55
4.1.7. Зерно: хлеб и семена . . . 56
4.1.8. Случайные события . . . 57
4.1.9. Задания для самостоятельной работы . . . 58
4.2. Игра «Blackjack» . . . 59
4.2.1. Игровая позиция . . . 60
4.2.2. Вывод карт на руке . . . 61
4.2.3. Раздача случайных карт . . . 62
4.2.4. Определение победителя . . . 63
4.2.5. Главный игровой цикл . . . 65
4.2.6. Ещё один игровой цикл . . . 66
4.2.7. Задания для самостоятельной работы . . . 67

**5 Программирование прямоугольных полей 73**
5.1. Игра «Ферзя в угол» . . . 74
5.1.1. Игровая позиция . . . 75
5.1.2. Начало игры . . . 75
5.1.3. Ввод координат . . . 76
5.1.4. Вывод поля . . . 77
5.1.5. Дорабатываем вывод поля . . . 78
5.1.6. Главный игровой цикл . . . 79
5.1.7. Делаем ходы . . . 80
5.1.8. Искусственный интеллект: анализ ходов . . . 81
5.1.9. Искусственный интеллект: проверка позиции . . . 82
5.1.10. Искусственный интеллект: пишем код . . . 83
5.1.11. Задания для самостоятельной работы . . . 84
5.2. Игра «Чудовища» . . . 89
5.2.1. Игровая позиция . . . 90
5.2.2. Структура программы . . . 91
5.2.3. Расставляем монстров . . . 92
5.2.4. Чиним расстановку монстров . . . 93
5.2.5. Ход игрока . . . 94
5.2.6. Ловим чудовищ . . . 95
5.2.7. Определяем расстояние до монстров . . . 96

5.2.8. Финальная часть . . . 97
5.3. Игра «Крестики-нолики» . . . 104
5.3.1. Игровая позиция . . . 105
5.3.2. Основной цикл, вывод поля . . . 106
5.3.3. Ход игрока, проверка условия победы . . . 107
5.3.4. Ход компьютера . . . 108
5.3.5. Осмысленные ходы компьютера . . . 109
5.3.6. Доделываем алгоритм игры . . . 111
5.3.7. Поиск завершения линии . . . 112
5.3.8. Проверяем все строки, столбцы и диагонали . . . 113
5.3.9. Всё для игры вничью . . . 114

**6 Полный перебор с помощью циклов 123**
6.1. Числовые ребусы с буквами . . . 124
6.1.1. Начинаем писать программу . . . 124
6.1.2. Добавляем код для перебора . . . 125
6.1.3. Проверяем ограничения . . . 126
6.1.4. Send more money! . . . 127
6.1.5. Задания для самостоятельной работы . . . 127
6.2. Игра «Быки и коровы» . . . 130
6.2.1. Отгадывает человек . . . 131
6.2.2. Основные функции . . . 131
6.2.3. Основной цикл игры . . . 133
6.2.4. Как хранить загаданное число . . . 134
6.2.5. Функция generate — генерация случайного числа . . . 135
6.2.6. Функция inputGuess — ввод числа игроком . . . 136
6.2.7. Функция countBullsAndCows — подсчёт «быков» и «коров» . . . 137
6.2.8. Отгадывает компьютер . . . 141
6.2.9. Структура программы . . . 141
6.2.10. Стуктуры данных . . . 143
6.2.11. Узнаём у пользователя число «быков» и «коров» . . . 144
6.2.12. Функция generate — поиск возможных ответов . . . 145
6.2.13. Функция save — сохранение полученных ответов . . . 146
6.2.14. Функция countBullsCows — подсчёт «быков» и «коров» . . . 147
6.2.15. Задания для самостоятельной работы . . . 148

**7 Эвристические алгоритмы 153**
7.1. Игра «Морской бой» . . . 154
7.1.1. Игровая позиция . . . 156
7.1.2. Структура программы . . . 157
7.1.3. Стрельба по кораблям . . . 158

7.1.4. Расстановка кораблей . . . 159
7.1.5. Расстановка кораблей: заполнение клеток . . . 161
7.1.6. Доработка функции стрельбы . . . 162
7.1.7. Расстановка кораблей игрока . . . 164
7.1.8. Расстановка кораблей компьютером . . . 164
7.1.9. Снова ход игрока . . . 165
7.1.10. Случайный ход компьютера . . . 166
7.1.11. Условие окончания игры . . . 167
7.1.12. Добавляем «туман войны» . . . 168
7.1.13. Вычёркиваем клетки вокруг потопленных кораблей . . . 169
7.1.14. Добивание раненого корабля . . . 170
7.1.15. Эвристики для игры компьютера . . . 173
7.1.16. Задания для самостоятельной работы . . . 175
7.2. Игра «Реверси» . . . 190
7.2.1. Игровая позиция . . . 192
7.2.2. Вывод позиции на экран . . . 192
7.2.3. Главный игровой цикл . . . 194
7.2.4. Проверка допустимости хода . . . 195
7.2.5. Ход игрока . . . 197
7.2.6. Переворачивание фишек . . . 197
7.2.7. Ход компьютера . . . 199
7.2.8. Окончание игры . . . 200
7.2.9. Задаём приоритет клеткам . . . 201

**8 Рекурсивные алгоритмы в играх 211**
8.1. Головоломка «8 ферзей» . . . 212
8.1.1. Игровая позиция . . . 212
8.1.2. Общий алгоритм поиска решений . . . 213
8.1.3. Проверка, что один ферзь атакует другого . . . 214
8.1.4. Перебор с помощью циклов . . . 215
8.1.5. Отсеиваем некорректные варианты . . . 216
8.1.6. Что такое рекурсия . . . 217
8.1.7. Выход из рекурсии . . . 219
8.1.8. Рекурсивный перебор всех решений . . . 219
8.1.9. И снова отсеивание некорректных вариантов . . . 220
8.2. Игра «Крестики-нолики» 4×4 . . . 227
8.2.1. Игровая позиция . . . 229
8.2.2. Главный игровой цикл . . . 229
8.2.3. Вывод игрового поля . . . 231
8.2.4. Ход игрока . . . 232
8.2.5. Случайный ход компьютера . . . 232

8.2.6. Определение победителя . . . 233
8.2.7. Выбор наилучшего хода . . . 234
8.2.8. Условие завершения рекурсии . . . 238
8.2.9. Встраиваем анализ ходов в игру . . . 239
8.2.10. Задания для самостоятельной работы . . . 240

**9 Когда рекурсия не подходит 247**
9.1. Головоломка «Лабиринт» . . . 248
9.1.1. Игровая позиция . . . 248
9.1.2. Структура программы . . . 249
9.1.3. Ввод лабиринта . . . 250
9.1.4. Вывод лабиринта . . . 250
9.1.5. Поиск кратчайшего пути . . . 251
9.1.6. Путь с развилками . . . 252
9.1.7. Собираем данные для восстановления пути . . . 253
9.1.8. Восстановление пути . . . 255
9.1.9. Поиск в ширину . . . 256
9.1.10. Реализация поиска в ширину . . . 259
9.1.11. Задания для самостоятельной работы . . . 259
9.2. Игра «8» . . . 264
9.2.1. Оценка числа позиций . . . 264
9.2.2. Игровая позиция . . . 265
9.2.3. Ввод позиции с клавиатуры . . . 266
9.2.4. Как перемещаются фишки . . . 267
9.2.5. Определяем возможные направления движения . . . 268
9.2.6. Пробуем рекурсивный перебор . . . 270
9.2.7. Почему рекурсия не подходит . . . 271
9.2.8. Определение повторяющихся позиций . . . 272
9.2.9. Хранение позиций в словаре . . . 274
9.2.10. Проблемы с рекурсией продолжаются . . . 275
9.2.11. Обход в ширину . . . 276
9.2.12. Реализация обхода в ширину . . . 277
9.2.13. Восстановление последовательности перемещений . . . 280
9.2.14. Задания для самостоятельной работы . . . 282
9.3. Игра «Солитер» . . . 288
9.3.1. Игровая позиция . . . 289
9.3.2. Вывод позиции на экран . . . 290
9.3.3. Ввод стартовой позиции . . . 292
9.3.4. Поиск в ширину . . . 293
9.3.5. Определение возможных перемещений . . . 294
9.3.6. Перемещение фишек . . . 296

9.3.7. Вывод ответа . . . 298
9.3.8. Задания для самостоятельной работы . . . 298

**10 Альфа-бета отсечение: ускоряем рекурсивный перебор 305**
10.1. Игра «Калах» . . . 306
10.1.1. Игровая позиция . . . 307
10.1.2. Главный игровой цикл . . . 309
10.1.3. Вывод игрового поля . . . 310
10.1.4. Чередование ходов . . . 311
10.1.5. Ход игрока . . . 312
10.1.6. Функция посева . . . 313
10.1.7. Посев камней . . . 313
10.1.8. Захват камней . . . 314
10.1.9. Проверка окончания игры . . . 314
10.1.10. Проверка повторного хода . . . 315
10.1.11. Случайный ход компьютера . . . 315
10.1.12. Рекурсивный перебор . . . 316
10.1.13. Оценка позиции . . . 318
10.1.14. Перебор позиций . . . 318
10.1.15. Альфа-бета отсечение . . . 320
10.1.16. Задания для самостоятельной работы . . . 323
10.2. Игра «Мини-шахматы» . . . 334
10.2.1. Игровая позиция . . . 334
10.2.2. Главный игровой цикл . . . 336
10.2.3. Вывод игрового поля . . . 337
10.2.4. Ход игрока . . . 338
10.2.5. Перемещение фигуры . . . 338
10.2.6. Взятие фигуры соперника . . . 339
10.2.7. Проверка корректности хода . . . 340
10.2.8. Перемещение фигур . . . 341
10.2.9. Ход пешки . . . 341
10.2.10. Ход короля . . . 342
10.2.11. Ход коня . . . 343
10.2.12. Ход слона . . . 344
10.2.13. Ход ладьи . . . 345
10.2.14. Ход ферзя . . . 346
10.2.15. Случайный ход компьютера . . . 346
10.2.16. Перебор первого хода . . . 347
10.2.17. Рекурсивный перебор ходов . . . 349
10.2.18. Альфа-бета отсечение . . . 350
10.2.19. Задания для самостоятельной работы . . . 352

**11 Самообучающиеся игры** **365**
11.1. Игра «Угадай животное» . . . . . 366
11.1.1. Алгоритм игры . . . . . 367
11.1.2. Игровая позиция . . . . . 368
11.1.3. Главный игровой цикл . . . . . 370
11.1.4. Задаём игроку вопрос . . . . . 371
11.1.5. Когда достигнут узел-отгадка . . . . . 372
11.1.6. Добавление нового вопроса . . . . . 372
11.1.7. Цикл для последовательности игр . . . . . 373
11.1.8. Задания для самостоятельной работы . . . . . 374
11.2. Игра «6 пешек» . . . . . 378
11.2.1. Игровая позиция . . . . . 379
11.2.2. Цикл для последовательности партий . . . . . 380
11.2.3. Игровой цикл . . . . . 381
11.2.4. Вывод игрового поля . . . . . 381
11.2.5. Проверка окончания игры . . . . . 381
11.2.6. Ход игрока . . . . . 382
11.2.7. Функция завершения хода . . . . . 384
11.2.8. Проверка корректности хода . . . . . 384
11.2.9. Ход компьютера . . . . . 386
11.2.10. Хранение недопустимых позиций . . . . . 387
11.2.11. Добавление и поиск недопустимых позиций . . . . . 388
11.2.12. Задания для самостоятельной работы . . . . . 389

# Глава 1

# Введение

## 1.1. Для кого эта книга

В этой книге описано множество классических игр. Но они не просто описаны, а пошагово разбирается процесс их программирования, для того чтобы показать применение алгоритмов решения задач. Ведь в игровых программах компьютеру часто приходится решать задачи выбора наилучшего хода, поиска путей, анализа решений игрока и тому подобное.

Поэтому книга будет полезна тем, кто изучает алгоритмы, чтобы научиться программировать, сдать экзамен, найти работу. Или просто написать игру своей мечты.

Чтобы читать эту книгу, нужно немного знать C++ или Python. Но очень глубокие знания не нужны, достаточно понимать, что такое переменные, ветвления, циклы и функции. А все библиотечные функции или сложные структуры данных знать не обязательно, они будут объясняться постепенно.

Главное, чтобы у вас уже была установлена какая-то среда разработки (даже компилятора/интерпретатора и текстового редактора будет достаточно) и вы умели ей пользоваться. А как и что программировать, можно будет прочитать в книге.

## 1.2. C++ или Python?

Как правило, программы на C++ состоят из большего количества строк, чем программы на Python. Но в большинстве случаев лишние строки заняты лишь фигурными скобками для циклов и ветвлений. Они не во всех местах обязательны (например, тело цикла может состоять всего из одного оператора), но улучшают читабельность кода, поэтому присутствуют во всех примерах в книге.

Но у C++ есть и преимущество. Программы на нём работают быстрее. У вас будет возможность убедиться в этом на примере нашей книги. Мы будем разрабатывать алгоритмы, которые делают много вычислений при поиске выгодных ходов в игре. Такие алгоритмы, будучи написаны на C++, работают в разы быстрее, чем на Python.

## 1.3. Как читать эту книгу

В книге разбираются классические игры и головоломки. Некоторые из них чисто математические, для других есть выигрышная стратегия у одного из игроков. Каждая программа, которую мы будем писать, разбирается от начала до конца по шагам. Каждый шаг занимает одну страницу, на которой есть небольшой фрагмент кода, добавляемый в конечную программу, а также его разбор. Почти

после каждого шага программу можно запускать, чтобы посмотреть, что в ней изменилось. Так легче избавляться от ошибок, допущенных при написании кода.

Строки во фрагменты листингов на Python иногда не вмещаются в ограничивающие прямоугольники по горизонтали, поэтому в них добавляются переносы на следующую строку, но без обязательного в таких случаях символа \. Это сделано исключительно для экономии места по горизонтали. В случае сомнений можно сверяться с полными листингами.

Игры в книге разделены на несколько классов:

1. Компьютер решает головоломки.
2. Компьютер помогает сгенерировать стартовую позицию и следит за выполнением правил.
3. Компьютер выступает в роли ведущего, который хранит скрытую информацию.
4. Компьютер-соперник, который умеет играть по правилам.

И даже если вы планируете написать многопользовательскую сетевую игру, это не означает, что изучение игр-головоломок будет бесполезным. Потому что в каждой большой программе есть множество модулей. И часть из них могут быть очень похожи на такие головоломки. Например, подпрограмма-ассистент игрока для выбора наилучшего набора товаров в игровом магазине. Или функция подсветки нескольких выгодных путей на карте королевства.

В любом случае хорошее владение классическими алгоритмами помогает выбирать лучшие решения задач. Даже если сам код, который их решает, берётся из готовой библиотеки.

Исходные тексты: `https://github.com/Dovgalyuk/AlgorithmsPythonCpp`

Глава 2

# Обзор языков C++ и Python

## 2.1. Структура программы

В C++ работа программы всегда начинается с функции `main`. Python, напротив, просто выполняет всё содержимое файла. Даже определение функций там — это создание в памяти объектов, содержащих исполняемый код.

Но оформлять код мы будем в одинаковом стиле. Сначала создание или объявление глобальных переменных и констант, а также типов в C++. Затем идут вспомогательные функции, а в самом конце — функция `main` или просто код, который будет выполняться интерпретатором Python (как замена основной функции).

## 2.2. Вывод на экран

Все игры в этой книге работают в текстовом режиме. Программы запускаются в обычной консоли и не требуют графического интерфейса.

Вывод в Python и в C++ работает практически одинаково (с поправкой на синтаксис, конечно).

Разница только в том, что в Python функция `print` в конце всегда переходит на следующую строку, а в C++ для этого явно нужно вывести символ перевода строки `'\n'`. Если нужны дополнительные переводы строк, символ `'\n'` можно также выводить и в Python-программах.

Если же перевод строки в конце оператора не нужен (например, когда необходимо в цикле напечатать несколько переменных в одной строке), то в C++ не нужно делать ничего, а в Python передавать в функцию `print` дополнительный параметр `end=''`.

**C++**

```
#include <iostream>

int main()
{
  // вывод числа в отдельной строке
  std::cout << 10 << '\n';
  // вывод нескольких чисел в одну строку
  for (int i = 0 ; i < 5 ; ++i)
  {
    std::cout << i << ' ';
  }
  // вывод трёх символов перевода строки
  std::cout << "\n\n\n";
}
```

**Python**

```
# вывод числа в отдельной строке
print(10)
# вывод нескольких чисел в одну строку
for i in range(5):
  print(f'{i} ', end='')
# вывод трёх символов перевода строки
print('\n\n')
```

## 2.3. Ввод данных

В любой игре пользователь будет что-то вводить в программу. Это координаты фигур, перемещаемых по игровому полю, ответы на вопросы или какие-то числовые значения.

Мы будем использовать только диалоговый ввод. То есть любое взаимодействие с пользователем будет завершаться нажатием кнопки Enter. К примеру, так не получится сделать управление персонажем с помощью стрелок, но можно ввести сколь угодно сложную строку.

Ввод в C++ и Python отличается довольно сильно. В C++ переменные любого простого типа можно сразу вводить одним оператором >>, а в Python с помощью функции `input` вводится целая строка, которую вручную нужно разбить на части или преобразовать в переменную другого типа.

А вот если в программе на C++ нужно ввести целую строку (с пробелами внутри), тот тут всё становится похожим на Python, потому что используется функция `getline`.

**C++**

```
#include <iostream>
#include <string>

int main()
{
  char c;
  int i, j;
  std::string s;
  // ввод одного символа
  std::cin >> c;
  // ввод двух чисел, разделённых
  ↪  пробелами
  std::cin >> i >> j;
  // ввод строки
  std::getline(std::cin, s);
}
```

**Python**

```
# ввод одного символа
c = input()[0]
# ввод двух чисел, разделённых пробелами
a = input().split()
i = int(a[0])
j = int(a[1])
# ввод строки
s = input()
```

## 2.4. Ветвления

Операторы ветвления в C++ и Python практически не отличаются. Одинаковые ключевые слова `if` и `else`, только в C++ немного больше скобок.

Несколько условий соединяются с помощью логических И и ИЛИ. В C++ они обозначаются как `&&` и `||`, а в Python как `and` и `or`.

В обоих языках вычисление логического выражения в условии завершается, если результат уже понятен. К примеру, если оператор И связывает две части

выражения, первая из которых ложна, то результат тоже будет ложным и поэтому вторая часть вычисляться не будет.

Это свойство можно использовать, чтобы в одном выражении сначала проверить значения переменных, а потом использовать их в вычислениях.

Например, функцию вычисления квадратного корня можно вызывать только для неотрицательных чисел. Поэтому сначала аргумент проверяется, а потом вызывается функция вычисления корня. И это не приводит к ошибке, потому что для отрицательных a вторая часть условия просто не обрабатывается.

C++
```
if (a > 0 && sqrt(a) < 100)
  ...
```

Python
```
if a > 0 and sqrt(a) < 100:
  ...
```

Есть также короткий способ записывать условия, если нужно выбрать результат вычисления одного из двух выражений. В следующем примере переменная b принимает значение 1, если a > 0, и 2 в противном случае.

C++
```
b = a > 0 ? 1 : 2;
```

Python
```
b = 1 if a > 0 else 2
```

## 2.5. Циклы

В C++ есть примерно 4 вида циклов, а в Python — 2. «Примерно» — потому что на самом деле некоторые варианты отличаются только записью, чтобы немного повысить удобство для программиста.

Первый цикл — это цикл с предусловием. Тело цикла выполняется до тех пор, пока истинно условие, записанное после ключевого слова **while**. Следующий цикл выполняется, пока переменная *a* не достигнет значения 10.

C++
```
int a = 0;
while (a < 10)
{
  std::cout << a << ' ';
  ++a;
}
```

Python
```
a = 0
while a < 10:
  print(f'{a} ', end='')
  a += 1
```

Для случаев, когда переменная просто меняется с постоянным шагом, в некоторых языках программирования есть циклы со счётчиком. В C++ и Python синтаксически таких циклов нет, но есть удобный способ использования цикла **for**:

C++
```
for (int a = 0 ; a < 10 ; ++a)
{
  std::cout << a << ' ';
}
```

Python
```
for a in range(10):
  print(f'{a} ', end='')
```

На самом деле в C++ в заголовке цикла можно записывать любые действия, а в Python использовать вместо `range(10)` любой контейнер (или генератор значений). Для перебора элементов контейнера в C++ также есть компактный синтаксис. Там же можно вполне оправданно использовать ключевое слово `auto`, если не хочется описывать тип переменной для элементов контейнера.

**C++**

```
for (auto a : data)
{
  process(a);
}
```

**Python**

```
for a in data:
  process(a)
```

И последний вид цикла, который мы будем часто использовать, это цикл с постусловием. Его особенность в том, что сначала выполняется тело цикла, а затем проверяется условие для продолжения итераций. В C++ для такого цикла есть конструкция `do while`, а в Python придётся её имитировать с помощью бесконечного цикла и условия с оператором `break`, который завершит цикл.

В следующем примере цикл выполняется, пока с клавиатуры не будет введено положительное число.

**C++**

```
int a;
do
{
  std::cin >> a;
}
while (a <= 0);
```

**Python**

```
while True:
  a = int(input())
  if a > 0:
    break
```

## 2.6. Функции

Функция — это фрагмент кода, который можно выполнить неоднократно из разных мест программы. И при этом его не нужно много раз копировать. Вместо этого мы пишем имя функции и передаём ей параметры.

**C++**

```
#include <iostream>
#include <string>

void hello(std::string name)
{
  std::cout << "Привет, " << name <<
  ↪  "!\n";
}

int main()
{
  hello("Маша");
  hello("Дима");
}
```

**Python**

```
def hello(name):
  print(f'Привет, {name}')

hello('Маша')
hello('Дима')
```

Функция может возвращать значение. В C++ тип возвращаемого значения указывается в её описании, а в Python определяется тем выражением, которое в итоге будет возвращаться.

C++

```
#include <iostream>

int mul(int a, int b)
{
  return a * b;
}

int main()
{
  std::cout << mul(7, 8) << " " << mul(2,
  ↪ 2);
}
```

Python

```
def mul(a, b):
  return a * b

print(f'{mul(7, 8)} {mul(2, 2)}')
```

## 2.7. Изменение переменных-параметров функции

Иногда нужно сделать так, чтобы функция не вернула значение, а поменяла что-то в переданном ей контейнере, например, массиве или словаре. Для этого в C++ такие контейнеры должны передаваться по ссылке, а в Python передача по ссылке происходит сама собой.

Когда переменная передаётся по ссылке, то операции внутри вызванной функции будут её изменять:

C++

```
#include <vector>

typedef std::vector<int> V;

void f(V &v)
{
  v.push_back(1);
}

int main()
{
  V v;
  // v = {}
  f(v);
  // v = {1}
  f(v);
  // v = {1, 1}
}
```

Python

```
def f(v):
  v.append(1)

v = []
# v = []
f(v)
# v = [1]
f(v)
# v = [1, 1]
```

Иногда хочется передать переменную-массив так, чтобы функция могла с ней что-то поделать, но исходное значение при этом не изменилось. В C++ для этого используется передача по значению, тогда автоматически будет создаваться

временная копия переменной. В Python же придётся самостоятельно создавать копию или при вызове функции, или внутри неё.

**C++**

```
#include <vector>

typedef std::vector<int> V;

void f(V v)
{
  v.push_back(1);
}

int main()
{
  V v;
  // v = {}
  f(v);
  // v = {}
  f(v);
  // v = {}
}
```

**Python**

```
import copy

def f(v):
  v.append(1)

v = []
# v = []
f(copy.deepcopy(v))
# v = []
f(copy.deepcopy(v))
# v = []
```

## 2.8. Массивы

Мы будем использовать в своих программах два вида массивов: с постоянным размером и расширяющиеся. Точнее, это в C++ будет так. В Python массивы всегда будут расширяющимися. И называются они почему-то списками (`list`).

А в C++ для массивов постоянного размера иногда будем применять массивы в стиле C (например, `int a[10]`), а иногда контейнеры `array` (например linecppstd::array<int, 10> a). Второй вариант полезен, когда массив нужно копировать целиком или передавать в качестве параметра в функцию.

Массивы с изменяемым размером в C++ реализуются с помощью контейнера `vector`. Для него указывается только тип хранимых данных, а сами данные можно добавлять по мере выполнения программы.

**C++**

```
#include <array>
#include <vector>

// C-массив из 3 элементов
int a[3] = {1, 2, 3};
// array из 3 элементов
std::array<int, 3> b = {1, 2, 3};
// vector из 3 элементов
std::vector<int> c = {1, 2, 3};
```

**Python**

```
# пустой массив
a = []
# пустой массив
a = list()
# массив из 3 элементов
a = [1, 2, 3]
```

## 2.9. Словари

Словари нужны, когда обычных массивов недостаточно. Ведь в массиве к значениям мы обращаемся по индексам, а индексы — это непрерывный диапазон целых чисел, начиная с нуля.

Если же нужно множество значений, чьи индексы идут не подряд либо вообще не являются целыми числами, то понадобится словарь. Это такая структура данных, где хранимые значения выбираются с помощью соответствующих им ключей. Ключ — это аналог индекса элементов массива.

C++

```
#include <unordered_map>
#include <string>

int main()
{
  std::unordered_map<int, int> a;
  a[1] = 5;
  a[1000000] = 10;
  std::unordered_map<std::string, int> b;
  b["cat"] = 3;
  b["fish"] = 4;
}
```

Python

```
a = dict()
a[1] = 5
a[1000000] = 10
# () это то же самое, что dict()
b = {}
b['cat'] = 3
b['fish'] = 4
```

## 2.10. Генерация случайных чисел

Традиционно в программах на C++ случайные (точнее, псевдослучайные) числа генерируются с помощью функции `rand()`, которая появилась ещё в языке C. Она возвращает значения от 0 до предопределённой константы `RAND_MAX`. Чаще всего эта константа равняется 32767.

Когда нам будут нужны случайные числа, они должны находиться в некотором диапазоне, отличающемся от стандартного. Их можно получить, вычисляя остаток от деления случайного числа на длину требуемого диапазона. Например, для получения чисел от 0 до 9 берётся остаток от деления на 10.

Но этот способ неидеален. Если для каждого из чисел от 0 до 32737 посчитать остаток, то мы увидим, что остатков 0–7 будет на один больше, чем 8–9. Первых будет 3274, а вторых 3273. Разница невелика, поэтому в простых программах ею можно пренебречь.

В стандартной библиотеке C++ есть и более точные и современные функции (чтобы получалось равномерное распределение), но использовать их сложнее, поэтому в наших программах мы будем применять более старый и более простой способ.

Вызывая функцию `rand` много раз подряд, мы получим псевдослучайную последовательность чисел. Но при каждом запуске программы она будет одинаковой, из-за того что последовательность инициализируется в стандартной библиотеке. Чтобы начать её с действительно случайного числа, нужно инициализировать генератор чем-то, что постоянно меняется. Например, показаниями часов реального времени, вызвав функцию `srand` с параметром `time(NULL)`. Тогда при каждом запуске программы последовательности случайных чисел будут разными.

Чтобы получить случайные числа в Python, нужно использовать библиотеку `random`. В основе библиотеки лежит хороший генератор псевдослучайных последовательностей, для игр лучшего и не нужно, а более серьёзными применениями мы заниматься не будем.

В библиотеке `random` есть функции, чтобы генерировать случайные числа или выбирать какое-то случайное значение из списка. Например, для получения случайного целого числа в диапазоне от 0 до 9 нужно вызвать `random.randint(0, 9)`

А чтобы выбрать случайным образом одно значение из, к примеру, `True` или `False`, можно сделать так: `random.choice([True, False])`. Эта функция выбирает один случайный элемент из переданной ей последовательности.

По умолчанию библиотека использует системное время, чтобы инициализировать генератор. Поэтому вручную, как в C++, этого делать не придётся.

Чаще всего мы будем генерировать целые числа, поэтому вот простая программа, выводящая на экран одно случайное число.

C++

```
#include <stdlib.h>
#include <time.h>
#include <iostream>

int main()
{
  srand((int)time(NULL));
  std::cout << rand() << "\n";
}
```

Python

```
import random

print(random.randint(0, 32767))
```

## 2.11. Другие отличия C++ и Python

В программах, которые вы будете писать, читая книгу, будут встречаться и другие особенности языков. Вы узнаете, как использовать символы, строки, массивы, словари и познакомитесь с особенностями их применения в тех же главах, в которых будут разбираться программы.

Глава 3

# Начнём с простых игр

## 3.1. Игра «Бросание кубика»

Очень многие игры используют элементы случайности. Например, если на кубике выпало 6, монстр побеждён. Или нужно определить координаты на карте, куда программа спрячет сокровища от игрока. Ну и привычная всем колода карт обычно тоже перемешивается, чтобы карты шли в случайном порядке.

Но это всё только маленькие фрагменты больших игр. Поэтому начнём с программы, которая просто «бросает кубик». Обычный игровой кубик, с 6 гранями, на каждой из которых от 1 до 6 точек: ⚀ ⚁ ⚂ ⚃ ⚄ ⚅.

**Экран 1. Пример игры «Бросание кубика»**

```
Сделайте ставку: 4
Вы проиграли! Выпало число 5
```

### 3.1.1. Основа программы

По примеру вывода видно, что программа сравнивает два числа — ставку игрока и выпавшее на кубике значение. Чтобы в зависимости от результата сравнения выдавать разные сообщения, нужно использовать ветвление, оператор `if`.

C++

```
#include <iostream>

int main()
{
  int guess;
  std::cout
    << "Сделайте ставку: ";
  std::cin >> guess;
  int dice = 5;
  if (dice == guess)
  {
    std::cout << "Вы выиграли!\n";
  }
  else
  {
    std::cout << "Вы проиграли! "
      << "Выпало число "
      << dice << "\n";
  }
}
```

Python

```
guess = int(input('Сделайте ставку: '))
dice = 5

if guess == dice:
  print('Вы выиграли!')
else:
  print('Вы проиграли! '
    f'Выпало число {dice}')
```

Наберите эту программу и запустите несколько раз. Заметили, что всегда надо ставить на 5?

### 3.1.2. Имитируем игральный кубик

Сейчас на нашем виртуальном кубике выпадает всё время одно и то же. Под пример выше подходит, там тоже было 5, но в настоящей игре хотелось бы видеть всё время разные значения. К счастью, для этого в языках программирования есть встроенные библиотеки с функциями генерации случайных чисел.

Сначала эти библиотеки нужно подключить к программе:

C++

```
#include <stdlib.h>
#include <time.h>
```

Python

```
import random
```

А дальше сгенерировать случайное число, «выпавшее» на кубике.

C++

```
int dice = 1 + rand() % 6;
```

Python

```
dice = random.randint(1, 6)
```

Функция `rand()` возвращает число в диапазоне от 0 до библиотечной константы `RAND_MAX`. В стандарте языка её значение не зафиксировано, но чаще всего она равна 32767.

Поэтому чтобы получить число от 1 до 6, берём остаток от деления «случайного» числа на 6 (результат будет от 0 до 5) и прибавляем 1.

Функция `randint()` принимает два параметра — границы диапазона, в который будет попадать сгенерированное число. Для игрального кубика это 1 и 6.

⌨ | Теперь соедините всё вместе и попробуйте поиграть.

Если вы используете C++, осталась одна маленькая деталь. Функция `rand()` при каждом запуске программы будет возвращать одно и то же. Это происходит из-за того, что генератор случайных чисел инициализируется одним и тем же числом. Чтобы получались разные значения, можно в начале программы инициализировать генератор с помощью часов реального времени (время не стоит на месте, поэтому случайные числа всё время будут разными):

C++

```
srand((int)time(NULL));
```

### 3.1.3. Задания для самостоятельной работы

**1.** Переделайте программу «бросание кубика» в «казино». Пусть игрок делает несколько бросков. В случае успешного броска сумма на его счёте увеличивается, а в случае неудачи уменьшается.

**Листинг 3.1. dice.cpp**

```
#include <iostream>
#include <time.h>
#include <stdlib.h>

int main()
{
    srand((int)time(NULL));
    int guess;
    std::cout << "Сделайте ставку: ";
    std::cin >> guess;

    int dice = 1 + rand() % 6;
    if (dice == guess)
    {
        std::cout << "Вы выиграли! \n";
    }
    else
    {
        std::cout << "Вы проиграли! "
            << "Выпало число " << dice << "\n";
    }
}
```

**Листинг 3.2. dice.py**

```
#!/usr/bin/python3
import random

guess = int(input('Сделайте ставку: '))
dice = random.randint(1, 6)

if guess == dice:
    print('Вы выиграли!')
else:
    print(f'Вы проиграли! Выпало число {dice}')
```

## 3.2. Игра «Отгадай число»

Эта игра намного веселее, чем «Бросание кубика». Смотрите сами:

**Экран 1. Пример партии в «Отгадай число»**

```
Введите число: 50
Загаданное число больше.
Введите число: 75
Загаданное число меньше.
Введите число: 62
Загаданное число больше.
Введите число: 68
Загаданное число меньше.
Введите число: 65
Загаданное число больше.
Введите число: 66
Вы выиграли!
```

Во-первых, у игрока появилось несколько попыток, чтобы отгадать. А во-вторых, программа не только говорит «верно» или «неверно», но и даёт подсказки. Если текущая попытка не совпадает с загаданным числом, игрок получает сообщение о том, в какую сторону это число отличается.

### 3.2.1. Переделываем «Бросание кубика»

Итак, нам нужен цикл, чтобы игрок мог бы делать несколько попыток. Должен получиться примерно такой сценарий:

**Экран 2. Отгадай число на кубике**

```
Введите число: 1
Неправильно, попробуйте ещё.
Введите число: 2
Неправильно, попробуйте ещё.
Введите число: 3
Неправильно, попробуйте ещё.
Введите число: 4
Вы выиграли!
```

**C++**

```
int guess;
do
{
  std::cout << "Введите число: ";
  std::cin >> guess;
  ...
}
while (dice != guess);
```

**Python**

```
guess = 0
while dice != guess:
  guess = int(
    input("Введите число: "))
  ...
```

Переменная `guess` на каждой итерации получает новое значение, которое вводит пользователь. Можно было бы её прямо в цикле и объявить, но она используется в условии цикла, поэтому объявлена до него.

В Python нет цикла `do..while`, поэтому здесь мы используем обычный цикл `while`. Для этого необходимо инициализировать переменную `guess` значением, которое точно не совпадёт с выпавшим на кубике числом.

Добавьте этот цикл в игру с кубиком. Перед циклом нужно инициализировать `dice` случайным числом, а внутри цикла проверять, отгадал ли игрок, и выводить соответствующее сообщение.

### 3.2.2. Подсказки для игрока

Цикл позволяет игроку исправиться, если он не отгадал. И в случае кубика не так сложно добиться победы. Но будет ли интересно играть, если диапазон загадываемых чисел будет от 1 до 100?

**C++**

```
int secret = 1 + rand() % 100;
```

**Python**

```
secret = random.randint(1, 100)
```

Мы поменяли `dice` на `secret`, потому что игра больше не про кубик. Заодно изменили диапазон генерируемых значений. Теперь нужно добавить подсказки:

**C++**

```
if (secret > guess)
{
  std::cout
    << "Загаданное число больше.\n";
}
else if (secret < guess)
{
  std::cout
    << "Загаданное число меньше.\n";
}
else
{
  std::cout << "Вы выиграли!\n";
}
```

**Python**

```
if secret > guess:
  print("Загаданное число больше.")
elif secret < guess:
  print("Загаданное число меньше.")
else:
  print("Вы выиграли!")
```

Сравнивая введённое число с загаданным, программа отвечает игроку, что его число больше или меньше. Если же ни одно из неравенств не выполняется, выводится сообщение, что игрок угадал.

Закончите программу и поиграйте немного. Какое наибольшее число попыток вам потребовалось, чтобы отгадать число? Какая стратегия будет оптимальной?

Добавьте в программу возможность проиграть, ограничив число попыток отгадывания.

**Листинг 3.3. guessnumber.cpp**

```
#include <iostream>
#include <time.h>
#include <stdlib.h>

int main()
{
    srand((int)time(NULL));
    int guess;
    int secret = 1 + rand() % 100;

    do
    {
        std::cout << "Введите число: ";
        std::cin >> guess;
        if (secret > guess)
        {
            std::cout << "Загаданное число больше. \n";
        }
        else if (secret < guess)
        {
            std::cout << "Загаданное число меньше. \n";
        }
        else
        {
            std::cout << "Вы выиграли! \n";
        }
    }
    while (secret != guess);
}
```

**Листинг 3.4. guessnumber.py**

```
#!/usr/bin/python3
import random

secret = random.randint(1, 100)
guess = 0
while guess != secret:
    guess = int(input("Введите число: "))
    if secret > guess:
        print("Загаданное число больше.")
    elif secret < guess:
        print("Загаданное число меньше.")
    else:
        print("Вы выиграли!")
```

## 3.3. Игра «23 спички»

На столе лежат 23 спички. На каждом ходу можно взять 1, 2 или 3 спички. Вы с компьютером ходите по очереди. Проигрывает тот, кто берёт последнюю спичку.

Компьютер даёт вам возможность ходить первому, поэтому вы сможете выиграть, если хорошенько подумаете.

В таком виде эта игра описана в книге [1], но вы можете её знать под названием «игра Баше» [5] или «игра с камушками» [10].

Конечно же, мы хотим сделать такую программу, которая сможет обыгрывать всех, кому мы её показываем. Чтобы сделать это, надо проанализировать позиции в игре с помощью программы [4] или математики [10].

Чтобы упростить итоговую игровую программу, проведём подготовку на бумаге. Сначала посмотрим на пример партии[1]:

**Экран 1. Компьютер побеждает в «23 спички»**

```
На столе лежит 23 спичек.
Сколько вы возьмёте? 3
Вы взяли 3, осталось 20 спичек.
Я беру 3, осталось 17 спичек.
Сколько вы возьмёте? 2
Вы взяли 2, осталось 15 спичек.
Я беру 2, осталось 13 спичек.
Сколько вы возьмёте? 3
Вы взяли 3, осталось 10 спичек.
Я беру 1, осталось 9 спичек.
Сколько вы возьмёте? 1
Вы взяли 1, осталось 8 спичек.
Я беру 3, осталось 5 спичек.
Сколько вы возьмёте? 3
Вы взяли 3, осталось 2 спичек.
Я беру 1, осталось 1 спичек.
Сколько вы возьмёте? 1
Вы взяли 1, осталось 0 спичек.
Последняя спичка ваша, я выиграл!
```

[1] Программа не всегда верно пишет окончание у слова «спичек», так как для выбора правильного окончания пришлось бы написать дополнительный код, что отвлекло бы нас от сути алгоритма и увеличило бы листинг.

Наверное, вам уже очевидно, что если один из игроков оказался в ситуации с единственной спичкой на столе, то он проиграл, потому что именно эту последнюю спичку ему придётся забрать. Если же спичек хотя бы две, то можно сделать минимум один ход, и игра продолжится. Конечно же, когда на столе от 2 до 4 спичек, надо оставить сопернику только одну. Как мы уже выяснили, в этом случае он проиграет.

Теперь следующий шаг. Кто выиграет, если на столе 5 спичек? Сколько бы ни взял тот, кому выпала эта позиция, он оставит противнику от 2 до 4 спичек. Как мы уже знаем, тогда этот противник сможет выиграть.

Если продолжить эти рассуждения, можно прийти к тому, что проигрывает тот, кому досталось $4 \cdot N + 1$ спичек. Тогда суммарно за ходы обоих игроков можно уменьшить $N$ на единицу. Это делается так: если один игрок взял $x$ спичек, то второй берёт $4 - x$. В итоге тот же игрок, который попал в исходную позицию с $4 \cdot N + 1$ спичками, придёт и в последнюю позицию с одной спичкой (это соответствует $N = 0$).

Имея эти знания, можно перейти к программированию.

### 3.3.1. Почти готовая игра

В этой игре (как и в большинстве других) игроки делают по несколько ходов, значит нам понадобится цикл. Игра завершается, если спичек не осталось. Поэтому цикл должен выглядеть так:

**C++**

```
#include <iostream>

int main()
{
  int matches = 23;
  std::cout << "На столе лежит "
    << matches << " спичек.\n";
  while (matches > 0)
  {
    ...
  }
}
```

**Python**

```
matches = 23
print('На столе лежит '
  f'{matches} спичек.')
while matches > 0:
  ...
```

Просто пустой цикл. Не спешите запускать эту программу, потому что она зависнет. Ведь переменная `matches` в цикле не меняется, поэтому он будет выполняться вечно.

Игроки ходят поочерёдно. Значит, в цикле сначала должен быть код для обработки хода человека (он ходит первым), а потом код для хода компьютера.

Начнём с обработки хода человека. Нужно получить от него число спичек, убедиться, что ход корректный, уменьшить число спичек на столе. И ещё нужно

проверить, не закончилась ли игра, потому что тогда компьютеру будет не нужно делать ход.

**C++**

```
int move;
do
{
  std::cout << "Сколько "
    "спичек вы возьмёте? ";
  std::cin >> move;
}
while (move < 1
  || move > matches
  || move > 3);
matches -= move;
std::cout << "Вы взяли " << move
  << ", осталось " << matches
  << " спичек.\n";
if (matches == 0)
{
  std::cout << "Вы проиграли.\n";
  break;
}
```

**Python**

```
move = 0
while move < 1
  or move > matches
  or move > 3:
  move = int(input('Сколько '
    'спичек вы возьмёте? '))
matches -= move
print(f'Вы взяли {move}, '
  f'осталось {matches} спичек')
if matches == 0:
  print('Вы проиграли.')
  break
```

Теперь программу можно запустить. Компьютер ходов не делает, поэтому и проиграть не сможет. Но можно проверить, что все условия срабатывают правильно: нельзя совсем не брать спичек и нельзя брать слишком много.

### 3.3.2. Учим компьютер играть

Для ходов человека мы вводили число с клавиатуры и проверяли его. Ходы компьютера проверять не нужно, поэтому код будет немного проще. Пусть сначала программа всегда «берёт» одну спичку:

**C++**

```
move = 1;
matches -= move;
std::cout << "Я беру " << move
  << ", осталось " << matches
  << " спичек.\n";
if (matches == 0)
{
  std::cout << "Вы выиграли.\n";
  break;
}
```

**Python**

```
move = 1
matches -= move
print(f'Я беру {move}, '
  f'осталось {matches} спичек.')
if matches == 0:
  print('Вы выиграли.')
  break
```

Уже можно играть. Насладитесь победой над программой, и мы продолжим.

После того как программа научилась играть, можно научить её думать. Выигрышную стратегию мы разбирали раньше, надо брать столько спичек, чтобы их сумма с последним ходом соперника равнялась 4.

Заменим строчку `move = 1` на следующий фрагмент:

C++
```
move = 4 - move;
if (matches <= move)
{
  move = 1;
}
```

Python
```
move = 4 - move
if matches <= move:
  move = 1
```

Получается уже более интеллектуальный алгоритм. Проверьте его на практике. Подумайте, будет ли он правильно работать, когда на столе в самом начале 23 спички. Если нет, то сколько спичек должно быть?

### 3.3.3. Доделываем выбор хода

Вы наверное уже заметили, что этот алгоритм не всегда может победить. А если точнее, он побеждает, только если в конце партии игрок специально поддаётся.

Мы уже выясняли, что проиграет тот, кто окажется в позиции 21, 17, 13 и так далее. Но если игрок сделает ход 1, то компьютер после этого возьмёт 3 спички. Теперь игроку останется 19 спичек. А вот если бы программа сделала ход 1, то попала бы в позицию 21. Из неё уже выиграть будет легко.

Поэтому надо не просто дополнять ход человека до четырёх, а сводить позицию к $4 \cdot N + 1$ спичкам. Это можно сделать так:

C++
```
move = (matches - 1) % 4;
if (move == 0)
{
  move = 1;
}
```

Python
```
move = (matches - 1) % 4
if move == 0:
  move = 1
```

Условие проверки на равенство нулю понадобилось, чтобы сделать корректный ход, когда человек нас раскусил и привёл в проигрышную позицию. Тогда формула в первой строке не сработает.

Закончите программу и поиграйте с ней. Попробуйте заводить её в проигрышную позицию или поддавайтесь, чтобы позволить выиграть.

### 3.3.4. Задания для самостоятельной работы

**1.** Поменяйте программу, чтобы игра начиналась не с 23 спичек, а со случайного количества от 10 до 100.

**2.** Ещё раз доработайте программу, чтобы компьютер ходил первым (если он вытащит «короткую спичку»).

**3.** Добавьте несколько уровней сложности. На простых уровнях компьютер может иногда ходить неоптимально (то есть «ошибаться»).

**Листинг 3.5. 23matches.cpp**

```
#include <iostream>

int main()
{
  int matches = 23;
  std::cout << "На столе лежит " << matches << " спичек.\n";
  while (matches > 0)
  {
    int move;
    do
    {
      std::cout << "Сколько спичек вы возьмёте? ";
      std::cin >> move;
    }
    while (move < 1 || move > matches || move > 3);
    matches -= move;
    std::cout << "Вы взяли " << move << ", осталось " << matches << "
    ↪ спичек.\n";
    if (matches == 0)
    {
      std::cout << "Последняя спичка ваша, я выиграл!\n";
      break;
    }
    move = (matches - 1) % 4;
    if (move == 0)
    {
      move = 1;
    }
    matches -= move;
    std::cout << "Я беру " << move << ", осталось " << matches << "
    ↪ спичек.\n";
    if (matches == 0)
    {
      std::cout << "Сегодня вам удалось победить. Посмотрим, что будет
      ↪ в следующий раз!\n";
      break;
    }
  }
}
```

**Листинг 3.6. 23matches.py**

```
#!/usr/bin/python3
matches = 23
print(f'На столе лежит {matches} спичек.')
while matches > 0:
  move = 0
  while move < 1 or move > matches or move > 3:
    move = int(input('Сколько спичек вы возьмёте? '))
  matches -= move
  print(f'Вы взяли {move}, осталось {matches} спичек')
  if matches == 0:
    print('Последняя спичка ваша, я выиграл!')
    break
  move = (matches - 1) % 4
  if move == 0:
    move = 1
  matches -= move
  print(f'Я беру {move}, осталось {matches} спичек.')
  if matches == 0:
    print('Сегодня вам удалось победить. Посмотрим, что будет в
    ↪ следующий раз!')
    break
```

Глава 4

# Изучаем структуру игровой программы

## 4.1. Игра «Хаммурапи»

Игра «Хаммурапи» — один из первых компьютерных экономических симуляторов. Придумали её в 1968 году, а широко известной игра стала после публикации её версии на языке Бейсик в 1973 году [1]. Позже появилось множество подобных игр, но сегодня мы попробуем повторить классический вариант. Заодно разберёмся, как устроены большинство игр, новых и старых.

Игра состоит из десяти раундов, в которых игрок, как вавилонский царь Хаммурапи, определяет, сколько зерна посеять, сколько потратить на пропитание народа, на покупку новой земли. Кроме того, происходят разные неприятности вроде неурожая, нашествия крыс или чумы.

Попробуем погрузиться в суровый древний мир. Конечно, мы не будем программировать на Бейсике. Но неожиданности всё же возникнут. В этот раз начнём с целой программы, а потом разберёмся, из чего она состоит.

**Экран 1. Начало игры с программой «Хаммурапи»**

```
                    ХАММУРАПИ

Попробуйте управлять древним шумерским государством
в течение десяти лет.

Хаммурапи, сообщаю Вам, в прошлом 1 году
0 людей умерли от голода, 5 человек прибыли в город.
Всего в городе живёт 100 человек.
Город владеет 1000 акрами земли.
Вы собрали 3 бушелей зерна с акра.
Крысы съели 200 бушелей зерна.
Сейчас у Вас 2800 бушелей в хранилище.

Стоимость земли сейчас составляет 27 бушелей за акр.
Сколько акров Вы хотите купить/продать? 0
Сколько бушелей зерна Вы потратите, чтобы накормить людей? 2000
Сколько акров земли Вы хотите засеять? 800

Хаммурапи, сообщаю Вам, в прошлом 2 году
0 людей умерли от голода, 5 человек прибыли в город.
Всего в городе живёт 105 человек.
Город владеет 1000 акрами земли.
......
```

**Листинг 4.1. hammurabi.cpp**

```
#include <iostream>
#include <stdlib.h>

int harvest_total, harvest, food;
int year, population, land, rats, starved;
int died_total, grain, percent_died, people_came;

void quit()
{
  std::cout << "\n\n\nДо встречи. \n\n";
  exit(0);
}

void printNotEnoughGrain()
{
  std::cout << "Подумайте ещё. У Вас всего "
    << grain << " бушелей зерна. \n";
}

void printNotEnoughLand()
{
  std::cout << "Подумайте ещё, у Вас есть только "
    << land << " акров земли. \n";
}

void endGameBad()
{
  std::cout << "Ваше правление было ужасным, \n"
    "Вас объявили национальным предателем и изгнали из
    ↪ резиденции!!! \n";
  quit();
}

void tradeLand()
{
  int cost = 17 + rand() % 11;
  std::cout << "Стоимость земли сейчас составляет "
    << cost << " бушелей за акр. \n";
  int buysell = 0;
  while (true)
```

```
  {
    std::cout << "Сколько акров Вы хотите купить/продать? ";
    std::cin >> buysell;
    if (buysell < 0 && -buysell > land)
    {
      printNotEnoughLand();
      continue;
    }
    if (buysell > 0 && cost * buysell > grain)
    {
      printNotEnoughGrain();
      continue;
    }
    break;
  }

  land = land + buysell;
  grain = grain - cost * buysell;
}

void feedPeople()
{
  std::cout << "\n";
  while (true)
  {
    std::cout << "Сколько бушелей зерна Вы потратите, чтобы накормить
    ↪ людей? ";
    std::cin >> food;
    if (food < 0)
    {
      continue;
    }
    if (food <= grain)
    {
      break;
    }
    printNotEnoughGrain();
  }
  grain = grain - food;
}
```

```
void plantSeeds()
{
  std::cout << "\n";
  int plant = 0;
  while (true)
  {
    std::cout << "Сколько акров земли Вы хотите засеять? ";
    std::cin >> plant;
    if (plant < 0)
    {
      continue;
    }
    if (plant > land)
    {
      printNotEnoughLand();
      continue;
    }
    if (plant / 2 > grain)
    {
      printNotEnoughGrain();
      continue;
    }
    if (plant > 10 * population)
    {
      std::cout << "Но у Вас только " << population << " человек для
      ↪ работы на полях! \n";
      continue;
    }
    break;
  }

  grain = grain - plant / 2;
  harvest = rand() % 6 + 1;
  harvest_total = plant * harvest;
}

void ratsInvasion()
{
  rats = 0;
  int c = rand() % 6 + 1;
  if (c % 2 == 0)
```

```
  {
    rats = grain / c;
  }
  grain -= rats;
}

void harvestGrain()
{
  grain = grain + harvest_total;
}

void changePopulation()
{
  people_came = (rand() % 6 + 1) * (20 * land + grain) / population /
  ↪  100 + 1;

  starved = population - food / 20;
  if (starved <= 0)
  {
    starved = 0;
  }
  else
  {
    if (starved > 0.45 * population)
    {
      std::cout << "\nЗа год от голода умерло " << starved << "
      ↪  человек!!!\n";
      endGameBad();
    }
    percent_died = ((year - 1) * percent_died + starved * 100 /
    ↪  population) / year;
    population = population - starved;
    died_total = died_total + starved;
  }
  population = population + people_came;
}

void plague()
{
  if (year > 1 && rand() % 100 < 15)
  {
```

```
    population -= population / 2;
    std::cout << "\nЭпидемия чумы! Половина населения умерла.\n";
  }
}

void report()
{
  std::cout << "\nХаммурапи, сообщаю Вам,\n";
  std::cout << "в прошлом " << year << " году " << starved << " людей
  ↪ умерли от голода, "
    << people_came << " человек прибыли в город.\n";
  std::cout << "Всего в городе живёт " << population << "
  ↪ человек.\n";
  std::cout << "Город владеет " << land << " акрами земли.\n";
  std::cout << "Вы собрали " << harvest << " бушелей зерна с
  ↪ акра.\n";
  std::cout << "Крысы съели " << rats << " бушелей зерна.\n";
  std::cout << "Сейчас у Вас " << grain << " бушелей в
  ↪ хранилище.\n\n";
}

void final()
{
  std::cout << "За прошедшие десять лет в среднем " << percent_died
  ↪ << " процентов\n"
    << "населения в год умирало от голода. Всего умерло "
    << died_total << " человек!!!\n";
  int L = land / population;
  std::cout << "В начале правления у Вас было 10 акров земли на
  ↪ человека,\n"
    "а в конце стало " << L << " акров на человека.\n\n";
  if (percent_died > 33 || L < 7)
  {
    endGameBad();
  }
  else if (percent_died > 10 || L < 9)
  {
    std::cout << "Своим жёстким руководством Вы переплюнули Ивана
    ↪ Грозного.\n"
      << "Оставшиеся люди будут долго ненавидеть Вас!!!\n";
  }
```

```
  else if (percent_died > 3 || L < 10)
  {
    std::cout << "Ваше правление было не самым плохим. \nХотя "
      << (int)(population * 0.8 * rand() / RAND_MAX) << " человек \n"
      << "предпочли бы, чтобы Ваша жизнь закончилась в результате
      ↪  покушения. \n";
  }
  else
  {
    std::cout << "Фантастический результат!!! Даже Карл Великий и \n"
      << "Пётр Первый не смогли бы лучше! \n";
  }
}

int main()
{
  std::cout << "\t\t\t\tХАММУРАПИ \n";
  std::cout << "\n\n\nПопробуйте управлять древним шумерским
  ↪  государством \n"
    "в течение десяти лет. \n\n";
  population = 100;
  rats = 200;
  harvest_total = 3000;
  grain = harvest_total - rats;
  harvest = 3;
  land = harvest_total / harvest;
  people_came = 5;
  while (true)
  {
    year = year + 1;
    plague();
    report();
    if (year == 11)
    {
      break;
    }
    tradeLand();
    feedPeople();
    plantSeeds();
    ratsInvasion();
    harvestGrain();
```

```
    changePopulation();
  }

  final();
  quit();
}
```

**Листинг 4.2. hammurabi.py**

```
#!/usr/bin/python3
import random
import sys

def quit():
    print('\n\n\nДо встречи.\n')
    sys.exit()

def printNotEnoughGrain():
    print(f'Подумайте ещё. У Вас всего {grain} бушелей зерна.')

def printNotEnoughLand():
    print(f'Подумайте ещё, у Вас есть только {land} акров земли.')

def endGameBad():
    print('Ваше правление было ужасным,\n'
        'Вас объявили национальным предателем и изгнали из
        ↪ резиденции!!!')
    quit()

def tradeLand():
    global land, grain
    cost = random.randint(17, 27)
    print(f'Стоимость земли сейчас составляет {cost} бушелей за
    ↪ акр.')
    buysell = 0
    while True:
        buysell = int(input('Сколько акров вы хотите купить/продать?
        ↪ '))
        if buysell < 0 and -buysell > land:
            printNotEnoughLand()
            continue
```

```
        if buysell > 0 and cost * buysell > grain:
            printNotEnoughGrain()
            continue
        break
    land = land + buysell
    grain = grain - cost * buysell

def feedPeople():
    global food, grain
    print('')
    while True:
        food = int(input('Сколько бушелей зерна Вы потратите, чтобы
        ↪ накормить людей? '))
        if food < 0:
            continue
        if food <= grain:
            break
        printNotEnoughGrain()
    grain = grain - food

def plantSeeds():
    global grain, harvest, harvest_total
    print('')
    plant = 0
    while True:
        plant = int(input('Сколько акров земли Вы хотите засеять? '))
        if plant < 0:
            continue
        if plant > land:
            printNotEnoughLand()
            continue
        if plant / 2 > grain:
            printNotEnoughGrain()
            continue
        if plant > 10 * population:
            print(f'Но у Вас только {population} человек для работы
            ↪ на полях!')
            continue
        break
    grain = grain - plant // 2
    harvest = random.randint(1, 6)
```

```
    harvest_total = plant * harvest

def ratsInvasion():
    global rats, grain
    rats = 0
    c = random.randint(1, 6)
    if c % 2 == 0:
        rats = grain // c
    grain -= rats

def harvestGrain():
    global grain
    grain = grain + harvest_total

def changePopulation():
    global people_came, population, starved, percent_died, died_total
    people_came = random.randint(1, 6) * (20 * land + grain) //
    ↪ population // 100 + 1
    starved = population - food // 20
    if starved <= 0:
        starved = 0
    else:
        if starved > 0.45 * population:
            print(f'\nЗа год от голода умерло {starved} человек!!!')
            endGameBad()
        percent_died = ((year - 1) * percent_died + starved * 100 //
        ↪ population) // year
        population = population - starved
        died_total = died_total + starved
    population = population + people_came

def plague():
    global year, population
    if year > 1 and random.randint(0, 99) < 15:
        population -= population // 2
        print('\nЭпидемия чумы! Половина населения умерла.')

def report():
    print('\nХаммурапи, сообщаю Вам,')
    print(f'в прошлом {year} году {starved} людей умерли от голода,
    ↪ ')
```

```
        f'{people_came} человек прибыли в город.')
    print(f'Всего в городе живёт {population} человек.')
    print(f'Город владеет {land} акрами земли.')
    print(f'Вы собрали {harvest} бушелей зерна с акра.')
    print(f'Крысы съели {rats} бушелей зерна.')
    print(f'Сейчас у Вас {grain} бушелей в хранилище.\n')

def final():
    print(f'За прошедшие десять лет в среднем {percent_died}
    ↪ процентов')
    print('населения в год умирало от голода. Всего умерло '
        f'{died_total} человек!!!')
    L = land // population
    print('В начале правления у Вас было 10 акров земли на человека,')
    print(f'а в конце стало {L} акров на человека.\n')
    if percent_died > 33 or L < 7:
        endGameBad()
    elif percent_died > 10 or L < 9:
        print('Своим жёстким руководством Вы переплюнули Ивана
        ↪ Грозного.')
        print('Оставшиеся люди будут долго ненавидеть Вас!!!')
    elif percent_died > 3 or L < 10:
        print('Ваше правление было не самым плохим.')
        print(f'Хотя {random.randint(0, int(population * 0.8))}
        ↪ человек')
        print('предпочли бы, чтобы Ваша жизнь закончилась в результате
        ↪ покушения.')
    else:
        print('Фантастический результат!!! Даже Карл Великий и')
        print('Пётр Первый не смогли бы лучше!')

print('\t\t\t\tХАММУРАПИ')
print('\n\n\nПопробуйте управлять древним шумерским государством')
print('в течение десяти лет.\n')

year = 0
starved = 0
population = 100
rats = 200
harvest_total = 3000
grain = harvest_total - rats
```

```
harvest = 3
land = harvest_total // harvest
people_came = 5
percent_died = 0
died_total = 0
while True:
    year = year + 1
    plague()
    report()
    if year == 11:
        break
    tradeLand()
    feedPeople()
    plantSeeds()
    ratsInvasion()
    harvestGrain()
    changePopulation()
final()
quit()
```

### 4.1.1. Функция `main`

Главная функция программы (функция `main` в C++ или код вне функции в Python) состоит из следующих частей:

- Вывод приглашения и кратких правил игры.
- Инициализация переменных.
- Главный игровой цикл.
- Вывод итогов игры.

Первая часть не требует особых комментариев, а остальные мы рассмотрим подробнее.

### 4.1.2. Инициализация переменных

Так как код игры разбит на несколько функций, они должны как-то обмениваться данными. В этой игре все важные данные хранятся в глобальных переменных, поэтому ни одна функция не имеет параметров, а вместо этого напрямую обращается к переменным.

Каждая переменная должна инициализироваться перед использованием. К примеру, `population`, то есть население, принимает значение 100. А `grain` (зерно в хранилищах) вычисляется как общее количество зерна `harvest_total` за вы-

четом съеденного крысами `rats`. Почему бы просто не записать в `grain` сразу нужное значение? Потому что перед первым ходом тоже выводится отчёт, где говорится, в том числе, и про крыс.

C++

```
int harvest_total, harvest, food;
int year, population, land, rats,
↪ starved;
int died_total, grain, percent_died,
↪ people_came;
population = 100;
rats = 200;
harvest_total = 3000;
grain = harvest_total - rats;
harvest = 3;
land = harvest_total / harvest;
people_came = 5;
```

Python

```
year = 0
starved = 0
population = 100
rats = 200
harvest_total = 3000
grain = harvest_total - rats
harvest = 3
land = harvest_total // harvest
people_came = 5
percent_died = 0
died_total = 0
```

Вот назначение остальных переменных:

- `year` — номер года;
- `starved` — число умерших от голода в прошлом году;
- `harvest` — сколько зерна принёс один акр земли;
- `land` — число акров земли во владении;
- `people_came` — число людей, прибывших в город в прошлом году;
- `percent_died` — средний процент умерших от голода за все годы;
- `died_total` — сколько всего человек умерло от голода.

В Python, из-за того что используются глобальные переменные, приходится явно указывать, что мы обращаемся именно к ним, если нужно записать такую переменную внутри функции. Для этого используется директива `global`.

Python

```
def feedPeople():
    global food, grain
    ...
    grain = grain - food
```

### 4.1.3. Главный игровой цикл

Все игры (кроме игры с однократным бросанием кубика) содержат в себе один главный цикл. В нём игрок делает ход, вводит какие-то данные, стреляет в монстров, а программа отвечает на это, меняет картинку на экране, выводит сообщения... И так раз за разом.

В цикле происходят такие события:

- Чума. Это очень суровая игра. Каждый ход начинается с подсчёта шансов на эпидемию чумы. За это отвечает функция `plague`.
- Вывод отчёта о ресурсах государства: какое население, сколько зерна, сколько земли и т.п. Это делает функция `report`.
- Торговля землёй. Каждый год можно купить больше земли или продать часть имеющейся. Функция `tradeLand`.
- Производство пищи. Часть зерна можно потратить, чтобы кормить население. Если еды будет слишком мало, люди начнут умирать от голода. Функция `feedPeople`.
- Засев полей. Оставшееся зерно можно использовать, чтобы вырастить новый урожай. Для этого также нужны работники. Поэтому если людей мало, всю землю обрабатывать не получится. Функция `plantSeeds`.
- Нашествие крыс. Если не потратить всё зерно, то оставшуюся его часть могут съесть крысы. Функция `ratsInvasion`.
- Сбор урожая. Это событие отделено от обработки земли, потому что если «пополнить» запасы зерна до нашествия крыс, то они нападут уже на новый урожай, а не на запасы, как было задумано. Функция `harvestGrain`.
- Увеличение популяции жителей. В городе появляются новые жители, если есть достаточное количество земли и зерна. Независимо от этого часть населения может умереть от голода. Функция `changePopulation`.

Итак, ход игрока заключается в выборе того, на что тратить зерно. Зерна чаще всего не хватает на всё, так что если плохо планировать действия, то либо люди начнут умирать, либо на следующий год зерна будет ещё меньше.

Игра заканчивается, если удалось выдержать 10 лет, либо когда от голода умирает слишком много людей.

### 4.1.4. Вывод итогов игры

Игра может закончиться и «благополучно», то есть через 10 ходов. За это отвечает проверка в середине цикла:

**C++**

```
while (true)
{
    year = year + 1;
    ...
    if (year == 11)
    {
        break;
    }
    ...
}
```

**Python**

```
while True:
    year = year + 1
    ...
    if year == 11:
        break
    ...
```

В условие цикла эта проверка не вынесена, потому что сообщение со статистикой необходимо выводить и в начале, и в конце игры. Поэтому часть тела цикла выполняется и перед первым ходом, и после последнего. В качестве альтернативы можно было бы продублировать вызов функции `report`.

После выхода из цикла вызывается функция `final`. В ней есть несколько вариантов сообщений об окончании игры.

В сообщении, выводимом по итогу игры, учитывается средний процент умерших от голода, а также количество земли, приходящееся на одного человека.

Процент умерших каждый год пересчитывается по такой формуле:

C++
```
percent_died = ((year - 1) * percent_died
↪  + starved * 100 / population) / year;
```

Python
```
percent_died = ((year - 1) * percent_died
↪  + starved * 100 // population) //
↪  year
```

В Python целочисленное деление выполняется с помощью оператора `//`. Если использовать обычный оператор деления, то может получиться дробное число. Это могло бы усложнить некоторые из вычислений и сравнений, поэтому мы используем целочисленное деление. Заодно работа Python-программы будет точно соответствовать коду на C++.

С количеством акров земли на человека всё намного проще:

C++
```
int L = land / population;
```

Python
```
L = land // population;
```

Отличный результат – это средний процент умерших не больше 3 и хотя бы 10 акров земли на человека.

### 4.1.5. Покупка и продажа земли

Покупкой и продажей земли управляет функция `tradeLand`. Никаких денег в игре нет, их функцию выполняет зерно. Поэтому стоимость акра[1] земли измеряется в бушелях[2] зерна.

Стоимость акра определяется каждый год случайным образом, чтобы потом использоваться как при покупке, так и при продаже.

C++
```
int cost = 17 + rand() % 11;
std::cout << "Стоимость земли сейчас
↪  составляет "
    << cost << " бушелей за акр.\n";
```

Python
```
cost = random.randint(17, 27)
```

[1]Изначально это площадь земли, обрабатываемая в день одним крестьянином с одним волом. Сейчас один акр — это примерно 4047 квадратных метров.

[2]Один бушель — это примерно 36 литров.

Чтобы упростить ввод, покупка и продажа управляется одним числом, вводимым с клавиатуры. Если игрок ввёл положительное число, то это покупка, а если отрицательное, то продажа.

C++

```
int buysell = 0;
std::cout << "Сколько акров Вы хотите
↪ купить/продать? ";
std::cin >> buysell;
...
land = land + buysell;
grain = grain - cost * buysell;
```

Python

```
buysell = 0
buysell = int(input('Сколько акров Вы
↪ хотите купить/продать? '))
...
land = land + buysell
grain = grain - cost * buysell
```

А что если игрок попытается потратить больше зерна, чем у него есть? Нужно попросить его ввести новое число. Но при этом было бы неплохо выдавать осмысленное сообщение: не хватает зерна на покупку, либо не хватает земли на продажу. Поэтому напишем «бесконечный» цикл с оператором `break`, который выполняется, если не была допущена ошибка при вводе:

C++

```
int buysell = 0;
while (true)
{
  std::cout << "Сколько акров Вы хотите
  ↪ купить/продать? ";
  std::cin >> buysell;
  if (buysell < 0 && -buysell > land)
  {
    printNotEnoughLand();
    continue;
  }
  if (buysell > 0 && cost * buysell >
  ↪ grain)
  {
    printNotEnoughGrain();
    continue;
  }
  break;
}
```

Python

```
buysell = 0
while True:
  buysell = int(input('Сколько акров Вы
  ↪ хотите купить/продать? '))
  if buysell < 0 and -buysell > land:
    printNotEnoughLand()
    continue
  if buysell > 0 and cost * buysell >
  ↪ grain:
    printNotEnoughGrain()
    continue
  break
```

### 4.1.6. Игровая позиция

Во фрагментах кода выше используются глобальные переменные, которые мы уже обсуждали раньше. Эти переменные используются для хранения параметров, меняющихся в течение игры. Их совокупность называется игровой позицией.

В отношении этой игры слово «позиция» звучит странно, но этот термин происходит от игр с фишками или фигурами. Например, в шахматах игровой позицией будет положение всех фигур на доске.

Игрок делает ходы, руководствуясь именно игровой позицией. Ставить ли шах, покупать ли новую землю, строить ли зиккурат. Не вся информация об игровой позиции может быть доступна игроку (например, он может не видеть порядок карт в колоде), но всю её где-то надо хранить.

Понятие «игровая позиция» нам ещё неоднократно встретится, потому что именно на её основе будет выбирать ходы тот искусственный интеллект, который мы будем создавать для других игр.

В игре «Хаммурапи» игровая позиция состоит из нескольких целочисленных переменных, описывающих текущее состояние города-государства: количество земли для обработки, объём зерна в хранилищах, население.

Так как действия и события разнесены по разным функциям, часть промежуточных данных также сохраняется в глобальных переменных. Например, количество съеденного крысами зерна сохраняется в переменную `rats`, чтобы потом показать это число в ежегодном отчёте.

Инициализируется игровая позиция в начале программы, а по ходу игры она модифицируется и используется в функциях, составляющих главный игровой цикл.

### 4.1.7. Зерно: хлеб и семена

Очень много всего в игре связано с зерном. В первую очередь его можно использовать для еды, за это отвечает функция `feedPeople`.

Структура этой функции похожа на уже рассмотренную `tradeLand`. Тут игрок вводит количество зерна, которое он бы хотел потратить на еду. Если число больше нуля и в хранилищах есть такое количество зерна, то цикл завершается. Иначе придётся вводить число заново.

Когда наконец-то введено корректное число, количество зерна в хранилище уменьшается:

C++
```
grain = grain - food;
```

Python
```
grain = grain - food
```

Переменная `food` глобальная, потому что она позже используется для подсчёта числа голодающих.

Следующая функция, `plantSeeds`, нужна для определения того, сколько зерна будет посеяно. Общая её структура такая же, как и у предыдущей: цикл, который продолжается до тех пор, пока игрок не введёт корректное значение.

Но проверок здесь уже больше: нельзя засеять больше земли, чем имеется во владении; можно использовать только оставшееся зерно; должно быть достаточное количество населения для работы в полях.

C++

```
if (plant > land)
{
    printNotEnoughLand();
    continue;
}
if (plant / 2 > grain)
{
    printNotEnoughGrain();
    continue;
}
if (plant > 10 * population)
{
    std::cout << "Но у Вас только " <<
    ↪ population << " человек для
    ↪ работы на полях!\n";
    continue;
}
```

Python

```
if plant > land:
    printNotEnoughLand()
    continue
if plant / 2 > grain:
    printNotEnoughGrain()
    continue
if plant > 10 * population:
    print(f'Но у Вас только {population}
    ↪ человек для работы на полях!')
    continue
```

Урожайность каждый год определяется случайным образом, от 1 до 6 бушелей на акр. В конце функция подсчитывает, сколько зерна удалось собрать.

C++

```
grain = grain - plant / 2;
harvest = rand() % 6 + 1;
harvest_total = plant * harvest;
```

Python

```
grain = grain - plant // 2
harvest = random.randint(1, 6)
harvest_total = plant * harvest
```

Однако к общему количеству зерна это число пока не прибавляется, потому что на неиспользованный остаток совершают набег крысы. А уже после этого в функции `harvestGrain` происходит «сбор урожая»:

C++

```
grain = grain + harvest_total;
```

Python

```
grain = grain + harvest_total
```

### 4.1.8. Случайные события

Случайных величин в игре не так уж мало: стоимость земли, размер урожая, приток новых людей. Но есть и целые события, которые происходят не каждый год. Это нашествие крыс и чума.

Функция с крысами имеет следующий вид:

C++

```
void ratsInvasion()
{
    rats = 0;
    int c = rand() % 6 + 1;
    if (c % 2 == 0)
    {
        rats = grain / c;
    }
    grain -= rats;
}
```

Python

```
def ratsInvasion():
    global rats, grain
    rats = 0
    c = random.randint(1, 6)
    if c % 2 == 0:
        rats = grain // c
    grain -= rats
```

Здесь генерируется случайное значение от 1 до 6. И крысы съедают долю зерна, равную этому значению, но только если оно чётное. То есть, в среднем, крысы нападают каждый второй год.

Чума обрабатывается в процедуре `plague`:

C++

```
void plague()
{
  if (year > 1 && rand() % 100 < 15)
  {
    population -= population / 2;
    std::cout << "Эпидемия чумы! Половина
    ↪  населения умерла.\n";
  }
}
```

Python

```
def plague():
  global year, population
  if year > 1 and random.randint(0, 99) <
  ↪  15:
    population -= population // 2
    print('\nЭпидемия чумы! Половина
    ↪  населения умерла.')
```

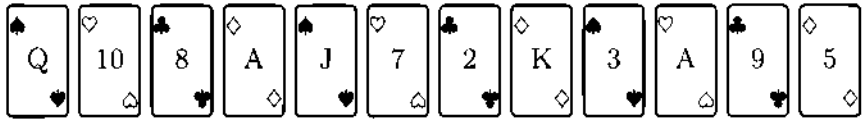
Как видно, она может произойти в любой год (кроме начала игры) с вероятностью 15%: случайное число от 0 до 99 должно быть меньше 15. Если такое событие происходит, население уменьшается вдвое. Но на показатель «умершие от голода» это не влияет. А количество земли на душу населения увеличивается. И так как этот показатель проверяется в конце игры, то в каком-то циничном смысле чума даже выгодна.

### 4.1.9. Задания для самостоятельной работы

**1.** Попробуйте набирать текст этой игры по частям. Сначала поиграйте без вредных событий вроде чумы или нашествия крыс. Удастся ли вам достичь выдающегося результата?

**2.** Придумайте какое-нибудь новое случайное событие. Для разнообразия можно сделать его позитивным.

## 4.2. Игра «Blackjack»




Игра «Blackjack» — это классическая карточная игра, в которой игрок соревнуется с раздающим (дилером, крупье). Игрок должен взять две или больше карт, чтобы набрать не больше 21 очка. Значения очков для карт от двойки до десятки — от 2 до 10 соответственно, у туза — 1 или 11 (в зависимости от того, какая сумма нужна), а у карт с картинками — 10.

Но игрок должен не только не превысить 21. Ещё нужно не проиграть дилеру, который тоже получает карты. Если дилеру удастся набрать больше очков, чем игроку (также не превысив 21), то он побеждает.

Карты на руке в этой игре всегда известны, а разнообразие ходов невелико. Поэтому сложного искусственного интеллекта здесь не предвидится. Отличная игра, для того чтобы потренироваться в разработке несложных программ.

**Экран 1. Игра «Blackjack»**

```
Мои карты: JD
Очки: 10

Ваши карты: 5C 8H
Очки: 13

Будете брать ещё карту? (Y/N) y
Ваши карты: 5C 8H 7H
Очки: 20

Будете брать ещё карту? (Y/N) n
Мои карты: JD 8C
Очки: 18

Мои карты: JD 8C 9H
Очки: 27

У вас 20 очков, у меня 27 очков
У меня перебор, вы выиграли!
```

### 4.2.1. Игровая позиция

Конечно, каждая игра с компьютером содержит игровой цикл, но пока отложим его рассмотрение, а сначала придумаем несколько других частей программы.

Не менее важная часть — это то, как хранится игровая позиция. В игре используются карты, и всё зависит от того, где какая карта находится в текущий момент. Карты могут быть в руках у игроков или в колоде. Причём порядок их в колоде неизвестен игроку, но может быть вполне определённым.

Удобнее всего хранить карты в массиве, который может расширяться или уменьшаться. Тогда можно эффективно и забирать их оттуда, и добавлять новые.

Как закодировать информацию о карте? У каждой из карт есть достоинство и масть. Мастей всего четыре, а достоинств тринадцать. Можно было бы закодировать всё это одним числом (например, от 0 до 51), но немного усложнился бы код вывода и подсчёта очков. А так как память нам экономить не нужно (карт же всего 52), будем хранить масть отдельно от достоинства.

Опишем эти структуры данных. Тип для хранения атрибутов одной карты будет называться `Card`, а тип массива для набора карт — `Deck`.

**C++**

```
#include <vector>

struct Card
{
  int rank;
  char suit;
};

typedef std::vector<Card> Deck;
```

Тип `Deck` есть только здесь. Эта дополнительная работа (по сравнению с программой на Python) помогает избегать ошибок в программе. Компилятор сразу предупредит, если с переменными этого типа будут совершаться неправильные операции.

**Python**

```
from dataclasses import dataclass

@dataclass
class Card:
  rank: int
  suit: str
```

Здесь для полей структуры объявлены типы, но интерпретатор Python их не использует. Полезны они тому, кто с программой будет разбираться, а также статическим анализаторам. Анализаторы кода часто используются в больших проектах, чтобы находить пропущенные ошибки. В частности, это могут быть ошибки несоответствия типов.

Масть могла бы быть просто числом от 0 до 3, но мы сразу будем хранить символ, который будет выводиться на экран во время игры. Этот символ совпадает с первой буквой английского названия масти (Hearts, Clubs, Diamonds, Spades).

Обозначение же достоинства карты в программе удобнее начинать с единицы (это будет туз). Дальше идут карты от двойки до десятки, а потом валет, дама и

король. В этом случае значение достоинства для карт от 2 до 10 будет совпадать с очками, которые мы за них получаем. А это упростит подсчёты.

Перед началом игры массив-колоду нужно инициализировать, записав туда весь набор карт:

C++

```
int main()
{
  Deck deck;
  for (int r = 1 ; r <= 13 ; ++r)
  {
    for (char s : {'H', 'D', 'C', 'S'})
    {
      deck.push_back({r, s});
    }
  }
}
```

Python

```
deck = []
for r in range(1, 14):
  for s in 'HDCS':
    deck.append(Card(r, s))
```

### 4.2.2. Вывод карт на руке

Тип данных «колода» (то есть массив карт) пригодится не только для хранения оставшихся карт, но и для тех, которые игроки уже получили на руки.

А раз так, можно разработать функцию для вывода списка карт на экран. Оба игрока будут получать карты, и игрок-человек захочет их увидеть, чтобы принимать решения.

Для того чтобы определить победителя, не нужно знать масти карт. И даже не все номиналы важны. Но это всё нужно и для «реалистичности», и для того чтобы следить, какие карты уже выбыли.

Номиналы цифровых карт совпадают с полем `rank` в описании карты, поэтому числа 2–10 можно сразу выводить на экран. 1 — это туз (Ace), а 11–13 — валет, дама и король (Jack, Queen, King). Для них мы будем выводить первую букву названия.

Для полной колоды функция должна генерировать такой вывод:

```
AH AD AC AS 2H 2D 2C 2S 3H 3D 3C 3S 4H 4D 4C 4S 5H 5D 5C 5S 6H 6D 6C
↪  6S 7H 7D 7C 7S 8H 8D 8C 8S 9H 9D 9C 9S 10H 10D 10C 10S JH JD JC
↪  JS QH QD QC QS KH KD KC KS
```

C++

```
#include <iostream>

const char pictures[] = "JQK";

void printDeck(const Deck &hand)
{
  for (auto &card : hand)
  {
    if (card.rank == 1)
    {
      std::cout << 'A';
    }
    else if (card.rank <= 10)
    {
      std::cout << card.rank;
    }
    else
    {
      std::cout << pictures[card.rank -
      ↪ 11];
    }
    std::cout << card.suit << " ";
  }
  std::cout << "\n";
}
```

Python

```
pictures = 'JQK'

def printDeck(is_dealer, hand):
  for card in hand:
    if card.rank == 1:
      print('A', end='')
    elif card.rank <= 10:
      print(card.rank, end='')
    else:
      print(pictures[card.rank - 11],
      ↪ end='')
    print(f'{card.suit} ', end='')
```

### 4.2.3. Раздача случайных карт

Случайный порядок закрытых карт (тех, что в колоде) необязательно хранить в каких-то переменных или массивах. Ведь можно выбирать карту из произвольного места массива с основной колодой, а затем удалять её оттуда.

Удалённая карта затем должна перейти кому-то из игроков, в их колоду-массив карт. У функции `deal` два аргумента: колода, куда нужно переместить карту (`hand`), и исходная («перемешанная») колода (`deck`).

C++

```
#include <stdlib.h>

void deal(Deck &hand, Deck &deck)
{
  std::swap(deck.back(), deck[rand() %
  ↪ deck.size()]);
  hand.push_back(deck.back());
  deck.pop_back();
}
```

Python

```
import random

def deal(hand, deck):
  card = random.randint(0, len(deck) - 1)
  deck[-1], deck[card] = deck[card],
  ↪ deck[-1]
  hand.append(deck[-1])
  deck.pop()
```

Хитрость этой процедуры в том, что тут используется эффективное удаление элемента массива. Так как порядок карт в массиве, откуда мы карту забираем, в итоге не важен, то функция меняет местами последний элемент со случайным. Теперь игроку достанется последняя карта из массива (как раз случайная). И по-

сле этого последний элемент массива можно просто отбросить. Так при удалении не придётся сдвигать все элементы, которые идут после удаляемого.

Теперь, когда функция выдачи карты готова, можно наконец раздать карты игрокам.

C++

```
#include <time.h>

int main()
{
  ...
  srand((int)time(NULL));
  Deck dealer, player;
  deal(dealer, deck);
  deal(player, deck);
  deal(player, deck);
  printDeck(dealer);
  printDeck(player);
  ...
}
```

Python

```
...
dealer = []
player = []
deal(dealer, deck)
deal(player, deck)
deal(player, deck)
printDeck(dealer)
printDeck(player)
...
```

### 4.2.4. Определение победителя

Раз у игроков появились карты, то можно и сравнить их достоинства. Как посчитать очки, чтобы узнать, кто победил?

Просто так номиналы карт просуммировать нельзя. Туз может стоить 1 или 11 очков (по выбору игрока), карты с картинками стоят по 10. Опять придётся писать ветвление.

Но ровно такое же ветвление мы делали при выводе колоды на экран. Почему бы не использовать функцию `printDeck` и для подсчёта очков? Конечно, в настоящей большой программе не стоит совмещать две столь разные функции в одной процедуре. Но для нашей маленькой игровой программы это вполне приемлемо.

Итак, если попалась карта с картинкой, нужно добавить к сумме 10 очков. Если это карта с числом, то стоимость карты равна этому числу. А что делать с тузом?

Стоимость туза может меняться по выбору игрока. Поэтому по умолчанию за тузов будем начислять по одному очку. Если же в конце подсчёта можно добавить ещё десять (не превысив 21), то это можно сделать. Так игрок сможет увидеть максимальную сумму для его набора карт. А два туза сразу не могут приносить по 11 очков, потому что это сразу будет перебором.

C++

```
const int MAX = 21;

int printDeck(bool is_dealer, const Deck
↪  &hand)
{
  if (is_dealer)
  {
    std::cout << "Мои карты: ";
  }
  else
  {
    std::cout << "Ваши карты: ";
  }
  int sum = 0;
  int aces = 0;
  for (auto &card : hand)
  {
    if (card.rank == 1)
    {
      ++aces;
      ++sum;
      std::cout << 'A';
    }
    else if (card.rank <= 10)
    {
      sum += card.rank;
      std::cout << card.rank;
    }
    else
    {
      sum += 10;
      std::cout << pictures[card.rank -
      ↪  11];
    }
    std::cout << card.suit << " ";
  }
  if (aces > 0 && sum <= MAX - 10)
  {
    sum += 10;
  }
  std::cout << "\nОчки: " << sum <<
  ↪  "\n\n";
  return sum;
}
```

Python

```
MAX = 21

def printDeck(is_dealer, hand):
  if is_dealer:
    print('Мои карты: ', end='')
  else:
    print('Ваши карты: ', end='')
  sum = 0
  aces = 0
  for card in hand:
    if card.rank == 1:
      aces += 1
      sum += 1
      print('A', end='')
    elif card.rank <= 10:
      sum += card.rank
      print(card.rank, end='')
    else:
      sum += 10
      print(pictures[card.rank - 11],
      ↪  end='')
    print(f'{card.suit} ', end='')
  if aces > 0 and sum <= MAX - 10:
    sum += 10
  print(f'\nОчки: {sum}\n')
  return sum
```

Ну и чтобы функция `printDeck` выводила подсказку о том, чьи это карты, передаём ей первый параметр — булеву переменную `is_dealer`.

Теперь функция не только печатает набор карт, но и возвращает сумму очков, поэтому можно легко определить, кто победил с начальным набором карт.

C++

```
int main()
{
  ...
  int dealer_sum = printDeck(true,
  ↪ dealer);
  int player_sum = printDeck(false,
  ↪ player);

  std::cout << "У вас " << player_sum <<
  ↪ " очков, у меня " << dealer_sum <<
  ↪ " очков\n";
  if (player_sum > MAX)
  {
    std::cout << "У вас перебор, вы
    ↪ проиграли!\n";
  }
  else if (dealer_sum > MAX)
  {
    std::cout << "У меня перебор, вы
    ↪ выиграли!\n";
  }
  else if (dealer_sum > player_sum)
  {
    std::cout << "Я выиграл!\n";
  }
  else
  {
    std::cout << "Вы выиграли!\n";
  }
}
```

Python

```
...
dealer_sum = printDeck(True, dealer)
player_sum = printDeck(False, player)

print(f'У вас {player_sum} очков, у меня
↪ {dealer_sum} очков\n')
if player_sum > MAX:
  print('У вас перебор, вы проиграли!')
elif dealer_sum > MAX:
  print('У меня перебор, вы выиграли!')
elif dealer_sum > player_sum:
  print('Я выиграл!')
else:
  print('Вы выиграли!')
```



«Поиграйте» в то, что получилось. Проверьте, что карты отображаются корректно и очки для них считаются правильно.

### 4.2.5. Главный игровой цикл

Самое интересное в программе — это игровой цикл. С его помощью игрок делает последовательные ходы. То есть оценивает ситуацию, принимает решение, смотрит на результат. Короче говоря, получает удовольствие от процесса.

Особенность игры «Blackjack» в том, что игроки добирают карты каждый по отдельности, а не чередуя ходы. Поэтому сначала сконструируем цикл для игрока-человека.

По правилам он должен брать очередную карту, а затем оценивать, стоит ли брать следующую. И, конечно, мы не будем заставлять игрока вычислять стоимость его карт, а будем после очередной карты выводить его счёт.

Получает карту игрок с помощью уже известной нам функции `deal`. А дополнительное условие остановки набора карт, если сумма очков превышает 21. Это значит, партия уже проиграна, перебор.

C++

```
while (player_sum < MAX)
{
  char c;
  std::cout << "Будете брать ещё карту?
  ↪  (Y/N) ";
  std::cin >> c;
  if (c != 'y' && c != 'Y')
  {
    break;
  }
  deal(player, deck);
  player_sum = printDeck(false, player);
}
```

Python

```
while player_sum < MAX:
  c = input('Будете брать ещё карту?
  ↪  (Y/N) ')
  if c != 'y' and c != 'Y':
    break
  deal(player, deck)
  player_sum = printDeck(False, player)
```

Ну теперь уже можно и поиграть почти по-настоящему. Главное, не набрать слишком много карт. Пока у компьютера на руке только одна карта, выиграть будет очень легко.

### 4.2.6. Ещё один игровой цикл

Осталось придумать ещё один игровой цикл, в котором уже компьютер набирает себе карты. В этой игре не потребуется реализовывать сложный искусственный интеллект. Ведь когда компьютер должен остановиться? Только когда выиграет, потому что в противном случае он точно проиграет, так как счёт игрока уже известен.

Поэтому в новом цикле дилер берёт одну карту за другой, пока не наберёт очков больше, чем у игрока. Если при этом получится больше двадцати одного, то он проиграл. В противном случае он победил.

И, конечно же, брать новые карты нет никакого смысла, если перебор уже и так возник у игрока-человека. Ведь он всё равно проиграл.

Цикл получения карт должен всегда делать хотя бы одну итерацию, потому что по правилам игроки должны иметь хотя бы по две карты. А в программе изначально у дилера была только одна.

C++

```
if (player_sum < MAX)
{
  do
  {
    deal(dealer, deck);
    dealer_sum = printDeck(true, dealer);
  }
  while (dealer_sum <= player_sum)
}
```

Python

```
if player_sum < MAX:
  deal(dealer, deck)
  dealer_sum = printDeck(True, dealer)
  while dealer_sum <= player_sum:
    deal(dealer, deck)
    dealer_sum = printDeck(True, dealer)
```



Программа готова! Теперь можно играть и выигрывать. Или проигрывать, если вы слишком азартны.

### 4.2.7. Задания для самостоятельной работы

**1.** Придумайте, как рисовать карты покрупнее, чтобы игроку было удобнее их обозревать.

**2.** Переделайте подсчёт очков, чтобы игрок видел оба варианта счёта при наличии у него в руках туза. Если, конечно же, оба эти варианта не превышают 21 очка.

**3.** Игра станет интереснее, если колода будет разыгрываться до конца. Доработайте программу, чтобы проходило несколько раундов, и считайте число побед в них. Не создавайте в каждом раунде новую колоду, а «дотрачивайте» старую.

**Листинг 4.3. blackjack.cpp**

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

const char pictures[] = "JQK";
const int MAX = 21;

struct Card
{
    int rank;
    char suit;
};

typedef std::vector<Card> Deck;

void deal(Deck &hand, Deck &deck)
{
    std::swap(deck.back(), deck[rand() % deck.size()]);
    hand.push_back(deck.back());
    deck.pop_back();
}

int printDeck(bool is_dealer, const Deck &hand)
{
    if (is_dealer)
    {
```

```
        std::cout << "Мои карты: ";
    }
    else
    {
        std::cout << "Ваши карты: ";
    }
    int sum = 0;
    int aces = 0;
    for (auto &card : hand)
    {
        if (card.rank == 1)
        {
            ++aces;
            ++sum;
            std::cout << 'A';
        }
        else if (card.rank <= 10)
        {
            sum += card.rank;
            std::cout << card.rank;
        }
        else
        {
            sum += 10;
            std::cout << pictures[card.rank - 11];
        }
        std::cout << card.suit << " ";
    }
    if (aces > 0 && sum <= MAX - 10)
    {
        sum += 10;
    }
    std::cout << "\nОчки: " << sum << "\n\n";
    return sum;
}

int main()
{
    Deck deck;
    for (int r = 1 ; r <= 13 ; ++r)
    {
```

```
        for (char s : {'H', 'D', 'C', 'S'})
        {
            deck.push_back({r, s});
        }
    }

    srand((int)time(NULL));
    Deck dealer, player;
    deal(dealer, deck);
    deal(player, deck);
    deal(player, deck);
    int dealer_sum = printDeck(true, dealer);
    int player_sum = printDeck(false, player);

    while (player_sum < MAX)
    {
        char c;
        std::cout << "Будете брать ещё карту? (Y/N) ";
        std::cin >> c;
        if (c != 'y' && c != 'Y')
        {
            break;
        }
        deal(player, deck);
        player_sum = printDeck(false, player);
    }

    if (player_sum < MAX)
    {
        do
        {
            deal(dealer, deck);
            dealer_sum = printDeck(true, dealer);
        }
        while (dealer_sum <= player_sum);
    }

    std::cout << "У вас " << player_sum << " очков, у меня " <<
    ↪  dealer_sum << " очков\n";
    if (player_sum > MAX)
    {
```

```
        std::cout << "У вас перебор, вы проиграли! \n";
    }
    else if (dealer_sum > MAX)
    {
        std::cout << "У меня перебор, вы выиграли! \n";
    }
    else if (dealer_sum > player_sum)
    {
        std::cout << "Я выиграл! \n";
    }
    else
    {
        std::cout << "Вы выиграли! \n";
    }
}
```

**Листинг 4.4. blackjack.py**

```
#!/usr/bin/python3
import random
from dataclasses import dataclass

@dataclass
class Card:
    rank: int
    suit: str

pictures = 'JQK'
MAX = 21

def deal(hand, deck):
    card = random.randint(0, len(deck) - 1)
    deck[-1], deck[card] = deck[card], deck[-1]
    hand.append(deck[-1])
    deck.pop()

def printDeck(is_dealer, hand):
    if is_dealer:
        print('Мои карты: ', end='')
    else:
        print('Ваши карты: ', end='')
```

```
    sum = 0
    aces = 0
    for card in hand:
        if card.rank == 1:
            aces += 1
            sum += 1
            print('A', end='')
        elif card.rank <= 10:
            sum += card.rank
            print(card.rank, end='')
        else:
            sum += 10
            print(pictures[card.rank - 11], end='')
        print(f'{card.suit} ', end='')
    if aces > 0 and sum <= MAX - 10:
        sum += 10
    print(f'\nОчки: {sum}\n')
    return sum

deck = []
for r in range(1, 14):
    for s in 'HDCS':
        deck.append(Card(r, s))

dealer = []
player = []
deal(dealer, deck)
deal(player, deck)
deal(player, deck)
dealer_sum = printDeck(True, dealer)
player_sum = printDeck(False, player)

while player_sum < MAX:
    c = input('Будете брать ещё карту? (Y/N) ')
    if c != 'y' and c != 'Y':
        break
    deal(player, deck)
    player_sum = printDeck(False, player)

if player_sum < MAX:
    deal(dealer, deck)
```

```
    dealer_sum = printDeck(True, dealer)
    while dealer_sum <= player_sum:
        deal(dealer, deck)
        dealer_sum = printDeck(True, dealer)

print(f'У вас {player_sum} очков, у меня {dealer_sum} очков\n')
if player_sum > MAX:
    print('У вас перебор, вы проиграли!')
elif dealer_sum > MAX:
    print('У меня перебор, вы выиграли!')
elif dealer_sum > player_sum:
    print('Я выиграл!')
else:
    print('Вы выиграли!')
```

Глава 5

# Программирование прямоугольных полей

## 5.1. Игра «Ферзя в угол»

Написание игры «Ферзя в угол» — это первый шаг к программированию шахмат. В игре соперники поочерёдно двигают одну единственную фигуру ферзя. Но в отличие от шахмат, тут ферзь может ходить только влево, вниз или по диагонали влево-вниз. Цель игры — своим ходом привести ферзя в левый нижний угол.

В нашей программе первый ход будет делать игрок, причём ход будет особенным — это будет выбор клетки для начального положения ферзя. Дальше компьютер и человек будут перемещать ферзя поочерёдно, пока кто-нибудь не победит.

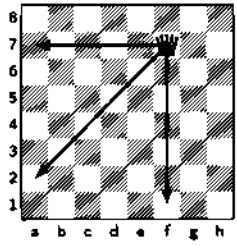
Из предыдущей главы мы знаем, что сначала нужно создать главный игровой цикл, определить, что будет игровой позицией, а также написать код, который инициализирует эту позицию.

**Экран 1. Игра в «Ферзя в угол»**

```
Игра "Ферзя в угол"
8 ........
7 ........
6 ........
5 ........
4 ........
3 ........
2 ........
1 ........
  abcdefgh
Выберите начальное положение
↪  (a8-g8, h1-h8): b8
8 .Q......
7 ........
6 ........
5 ........
4 ........
3 ........
2 ........
1 ........
  abcdefgh
```

```
Мой ход a8
8 Q.......
7 ........
6 ........
5 ........
4 ........
3 ........
2 ........
1 ........
  abcdefgh
Куда вы будете ходить? a1
8 ........
7 ........
6 ........
5 ........
4 ........
3 ........
2 ........
1 Q.......
  abcdefgh
Вы выиграли, поздравляю!
```

### 5.1.1. Игровая позиция




В игре есть поле 8 × 8 и одна фигура. Игровая позиция — это однозначное описание момента игры. Игровое поле здесь никогда не меняется, число клеток постоянно, диапазон значений координат ферзя тоже.

Поэтому у нас есть два варианта описания позиции:

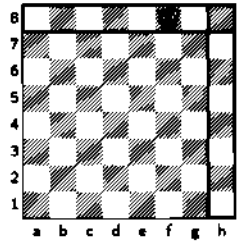
- Состояние каждой клетки — какая там стоит фигура. В нашем случае будет или ничего, или ферзь.
- Координаты ферзя на поле.

Конечно, хранить координаты ферзя намного удобнее, чем состояние всех клеток поля. Для этого понадобится две переменные — $x$ и $y$ вместо массива из 64 значений.

В этих переменных будем хранить значения от 1 до 8. И хотя на рисунке номер строки обозначается буквой, в Python использовать для этого буквы неудобно. В C++ координата $x$ могла бы принимать значения от `'a'` до `'h'`, и это не ухудшило бы код программы (вы убедитесь в этом позднее). Но для единообразия мы будем также использовать числа от 1 до 8.

### 5.1.2. Начало игры

Начинается игра с правой вертикали или верхней горизонтали поля.




Чтобы игрок не путал вертикаль и горизонталь, пусть вводит их в шахматной нотации — букву и цифру, соответствующие клетке. Внутри программы мы букву тоже преобразуем в число, и получится две координаты от 1 до 8.

Ввод координат будет использоваться каждый ход. Только в начале игры клетка должна быть справа или вверху поля, а потом каждый следующий ход должен удовлетворять правилам перемещения ферзя.

То есть ввод у нас будет одинаковый (координаты клетки), а проверки будут разные. Поэтому непосредственно ввод координат и проверку их попадания в границы поля вынесем в отдельную функцию `inputPos`.

**C++**

```
void inputPos(int &x, int &y)
{
    ...
}

int main()
{
    int x, y;
    do
    {
        inputPos(x, y);
    }
    while (x != 8 && y != 8);
    ...
}
```

**Python**

```
def inputPos():
    ...
    return x, y

x = 0
y = 0
while x != 8 and y != 8:
    x, y = inputPos()
```

Функция `inputPos` принимает два параметра по ссылке. Но она не будет использовать их значения, а будет в эти переменные записывать результат. Всё из-за того, что нам нужно сразу два числа, а возвращаемое значение у функции только одно.

В Python есть удобная возможность возвращать из функции сразу несколько значений, перечисляя их через запятую. При этом на самом деле возвращается одно значение — кортеж, который мы снова распаковываем в две переменные в точке вызова функции.

Тут нужен цикл, потому что игрок должен выбирать стартовые координаты только из правой или верхней частей поля. Если это не так, ввод придётся повторить.

### 5.1.3. Ввод координат

Чтобы получить координаты на поле, нужно считать с клавиатуры два символа — столбец и строку, а затем преобразовать их в числа от 1 до 8. А если строка или столбец неправильные, игрок должен повторить ввод, пока получится корректная клетка на поле.

В кодировке ASCII (которая скорее всего и используется на вашем компьютере) всем буквам соответствуют числовые коды. Для цифр и латинского алфавита эти коды идут подряд по возрастанию. Например, символ `'0'` имеет код 48, `'1'` — код 49, `'2'` — код 50. Поэтому, чтобы получить число от 1 до 8 из символов от `'1'` до `'8'`, нужно из кодов этих символов вычесть код символа `'0'`.

Аналогично можно сделать и с буквами. Только для алфавита нет аналога нуля, поэтому мы вычитаем из введённой переменной код символа `'a'` и прибавляем к результату единицу. После этого для `'a'` получится 1, для `'b'` — 2 и так далее.

**C++**

```
void inputPos(int &x, int &y)
{
    char a, b;
    do
    {
        std::cin >> a >> b;
    }
    while (a < 'a' || a > 'h'
        || b < '1' || b > '8');
    x = a - 'a' + 1;
    y = b - '0';
}
```

В C++ символьный тип `char` — это всего лишь маленький целочисленный тип. Поэтому такие переменные можно использовать в обычных арифметических выражениях. В выражении эта переменная превратится в числовой код соответствующего символа.

**Python**

```
def inputPos():
    s = ''
    while len(s) != 2
        or s[0] < 'a' or s[0] > 'h'
        or s[1] < '1' or s[1] > '8':
        s = input()
    return ord(s[0]) - ord('a') + 1,
    ↪  ord(s[1]) - ord('0')
```

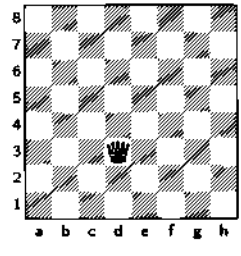
Прочитать ровно два отдельных символа с клавиатуры не так-то просто, поэтому мы считываем строку, а затем проверяем, что введены ровно два символа. Для вычисления номеров строки и столбца используется функция получения кода символа — `ord`.

В Python вместо арифметических операций с кодами символов можно было бы использовать функцию `int`. Сможете это сделать как для строки, так и для столбца? Подсказка: используйте основание системы счисления, не равное 10.

### 5.1.4. Вывод поля

Чтобы играть в эту игру, достаточно знать координаты ферзя на поле. Других фигур нет, поэтому легко представить, что происходит. Но будет намного удобнее, если программа «нарисует» игровое поле на экране.

Например, для позиции $(4, 3)$ получается такая доска:




Её можно представить на экране, используя буквы и символы:

```
8 ........
7 ........
6 ........
5 ........
4 ........
3 ...Q....
2 ........
1 ........
  abcdefgh
```

Пока проигнорируем цифры и выведем точки с ферзём с помощью двух вложенных циклов:

C++

```
for (int i = 8 ; i > 0 ; --i)
{
    for (int j = 1 ; j < 9 ; ++j)
    {
        std::cout <<
            j == x && i == y
            ? 'Q' : '.';
    }
    std::cout << '\n';
}
```

Python

```
for i in range(8, 0, -1):
    for j in range(1, 9):
        if j == x and i == y:
            print('Q', end='')
        else:
            print('.', end='')
    print('')
```

### 5.1.5. Дорабатываем вывод поля

Добавим теперь вывод координатной сетки. Ведь без неё сложно будет правильно ввести свой ход.

C++

```
for (int i = 8 ; i > 0 ; --i)
{
    std::cout << i << ' ';
    for (int j = 1 ; j < 9 ; ++j)
    {
        std::cout <<
            j == x && i == y
            ? 'Q' : '.';
    }
    std::cout << '\n';
}
std::cout << "  abcdefgh\n";
```

Python

```
for i in range(8, 0, -1):
    print(f'{i} ', end='')
    for j in range(1, 9):
        if j == x and i == y:
            print('Q', end='')
        else:
            print('.', end='')
    print('')
print('  abcdefgh')
```

Можно не выводить по одной точке, а сгенерировать сразу строку из точек, которую и выводить. Только условие всё равно останется, потому что иногда в этой строке из точек появляется ферзь.

C++

```
for (int i = 8 ; i > 0 ; --i)
{
    std::string s(8, '.');
    if (i == y)
    {
        s[x - 1] = 'Q';
    }
    std::cout << i << ' ' << s << '\n';
}
std::cout << "  abcdefgh\n";
```

Python

```
for i in range(8, 0, -1):
    s = '.' * 8
    if i == y:
        s = s[:x - 1] + 'Q' + s[x:]
    print(f'{i} {s}')
print('  abcdefgh')
```

Наберите этот код и проверьте, как он работает для разных значений $x$ и $y$.

### 5.1.6. Главный игровой цикл

В главном цикле уже знакомый порядок действий:

- Ход компьютера.
- Вывод поля.
- Проверка условия поражения.
- Ход игрока.
- Вывод поля.
- Проверка условия победы.

Для организации последовательности ходов можно использовать бесконечный цикл. Точнее, он был бы бесконечным, если бы не использование оператора `break`.

Так как здесь нужны две точки выхода из цикла (победа игрока и победа компьютера), то хотя бы без одного `break` не обойтись.

Поэтому нет смысла писать `while (x != 1 || y != 1)` ещё и в заголовке цикла. Выражение `true` в качестве условия продолжения цикла подчёркивает, что стоит присмотреться к телу цикла повнимательнее, чтобы понять его работу.

C++

```
...
while (true)
{
    ...
    break;
    ...
}
```

Python

```
...
while True:
    ...
    break
    ...
```

Вывод поля мы написали раньше. Оказалось, его нужно использовать дважды, поэтому вынесем код вывода в функцию `printField`.

C++
```
void printField(int x, int y)
{
    ...
}
```

Python
```
def printField(x, y):
    ...
```

«Условие победы» означает, что выигрывает человек. «Условие поражения» — проигрыш человека. В этой игре победы можно достичь только своим ходом, противник не может «упасть с доски» и проиграть, поэтому каждое из этих условий проверяется только после хода соответствующего игрока, человека или программы.

C++
```
...
while (true)
{
    ...
    printField(x, y);
    if (x == 1 && y == 1)
    {
        std::cout << "Вы проиграли!\n";
        break;
    }
    ...
    printField(x, y);
    if (x == 1 && y == 1)
    {
        std::cout << "Вы выиграли,
        ↪ поздравляю!\n";
        break;
    }
}
```

Python
```
...
while True:
    ...
    printField(x, y)
    if x == 1 and y == 1:
        print('Вы проиграли!')
        break
    ...
    printField(x, y)
    if x == 1 and y == 1:
        print('Вы выиграли, поздравляю!')
        break
```

Осталось заменить многоточия в получившемся цикле строками для совершения ходов человеком и программой.

### 5.1.7. Делаем ходы

Ход игрока — это координаты клетки, куда должен переместиться ферзь. Для считывания координат с клавиатуры мы уже раньше придумали функцию `input`.

Но после её вызова надо проверить, может ли ферзь так переместиться из текущей позиции. Всего есть три варианта:

- Ход влево. Уменьшается координата $x$, а $y$ не меняется.
- Ход вниз. Уменьшается координата $y$, а $x$ не меняется.
- Ход по диагонали. $x$ и $y$ уменьшаются на одинаковую величину.

Напишем эти условия по отдельности:

C++

```
int x1, y1;
while (true)
{
    input(x1, y1);
    int dx = x1 - x;
    int dy = y1 - y;
    if ((dx < 0 && dy == 0)
        || (dy < 0 && dx == 0)
        || (dx < 0 && dx == dy))
    {
        break;
    }
}
x = x1;
y = y1;
```

Python

```
while True:
    x1, y1 = inputPos()
    dx = x1 - x
    dy = y1 - y
    if (dx < 0 and dy == 0)
        or (dy < 0 and dx == 0)
        or (dx < 0 and dx == dy):
        break
x = x1
y = y1
```

Уже почти можно играть. Сделаем простенький выбор хода компьютером: двигаться влево, а если это невозможно, то вниз.

C++

```
if (x > 1)
{
    --x;
}
else
{
    --y;
}
```

Python

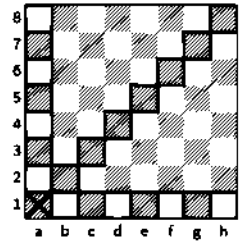
```
if x > 1:
    x -= 1
else:
    y -= 1
```

Соедините все эти части и поиграйте. Но не просто так, а протестируйте, как программа реагирует на разные варианты хода, корректные и некорректные. Попытайтесь выиграть и проиграть, подходя к последней клетке с разных сторон.

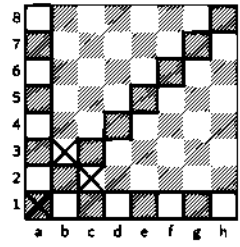
### 5.1.8. Искусственный интеллект: анализ ходов

Теперь попробуем сделать программу непобедимой. Ну насколько это будет возможно. Игра «Ферзя в угол» относится к играм с открытой (или полной) информацией. Это значит, что вся игровая позиция всегда доступна любому игроку. Ничего не скрыто, как содержимое колоды в карточных играх. Кроме того, в этой игре нет никаких случайностей, вроде бросков кубика. Благодаря таким свойствам мы можем найти стратегию, при которой будем всегда выигрывать. За исключением тех случаев, когда соперник сам знает такую стратегию и ходит первым.

Начнём анализ игры с конца. Из каких клеток можно сделать выигрышный ход? Очевидно, это $a1 - a8$, $a1 - h1$ и диагональ $a1 - h8$:



Следовательно, если из какой-то клетки можно попасть только в одну из перечисленных, эта клетка будет «проигрышной». Такими полями будут $b3$ и $c2$:

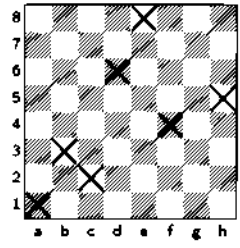


Итак, если мы своим ходом можем попасть в $a1$, $b3$ или $c2$, то победа у нас в руках. Какие из оставшихся клеток подходят для этого?

⌨ | Закончите рассуждения и найдите все «проигрышные клетки».

### 5.1.9. Искусственный интеллект: проверка позиции

Вот такой набор проигрышных полей должен был у вас получиться:



Значит, если мы сделаем ход в одну из этих позиций, то что бы ни делал противник, его можно будет обыграть. Для этого надо всё время ставить ферзя в клетки, помеченные крестиком. Попробуем научить этому компьютер.

Выпишем координаты проигрышных позиций. Это будут поля $a1$, $b3$, $c2$, $d6$, $f4$ $e8$, $h5$. Или, если перевести это в числа, $(1,1)$, $(2,3)$, $(3,2)$, $(4,6)$, $(6,4)$, $(5,8)$, $(8,5)$. Причём клетку $(5,8)$ (и аналогичную ей $(8,5)$) можно сразу исключить, потому что в неё можно попасть только из $(6,8)$, $(7,8)$, $(8,8)$. Но из этих полей мы и так можем перейти в проигрышные для противника $(4,6)$, $(2,3)$, $(1,1)$ соответственно.

Так что у нас остаётся всего пять полей для проверки. Можно на каждом ходу компьютера проверять, достижима ли одна из этих клеток. И если это так, то ставить ферзя туда. Так как эти проверки придётся делать несколько раз, то можно вынести их в функцию:

C++

```
bool checkMove(int x, int y,
    int &qx, int &qy)
{
    if (x > qx || y > qy
        || (x == qx && y == qy))
    {
        return false;
    }
    if (x == qx || y == qy
        || y - qy == x - qx)
    {
        qx = x;
        qy = y;
        return true;
    }
    return false;
}
```

Функция возвращает истину, если ход возможен. Так мы сможем проверить, стоит ли пробовать следующие клетки.

Python

```
def checkMove(x, y, qx, qy):
    if x > qx or y > qy
        or (x == qx and y == qy):
        return None
    if x == qx or y == qy
        or y - qy == x - qx:
        return x, y
    return None
```

Функция возвращает или кортеж `(x, y)`, если ход возможен, или `None`, если нет.

### 5.1.10. Искусственный интеллект: пишем код

C++

```
if (!checkMove(1, 1, x, y)
    && !checkMove(2, 3, x, y)
    && !checkMove(3, 2, x, y)
    && !checkMove(6, 4, x, y)
    && !checkMove(4, 6, x, y))
{
    // делаем произвольный ход
    ...
}
```

Python

```
t = checkMove(1, 1, x, y)
    or checkMove(2, 3, x, y)
    or checkMove(3, 2, x, y)
    or checkMove(6, 4, x, y)
    or checkMove(4, 6, x, y)
if t is not None:
    x, y = t
else:
    # делаем произвольный ход
    ...
```

Здесь мы пытаемся походить в одну из выгодных нам клеток. Если это не получается сделать (функция всегда завершается неудачно), значит, игрок перехитрил программу, поставив ферзя в начальное положение, из которого нельзя выиграть (то есть клетки $e8$ или $h5$. В этом случае можно делать произвольный ход, как это уже было раньше в программе (например, на клетку влево или вниз).

Все части программы готовы. Теперь можно играть против умного компьютера. Попробуйте его победить или хотя бы затянуть партию, если попали в проигрышную позицию.

### 5.1.11. Задания для самостоятельной работы

**1.** Добавьте ввод размерности поля игроком (или генерацию размеров случайным образом). Что при этом изменится в программе?

**2.** Переделайте код, который отвечает за «проигрышные» ходы, то есть делает ходы, когда программа видит, что ей не выиграть. Пусть они будут случайными на произвольное расстояние в одном из трёх допустимых направлений, а не просто перемещением на одну клетку.

**Листинг 5.1. queen.cpp**

```
#include <iostream>

void inputPos(int &x, int &y)
{
  char a, b;
  do
  {
    std::cin >> a >> b;
  }
  while (a < 'a' || a > 'h'
    || b < '1' || b > '8');
  x = a - 'a' + 1;
  y = b - '0';
}

void printField(int x, int y)
{
  for (int i = 8 ; i > 0 ; --i)
  {
    std::string s(8, '.');
    if (i == y)
    {
      s[x - 1] = 'Q';
    }
    std::cout << i << ' ' << s << '\n';
  }
```

```
  std::cout << "  abcdefgh\n";
}

bool checkMove(int x, int y, int &qx, int &qy)
{
  if (x > qx || y > qy
    || (x == qx && y == qy))
  {
    return false;
  }
  if (x == qx || y == qy
    || y - qy == x - qx)
  {
    qx = x;
    qy = y;
    return true;
  }
  return false;
}

int main()
{
  std::cout << "Игра \"Ферзя в угол\"\n";
  int x, y;
  printField(0, 0);
  do
  {
    std::cout << "Выберите начальное положение (a8-g8, h1-h8): ";
    inputPos(x, y);
  }
  while (x != 8 && y != 8);

  printField(x, y);
  while (true)
  {
    if (!checkMove(1, 1, x, y)
      && !checkMove(2, 3, x, y)
      && !checkMove(3, 2, x, y)
      && !checkMove(6, 4, x, y)
      && !checkMove(4, 6, x, y))
    {
```

```
      if (x > 1)
      {
        --x;
      }
      else
      {
        --y;
      }
    }
    std::cout << "Мой ход " << (char)('a' + x - 1) << y << "\n";
    printField(x, y);
    if (x == 1 && y == 1)
    {
      std::cout << "Вы проиграли! \n";
      break;
    }
    int x1, y1;
    while (true)
    {
      std::cout << "Куда вы будете ходить? ";
      inputPos(x1, y1);
      int dx = x1 - x;
      int dy = y1 - y;
      if ((dx < 0 && dy == 0)
        || (dy < 0 && dx == 0)
        || (dx < 0 && dx == dy))
      {
        break;
      }
    }
    x = x1;
    y = y1;
    printField(x, y);
    if (x == 1 && y == 1)
    {
      std::cout << "Вы выиграли, поздравляю! \n";
      break;
    }
  }
}
```

**Листинг 5.2. queen.py**

```
#!/usr/bin/python3
def inputPos():
    s = ''
    while len(s) != 2 or s[0] < 'a' or s[0] > 'h' or s[1] < '1' or
    ↪ s[1] > '8':
        s = input()
    return int(s[0], 18) - 9, int(s[1])

def printField(x, y):
    for i in range(8, 0, -1):
        s = '.' * 8
        if i == y:
            s = s[:x - 1] + 'Q' + s[x:]
        print(f'{i} {s}')
    print('  abcdefgh')

def checkMove(x, y, qx, qy):
    if x > qx or y > qy or (x == qx and y == qy):
        return None
    if x == qx or y == qy or y - qy == x - qx:
        return x, y
    return None

print('Игра "Ферзя в угол"')
printField(0, 0)
x = 0
y = 0
while x != 8 and y != 8:
    print('Выберите начальное положение (a8-g8, h1-h8): ', end='')
    x, y = inputPos()

while True:
    t = checkMove(1, 1, x, y) or checkMove(2, 3, x, y) \
        or checkMove(3, 2, x, y) or checkMove(6, 4, x, y) \
        or checkMove(4, 6, x, y)
    if t is not None:
        x, y = t
    else:
        if x > 1:
            x -= 1
```

```
    else:
        y -= 1
print(f'Мой ход {chr(ord("a") + x - 1)}{y}')
printField(x, y)
if x == 1 and y == 1:
    print('Вы проиграли!')
    break
while True:
    print('Куда вы будете ходить? ', end='')
    x1, y1 = inputPos()
    dx = x1 - x
    dy = y1 - y
    if (dx < 0 and dy == 0) \
        or (dy < 0 and dx == 0) \
        or (dx < 0 and dx == dy):
        break
x = x1
y = y1
printField(x, y)
if x == 1 and y == 1:
    print('Вы выиграли, поздравляю!')
    break
```

## 5.2. Игра «Чудовища»

На болоте спрятались четыре чудовища. Вам нужно их поймать. Болото представляет собой квадрат, разделённый на клетки с координатами от $0,0$ до $9,9$. У вас есть десять попыток, чтобы поймать всех чудовищ.

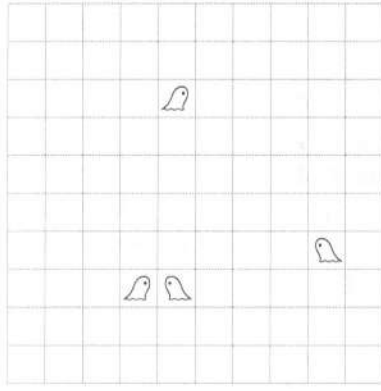
**Экран 1. Игра «Чудовища»**

```
Ход 1
Введите координаты: 4 8
Монстр 1 на расстоянии 1.0
Монстр 2 на расстоянии 5.1
Монстр 3 на расстоянии 6.4
Монстр 4 на расстоянии 5.0
Ход 2
Введите координаты: 5 8
Монстр 1 пойман!
Монстр 2 на расстоянии 5.0
Монстр 3 на расстоянии 5.66
Монстр 4 на расстоянии 5.1
Ход 3
Введите координаты: 5 3
Монстр 2 пойман!
Монстр 3 на расстоянии 4.12
Монстр 4 на расстоянии 1.0
Ход 4
Введите координаты: 5 4
Монстр 3 на расстоянии 4.0
Монстр 4 на расстоянии 1.41
Ход 5
Введите координаты: 4 3
Монстр 3 на расстоянии 5.1
Монстр 4 пойман!
Ход 6
Введите координаты: 5 8
Монстр 3 на расстоянии 5.66
Ход 7
Введите координаты: 9 4
Монстр 3 пойман!
Поздравляем, вы поймали всех монстров за 7 ходов!
```

Каждая попытка начинается с ввода координат. Если по этим координатам есть чудовище, оно считается пойманным. Для тех же чудовищ, которые ещё не пойманы, выводится расстояние от них до выбранной клетки.

К счастью, чудовища не перемещаются. Успеете ли вы поймать их всех?

### 5.2.1. Игровая позиция



Чем игровая позиция здесь отличается от игры про ферзя? Теперь у нас не одна фигура, а целых четыре. Можно для хранения их координат на поле завести 8 переменных-координат. Конечно же, лучше для этого использовать массив, так как операции со всеми чудовищами будут одинаковые – сравнение их координат и вычисление расстояний.

**C++**

```
const int M = 4;
int x[M], y[M];
```

Массивы здесь не инициализируем, потому что потом будем туда записывать случайные координаты (монстры ведь прячутся от игрока).

**Python**

```
M = 4
x = [0] * M
y = [0] * M
```

Констант в Python нет, поэтому для хранения количества чудовищ используем просто переменную $M$.

Константа (переменная) $M$ нужна для обозначения количества чудовищ. Если мы захотим изменить программу, увеличив число чудовищ, это можно будет сделать всего лишь исправлением значения $M$.

С помощью координат монстров мы сможем проверять расстояние до них. Но как мы будем узнавать, пойман какой-либо монстр или нет? Видимо, каждому монстру нужен ещё признак «пойманности». Его можно хранить в булевой переменной. Точнее, нам понадобится массив из булевых переменных, потому что монстров несколько:

C++
```
bool caught[M];
```

Python
```
caught = [False] * M
```

### 5.2.2. Структура программы

В самом начале нужно «спрятать» монстров на поле. Затем начнётся главный игровой цикл. Программа в этой игре только проверяет ходы игрока, а не действует как его соперник. Поэтому в главном игровом цикле обрабатываются только последствия ходов человека.

Заканчивается цикл либо после 10 попыток, либо если все монстры найдены.

C++
```
#include <time.h>
#include <stdlib.h>

int main()
{
    srand((int)time(NULL));

    int move;
    placeMonsters();
    for (move = 1
        ; move <= 10 ; ++move)
    {
        printMonsters();
        inputMove();
        checkDistance();
        if (allCaught())
        {
            break;
        }
    }
    finalMessage(move);
}
```

Python
```
import random

placeMonsters()
move = 1
while move <= 10:
    printMonsters()
    inputMove()
    checkDistance()
    if allCaught():
        break
    move += 1
finalMessage(move)
```

Чтобы было проще писать программу дальше, все действия в основном алгоритме мы записали в виде вызовов функций. Осталось эти функции реализовать — и программа готова. Все функции сейчас без параметров и возвращаемых значений, но с этим мы определимся в процессе того, как будем их писать. Вот эти функции будут нужны:

- `placeMonsters` — «прячет» монстров на игровом поле.
- `printMonsters` — печатает координаты монстров. Эта функция не нужна в финальной версии программы, ведь монстры прячутся, и мы не можем знать их положение. Но будет удобно видеть все координаты, когда мы отлаживаем программу.
- `inputMove` — ход игрока, ввод координат проверяемой клетки.

- `checkDistance` — проверяет, пойман ли какой-то монстр, и выводит на экран расстояние до тех из них, которые не были пойманы.
- `allCaught` — возвращает «истину», если игроку удалось поймать всех монстров.
- `finalMessage` — вывод итогов игры: сколько монстров поймано, сколько ходов потребовалось.

Создайте набор пустых функций, используемых во фрагменте кода выше. Ну и его тоже заодно наберите. Получится «рабочая» программа, которую можно запустить, и она даже не зависнет.

### 5.2.3. Расставляем монстров

Прежде чем определять, где будут монстры, напишем функцию для отладочного вывода их координат. Без неё сложно будет проверять, правильно ли даются подсказки игроку и корректно ли размещаются монстры.

**C++**

```
void printMonsters()
{
  std::cout << "Монстры:\n";
  for (int i = 0 ; i < M ; ++i)
  {
    std::cout << i << ": ";
    if (caught[i])
    {
      std::cout << "пойман\n";
    }
    else
    {
      std::cout << x[i] << " "
        << y[i] << "\n";
    }
  }
}
```

**Python**

```
def printMonsters():
  print('Монстры:')
  for i in range(M):
    print(f'{i}: ', end='')
    if caught[i]:
      print('пойман')
    else:
      print(f'{x[i]} {y[i]}')
```

Положение монстра на поле — это пара координат. Поэтому расстановка монстров заключается в случайном выборе их координат. Случайные числа мы генерировать умеем, поэтому код расстановки монстров может выглядеть так:

**C++**

```
const int S = 10;

void placeMonsters()
{
  for (int i = 0 ; i < M ; ++i)
  {
    x[i] = rand() % S;
    y[i] = rand() % S;
  }
}
```

**Python**

```
S = 10

def placeMonsters():
  for i in range(M):
    x[i] = random.randint(0, S - 1)
    y[i] = random.randint(0, S - 1)
```

Новая константа $S$ — это размер игрового поля (болота). Мы завели её, потому что это значение используется как при генерации координат монстров, так и при проверке хода игрока.

Вставьте эти две функции в нашу программу и потестируйте. Правильно ли размещаются монстры при разных запусках программы?

### 5.2.4. Чиним расстановку монстров

Если вы тщательно тестировали функцию расстановки монстров, то могли встретиться с ситуацией вроде этой:

**Экран 2. Два монстра с одинаковыми координатами**

```
Монстры:
0: 5 1
1: 7 8
2: 7 8
3: 6 0
```

Второй и третий (то есть первый и второй, если считать от нуля) монстры заняли одну и ту же клетку! Позиции монстров должны различаться, иначе ловить их не так интересно. Каковы шансы увидеть совпадение координат? Для каждого монстра мы можем выбрать одну из $10 \times 10 = 100$ позиций. Всего монстров 4, значит, вариантов расстановки будет $100^4 = 10^8$.

Теперь рассмотрим случай, когда совпадают координаты первого и второго монстра. Тогда для первого можно будет выбрать одну из 100 позиций, для второго берём ту же, что и для первого, а для третьего и четвёртого снова будет по 100 позиций. Итого $100 \times 100 \times 100 = 10^6$ вариантов. Всего таких совпадений первый-второй, первый-третий, первый-четвёртый, второй-третий и так далее будет 6. Значит, неудачных расстановок всего $6 \times 10^6$.

Этот способ не даёт точного ответа, потому что мы дважды посчитали, к примеру, ту ситуацию, когда одновременно совпадают координаты первого со вторым и третьего с четвёртым монстров. Попробуйте уточнить нашу оценку, чтобы получился идеальный ответ.

Итак, примерно $6 \times 10^6$ расстановок из $10^8$ нас не устраивают. То есть один из $\frac{10^8}{6 \times 10^6} \approx 16$ запусков программы выдаст совпадающие положения монстров. Как это исправить?

Когда мы ищем место для очередного монстра, нужно, чтобы он не попадал в уже занятую клетку. Такую проверку можно сделать с помощью цикла, который проверяет всех уже размещённых монстров:

C++
```
for (int i = 0 ; i < M ; ++i)
{
  bool again = true;
  while (again)
  {
    x[i] = rand() % S;
    y[i] = rand() % S;
    again = false;
    for (int j = 0 ; j < i ; ++j)
    {
      if (x[i] == x[j]
        && y[i] == y[j])
        again = true;
    }
  }
}
```

Python
```
for i in range(M):
  again = True
  while again:
    x[i] = random.randint(0, S - 1)
    y[i] = random.randint(0, S - 1)
    again = False
    for j in range(i):
      if x[i] == x[j] and
        y[i] == y[j]:
        again = True
```

### 5.2.5. Ход игрока

В игре «Ферзя в угол» функция ввода координат принимала две ссылки на переменные $x$ и $y$, что позволяло записывать в них результат её работы. Попробуем для нашей новой игры придумать более элегантный способ, чтобы у функции было только одно возвращаемое значение.

Можно использовать кортежи — упорядоченные наборы из фиксированного числа значений. Но это не очень элегантный способ, ведь там не будет названий полей вроде $x$ и $y$, а будут только номера. Ещё кортежи в Python нельзя изменять, что тоже может быть не очень удобно.

Более наглядный способ — это структуры данных с именованными полями. Внутри они мало чем отличаются от кортежей, но зато имеют удобный синтаксис: поля имеют имена, а в C++ ещё и типы, поэтому перепутать индексы не получится.

Вот так можно описать структуру из двух значений $x$ и $y$:

C++
```
struct Pos { int x, y; };
```

Python
```
from dataclasses import dataclass

@dataclass
class Pos:
    x: int
    y: int
```

Тогда наша функция `inputMove`, которая пока ничего не делает, должна выглядеть примерно так:

C++
```
Pos inputMove()
{
    return Pos(1, 2);
}
```

Python
```
def inputMove():
    return Pos(1, 2)
```

1 и 2 инициализируют $x$ и $y$ соответственно. Теперь осталось сделать так, чтобы вместо этих констант возвращались значения, введённые пользователем. И не забываем проверить корректность этих значений:

C++
```
Pos inputMove()
{
  int x, y;
  do
  {
    std::cin >> x >> y;
  }
  while (x < 0 || x >= S
    || y < 0 || y >= S);
  return Pos(x, y);
}
```

Python
```
def inputMove():
  x = -1
  y = -1
  while x < 0 or x >= S or
    y < 0 or y >= S:
    x, y = [int(s) for s in
    ↪ input().split()]
  return Pos(x, y)
```

Исправьте ранее написанный код, чтобы он использовал наш новый тип `Pos` вместо отдельных координат. Например, объявление массива для хранения положений монстров теперь будет выглядеть как `Pos m[M];` на C++ и `m = [Pos(0, 0)] * M` на Python.

Наконец-то с нашей программой можно играть! Ходы вводятся совсем по-настоящему. Правда, поймать пока никого не удаётся.

### 5.2.6. Ловим чудовищ

Мы уже предусмотрели функцию `checkDistance`, которая напечатает расстояние до всех чудовищ. А заодно и пометит тех, которые в результате этого хода были пойманы. Но что мы не предусмотрели, это как именно функция получит координаты, введённые пользователем.

Есть всего два варианта: глобальные переменные и аргументы функции. Первый удобен, потому что не надо писать ничего лишнего — только объявление переменной и обращения к ней. Но в серьёзных программах предпочтительнее именно второй, так как он не засоряет глобальный код лишними объявлениями, они будут расположены локально в нужной функции. Кроме того, никто не сможет по ошибке поменять эти переменные, сразу будет понятно, где автор программы запланировал их использовать.

Значит, будем использовать параметры. Точнее, всего один параметр, ведь мы упаковали обе координаты в одну структуру. Поэтому эти координаты можно передавать как одну переменную.

**C++**

```
void checkDistance(Pos pos)
{
  for (int i = 0 ; i < M ; ++i)
  {
    if (caught[i])
    {
      continue;
    }
    std::cout << "Монстр "
      << i + 1;
    if (pos.x == m[i].x
      && pos.y == m[i].y)
    {
      caught[i] = true;
      std::cout << " поймал!\n";
    }
    else
    {
      double dist = 1;
      std::cout
        << " на расстоянии "
        << dist << "\n";
    }
  }
}

checkDistance(inputMove());
```

**Python**

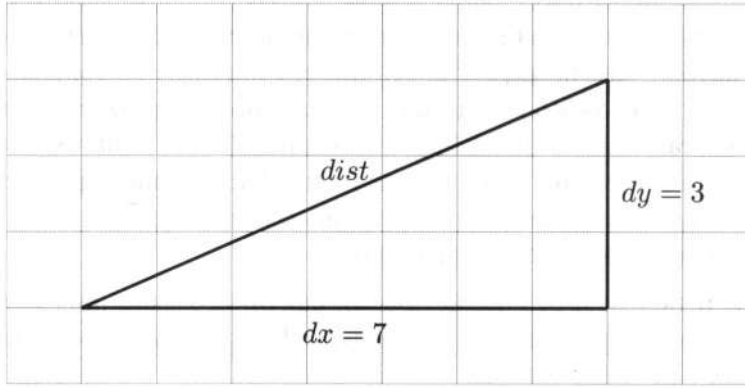
```
def checkDistance(pos):
  for i in range(M):
    if caught[i]:
      continue
    print(f'Монстр {i + 1}', end='')
    if pos.x == m[i].x and pos.y ==
    ↪ m[i].y:
      caught[i] = True;
      print(' поймал!')
    else:
      dist = 1
      print(f' на расстоянии
      ↪ {dist:.3}')

checkDistance(inputMove())
```

Обратите внимание, что мы не стали объявлять новую переменную под возвращаемое значение функции `inputMove`, а сразу передали его дальше в `checkDistanc`

Добавьте эту функцию в программу и попробуйте поиграть. Получается ли поймать хоть кого-то без правильных подсказок?

### 5.2.7. Определяем расстояние до монстров

Наша программа уже умеет проверять равенство координат. Поэтому, если быть очень настойчивым, можно взять и поймать всех чудовищ, пробуя вариант за вариантом.



Очевидно, что играть с подсказками намного интереснее. В нашем случае подсказками будут расстояния от введённых координат до всех монстров, которые ещё не пойманы.

Самый простой способ измерения расстояния — это сумма разностей координат: нужно сложить расстояние по горизонтали с расстоянием по вертикали[1].

C++

```
#include <cmath>

int dx = pos.x - m[i].x;
int dy = pos.y - m[i].y;
double dist = std::abs(dx)
    + std::abs(dy);
```

Python

```
dx = pos.x - m[i].x
dy = pos.y - m[i].y
dist = abs(dx) + abs(dy)
```

Функцию `abs` мы использовали для нахождения модуля числа. Ведь разность координат может быть как положительная, так и отрицательная.

Второй способ — использовать теорему Пифагора. Ведь если построить треугольник, в котором гипотенуза будет соединять наши точки, а катетами будут горизонтальное и вертикальное перемещения, соответствующие разностям координат, то длину гипотенузы (то есть кратчайшее расстояние между точками) можно посчитать так:

C++

```
double dist = std::sqrt(dx * dx
                        + dy * dy);
```

Python

```
import math

dist = math.sqrt(dx * dx + dy * dy)
```

Здесь уже брать разницу по модулю не нужно, потому что мы возводим эти числа в квадрат. Отрицательные числа станут положительными автоматически.



Поиграйте с обоими вариантами определения расстояния. Какой оказался более сложным? Сможете поймать всех чудовищ?

### 5.2.8. Финальная часть

Программа уже научилась сообщать нам, что монстры пойманы. Но, если поймать всех, игра всё равно продолжается до истечения всех 10 ходов. Чтобы это исправить, нужно доделать функцию `allCaught`. Она должна вернуть «истину», если свободно разгуливающих чудовищ уже не осталось. То есть все ячейки массива `caught` содержат «истину».

[1] Ещё это называется манхэттенским расстоянием.

C++

```
bool allCaught()
{
    for (int i = 0 ; i < M ; ++i)
    {
        if (!caught[i])
        {
            return false;
        }
    }
    return true;
}
```

Python

```
def allCaught():
    for c in caught:
        if not c:
            return False
    return True
```

Смотрите, как изменилась функция. Раньше она возвращала «ложь» всегда, а теперь только если найден непойманный монстр. Считать всех не понадобилось, ведь даже одного чудовища достаточно для продолжения игры.

Ну и последнее, что осталось сделать, это напечатать поздравления игроку, если ему удалось поймать всех чудовищ. Но ведь в функцию **finalMessage**, которая это делает, передаётся только номер последнего хода **move**. Как же она поймёт, выводить поздравление или нет? А очень просто — если монстры не были пойманы, **move** достигнет значения 11 и основной цикл завершится по условию из его заголовка. Значит, если **move** не превышает 10, то игра завершилась из-за поимки всех чудовищ.

C++

```
const int G = 10;
void finalMessage(int move)
{
  if (move <= G)
  {
    std::cout << "Поздравляем, вы поймали
    ↪ всех монстров за " << move << "
    ↪ ходов!\n";
  }
  else
  {
    std::cout << "Вы проиграли, можете
    ↪ попробовать снова.\n";
  }
}
```

Python

```
def finalMessage(move):
  if move <= G:
    print(f'Поздравляем, вы поймали всех
    ↪ монстров за {move} ходов!')
  else:
    print('Вы проиграли, можете
    ↪ попробовать снова.')
```

Тут мы заодно завели константу $G$ для числа ходов, чтобы дважды не писать 10. Ведь это число может измениться.

Вот и всё. Можете ещё доработать функцию **finalMessage**, чтобы она выводила, сколько чудовищ остались на свободе.

### Листинг 5.3. mugwump.cpp

```
#include <iostream>
#include <cmath>
```

```
const int M = 4, S = 10, G = 10;
struct Pos { int x, y; };
Pos m[M];
bool caught[M];

void placeMonsters()
{
    for (int i = 0 ; i < M ; ++i)
    {
        bool again = true;
        while (again)
        {
            m[i].x = rand() % S;
            m[i].y = rand() % S;
            again = false;
            for (int j = 0 ; j < i ; ++j)
            {
                if (m[i].x == m[j].x && m[i].y == m[j].y)
                {
                    again = true;
                }
            }
        }
    }
}

void printMonsters()
{
    std::cout << "Монстры: \n";
    for (int i = 0 ; i < M ; ++i)
    {
        std::cout << i << ": ";
        if (caught[i])
        {
            std::cout << "пойман \n";
        }
        else
        {
            std::cout << m[i].x << " " << m[i].y << "\n";
        }
```

```
    }
}

Pos inputMove()
{
    int x, y;
    do
    {
        std::cout << "Введите координаты: ";
        std::cin >> x >> y;
    }
    while (x < 0 || x >= S || y < 0 || y >= S);
    return Pos{x, y};
}

void checkDistance(Pos pos)
{
    for (int i = 0 ; i < M ; ++i)
    {
        if (caught[i])
        {
            continue;
        }
        std::cout << "Монстр " << i + 1;
        if (pos.x == m[i].x && pos.y == m[i].y)
        {
            caught[i] = true;
            std::cout << " пойман! \n";
        }
        else
        {
            int dx = pos.x - m[i].x;
            int dy = pos.y - m[i].y;
            double dist = std::sqrt(dx * dx + dy * dy);
            std::cout << " на расстоянии " << dist << "\n";
        }
    }
}

bool allCaught()
{
```

```
    for (int i = 0 ; i < M ; ++i)
    {
        if (!caught[i])
        {
            return false;
        }
    }
    return true;
}

void finalMessage(int move)
{
    if (move <= G)
    {
        std::cout << "Поздравляем, вы поймали всех монстров за " <<
        ↪ move << " ходов! \n";
    }
    else
    {
        std::cout << "Вы проиграли, можете попробовать снова. \n";
    }
}

int main()
{
    srand((int)time(NULL));
    int move;
    placeMonsters();
    for (move = 1 ; move <= G ; ++move)
    {
        std::cout << "Ход " << move << " \n";
        //printMonsters();
        checkDistance(inputMove());
        if (allCaught())
        {
            break;
        }
    }
    finalMessage(move);
}
```

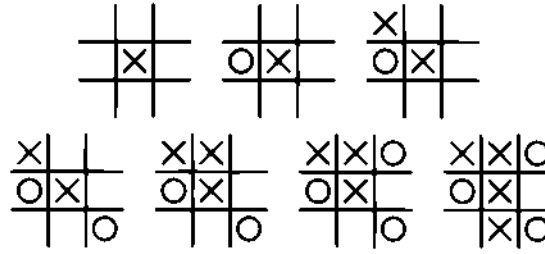
**Листинг 5.4. mugwump.py**

```
#!/usr/bin/python3
import random
from dataclasses import dataclass
import math

@dataclass
class Pos:
    x: int
    y: int
M = 4
S = 10
G = 10
m = [Pos(0, 0)] * M
caught = [False] * M

def printMonsters():
    print('Монстры:')
    for i in range(M):
        print(f'{i}: ', end='')
        if caught[i]:
            print('пойман')
        else:
            print(f'{m[i].x} {m[i].y}')

def placeMonsters():
    for i in range(M):
        again = True
        while again:
            m[i] = Pos(random.randint(0, S - 1), \
                    random.randint(0, S - 1))
            again = False
            for j in range(i):
                if m[i].x == m[j].x and m[i].y == m[j].y:
                    again = True

def inputMove():
    x = -1
    while x < 0 or x >= S or y < 0 or y >= S:
        x, y = [int(s) for s in input('Введите координаты: ').split()]
    return Pos(x, y)
```

```
def checkDistance(pos):
    for i in range(M):
        if caught[i]:
            continue
        print(f'Монстр {i + 1}', end='')
        if pos.x == m[i].x and pos.y == m[i].y:
            caught[i] = True;
            print(' пойман!')
        else:
            dx = pos.x - m[i].x
            dy = pos.y - m[i].y
            dist = math.sqrt(dx * dx + dy * dy)
            print(f' на расстоянии {dist:.3}')

def allCaught():
    for c in caught:
        if not c:
            return False
    return True

def finalMessage(move):
    if move <= G:
        print(f'Поздравляем, вы поймали всех монстров за {move}
        ↪ ходов!')
    else:
        print('Вы проиграли, можете попробовать снова.')

placeMonsters()
move = 1
while move <= G:
    print(f'Ход {move}')
    #printMonsters()
    checkDistance(inputMove())
    if allCaught():
        break
    move += 1
finalMessage(move)
```

## 5.3. Игра «Крестики-нолики»

Крестики-нолики известны всем. На поле 3 × 3 игроки поочерёдно ставят свои символы (крестики или нолики). Побеждает тот, кто первым выстроит ряд из трёх своих символов. Вот пример партии, где выигрывают крестики:



**Экран 1. Та же партия при игре на компьютере**

```
1 2 3
4 5 6
7 8 9

Введите ваш ход - номер
↪ клетки: 5
1 2 3
4 X 6
7 8 9

Мой ход:
1 2 3
0 X 6
7 8 9

Введите ваш ход - номер
↪ клетки: 1
X 2 3
0 X 6
7 8 9

Мой ход:
X 2 3
0 X 6
7 8 0

Введите ваш ход - номер
↪ клетки: 2
X X 3
0 X 6
7 8 0

Мой ход:
X X 0
0 X 6
7 8 0

Введите ваш ход - номер
↪ клетки: 8
X X 0
0 X 6
7 X 0

Вы победили!
```

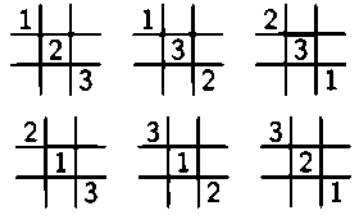
### 5.3.1. Игровая позиция

На первый взгляд, эта игра мало отличается от «Чудовищ» или «Ферзя в угол». Всё то же квадратное поле из клеточек, в которых расположены фигуры. Правда, на этот раз «фигуры» у нас двух видов — крестики и нолики.

Что будет, если поступить, как в прошлый раз? Придётся завести массив для хранения «фигур». Или даже два массива, один для крестиков, один для ноликов. Но что мы с этими массивами потом будем делать?

Дальше нам понадобится определять, победил ли кто-то из игроков. А для этого нужно проверить, стоят ли крестики (или нолики) на одной линии. Можете представить, как это выглядит в случае массива фигур? Ведь они никак не упорядочены, поэтому придётся находить все комбинации.

Вот пример с разным порядком заполнения главной диагонали:




Первый ход может быть в один из углов, а может быть в центр. Дальше для второго хода тоже остаётся два варианта. И всё это надо учесть, так как эти координаты будут записаны в массив в заранее неизвестном порядке.

Второй вариант хранения игровой позиции — это один массив на всё поле. Одна ячейка массива — одна клетка поля. Храниться там может либо крестик, либо нолик, либо признак того, что клетка пустая. Например, каждая клетка может быть одним символом. Тогда нам понадобится такая структура данных:

**C++**

```
char field[3][3];
```

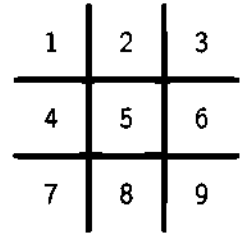
Для глобальной переменной инициализация не нужна, туда автоматически запишутся нули.

**Python**

```
field = [[None] * 3, [None] * 3, [None] *
↪ 3]
```

Тут хотелось бы написать `field = [[None] * 3] * 3`, но тогда всем строкам игрового поля будет соответствовать один и тот же объект-массив из трёх элементов. И если мы попытаемся его изменить, это повлияет на все строки сразу.

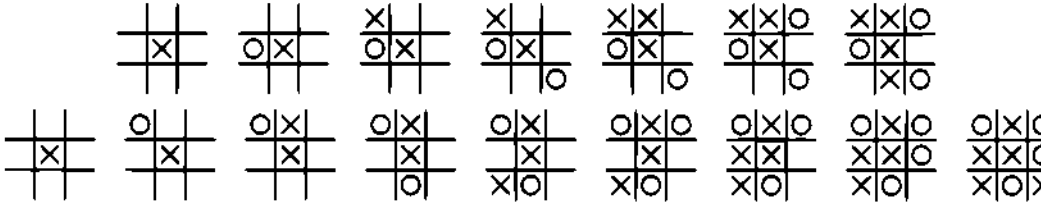
Изначально в ячейках нет полезных символов. Это будет означать, что клетки пустые. Для игрока (при выводе на экран и при вводе хода) координату клетки будем обозначать числом от 1 до 9:




Можно было бы и массив сделать одномерным, но тогда чуть сложнее станет код проверки выигрышной позиции. А двумерные массивы нам ещё пригодятся в более сложных играх, поэтому с ними полезно познакомиться прямо сейчас.

### 5.3.2. Основной цикл, вывод поля

Первыми в игре ходят крестики, но мы ещё не определили, за кого будет играть компьютер. Известно, что при правильной игре крестики никогда не проигрывают. Вот варианты развития событий. Как бы ни играли нолики, ничего не выходит:



Значит, если оставить право ходить первым достаточно хорошему алгоритму, то человеку будет совсем не интересно. Поэтому наша программа будет играть ноликами, чтобы у человека оставался хотя бы теоретический шанс на победу. Теоретический он, потому что при правильной игре нолики тоже не проигрывают. Но сможем ли мы построить настолько хороший алгоритм? Увидим позднее.

А сейчас подготовим основной цикл игры. В нём выводится игровое поле, потом делает ход программа, потом поле снова выводится и делает ход человек.

C++

```
while (true)
{
    printField();
    playerMove();
    printField();
    computerMove();
}
```

Python

```
while True:
    printField()
    playerMove()
    printField()
    computerMove()
```

Чего не хватает в этом цикле? Условия завершения. Из цикла нужно выходить, если кто-то победил или если не осталось пустых клеток. Это всё мы добавим позднее, а сейчас напишем функцию для вывода поля на экран.

C++

```
void printField()
{
  for (int i = 0 ; i < 3 ; ++i)
  {
    for (int j = 0 ; j < 3 ; ++j)
      if (field[i][j])
        std::cout << field[i][j];
      else
        std::cout << i * 3 + j + 1;
      std::cout << " ";
    std::cout << "\n";
  }
  std::cout << "\n";
}
```

Python

```
def printField():
  for i in range(0, 3):
    for j in range(0, 3):
      cell = field[i][j] \
             or i * 3 + j + 1
      print(f'{cell} ', end='')
    print()
  print()
```

Из номеров строк и столбца мы находим число, соответствующее номеру клетки, если она пустая.

В пустой ячейке игрового поля хранится `None`. Поэтому, когда нужно вывести номер такой ячейки на экран, применяется оператор `x or z`. Он возвращает `z`, если `x` содержит `None`. Поэтому когда ячейка массива хранит `None`, мы вычисляем её номер на основе строки и столбца и выводим его на экран.

### 5.3.3. Ход игрока, проверка условия победы

Продолжим заполнять недостающие функции. Первой будет `playerMove`, она должна получить от игрока номер клетки для его хода и проверить, что этот ход корректный. Потом нужно модифицировать игровое поле, чтобы там появился крестик в соответствующей клетке. Параметров и возвращаемых значений у функции нет, потому что человек всегда играет за крестиков, а игровое поле — это глобальная переменная.

C++

```
void playerMove()
{
  int i, j, move;
  do
  {
    std::cout << "Введите ваш ход: ";
    std::cin >> move;
    i = (move - 1) / 3;
    j = (move - 1) % 3;
  }
  while (move <= 0 || move > 9
    || field[i][j]);
  field[i][j] = 'X';
}
```

Python

```
def playerMove():
  i = -1
  j = -1
  while i < 0 or i >= 3 or \
    j < 0 or j >= 3 or \
    field[i][j] is not None:
    move = int(input('Введите ваш ход:
    ↪  '))
    i = (move - 1) // 3
    j = (move - 1) % 3
  field[i][j] = 'X'
```

Ещё нам понадобится функция для проверки условия победы. Удобно, если она будет возвращать «истину», когда на поле возник ряд (столбец, диагональ) из трёх одинаковых символов. Так как пустые клетки закодированы разными цифрами, проверять сам тип символа не нужно, из пустых клеток последовательности не получится. Функция вызывается после каждого хода. Ход каждого игрока может быть только победным, поэтому если условие «три символа в ряд» стало выполняться, то выиграл игрок, сделавший последний ход.

C++

```
bool checkWin()
{
  for (int i = 0 ; i < 3 ; ++i)
  {
    if (field[i][0]
      && field[i][0] == field[i][1]
      && field[i][1] == field[i][2])
    {
      return true;
    }
    if (field[0][i]
      && field[0][i] == field[1][i]
      && field[1][i] == field[2][i])
    {
      return true;
    }
  }
  return (field[0][0]
      && field[0][0] == field[1][1]
      && field[1][1] == field[2][2])
    || (field[0][2]
      && field[0][2] == field[1][1]
      && field[1][1] == field[2][0]);
}
```

Python

```
def checkWin():
  for i in range(3):
    if field[i][0] is not None and
    ↪  field[i][0] == field[i][1] ==
    ↪  field[1][2]:
      return True
    if field[0][i] is not None and
    ↪  field[0][i] == field[1][i] ==
    ↪  field[2][i]:
      return True

  return (field[0][0] is not None and
  ↪  field[0][0] == field[1][1] ==
  ↪  field[2][2]) \
    or (field[0][2] is not None and
    ↪  field[0][2] == field[1][1] ==
    ↪  field[2][0])
```

### 5.3.4. Ход компьютера

Ход компьютера — это самая сложная часть игровой программы. Даже в крестиках-ноликах нам придётся потратить значительные усилия на реализацию «искусственного интеллекта».

Начать стоит с простейшего варианта, чтобы отладить ранее созданные функции. Пусть компьютер делает случайные ходы, выбирая произвольную пустую клетку.

Структура кода будет такая же, как и для хода человека. Там мы считывали номер клетки для хода, а тут будем генерировать случайным образом столбец и строку. Для проверки корректности хода останется только условие, что соответствующая клетка пустая.

C++

```
void computerMove()
{
    int i, j;
    do
    {
        i = rand() % 3;
        j = rand() % 3;
    }
    while (field[i][j]);
    field[i][j] = '0';
}
```

Python

```
def computerMove():
    i = 0
    j = 0
    while field[i][j] is not None:
        i = random.randint(0, 2)
        j = random.randint(0, 2)
    field[i][j] = '0'
```

Теперь, когда мы сделали все запланированные функции, нужно доработать основной цикл, чтобы добавить туда проверку условия победы.

C++

```
while (true)
{
  printField();
  if (checkWin())
  {
    std::cout << "Я выиграл!\n";
    break;
  }
  playerMove();
  printField();
  if (checkWin())
  {
    std::cout << "Вы победили!\n";
    break;
  }
  computerMove();
}
```

Python

```
while True:
  printField()
  if checkWin():
    print('Я выиграл!')
    break
  playerMove()
  printField()
  if checkWin():
    print('Вы победили!')
    break
  computerMove()
```

Почему же условия расположены именно так? Это нужно, чтобы сообщение о завершении игры появлялось после обновлённого игрового поля на экране. Проверка победного условия перед первым ходом игрока выглядит странно, но это только для первой итерации цикла. На следующих итерациях она будет происходить уже после того, как компьютер сделает ход.

### 5.3.5. Осмысленные ходы компьютера

Конечно же, случайные ходы не приведут «искусственный интеллект» к победе. Нужно придумать алгоритм, чтобы в ходах программы был какой-то смысл.

Вы наверняка знаете, как играть в крестики-нолики правильно (то есть не проигрывать). Чтобы научить этому компьютер, алгоритм игры надо формализовать, разбить на понятные шаги. Для поля $3 \times 3$ всё довольно просто. Чтобы не проиграть (а если повезёт, то и выиграть), достаточно следующих правил:

Если вдруг играющий за крестики не занял центр первым ходом, тогда это стоит сделать ноликам.

Цель игрока — это победа. Поэтому если есть такая клетка, в которую можно сходить и сразу выиграть, то это и нужно делать!

Если выиграть не удалось, стоит помешать выиграть сопернику. Если появилось два крестика в одну линию, нужно их нейтрализовать, поставив нолик.

Когда занят центр, выгоднее делать ход в какой-нибудь угол. Так программа точно не проиграет. А может ещё и выиграть, если центр раньше был занят ноликами.

Ну и если хорошие ходы кончились, будем ходить в те клетки, что остались.

Переведём это всё в код. Самое простое — пытаться походить в центр. Напишем соответствующий код сразу.

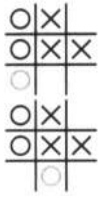
C++

```
void computerMove()
{
  int i, j;
  // ход в центр
  if (!field[1][1])
  {
    field[1][1] = '0';
    return;
  }
  // выигрышный ход
  ...
  // не дать выиграть сопернику
  ...
  // ход в угол
  ...
  // все остальные ходы
  do
  {
    i = rand() % 3;
    j = rand() % 3;
  }
  while (field[i][j]);
  field[i][j] = '0';
}
```

Python

```
def computerMove():
  # ход в центр
  if field[1][1] is None:
    field[1][1] = '0'
    return;
  # выигрышный ход
  ...
  # не дать выиграть сопернику
  ...
  # ход в угол
  ...
  # все остальные ходы
  i = 0
  j = 0
  while field[i][j] is not None:
    i = random.randint(0, 2)
    j = random.randint(0, 2)
  field[i][j] = '0'
```

### 5.3.6. Доделываем алгоритм игры



Из оставшихся вариантов ходов два очень похожи между собой. Это ход, когда мы выигрываем, а также ход, когда мы не даём выиграть сопернику. В обоих случаях нужно найти линию, где уже есть два одинаковых символа, а одна клетка пустая. Тогда нужно ставить в пустой клетке свой символ. Это приведёт или к победе, или к блокировке победного хода соперника.

Попробуем сделать поиск хода для обоих этих случаев в одной функции. Туда можно будет передавать символ, который мы проверяем: `'O'`, если ищем выигрышный ход, или `'X'`, если ищем возможность заблокировать противника.

Функция, которая перебирает разные линии на поле, у нас уже была — это поиск трёх одинаковых символов в ряд для проверки условия победы. Возьмём общую структуру кода оттуда:

C++

```
bool tryFinishLine(char xo)
{
  for (int i = 0 ; i < 3 ; ++i)
  {
    // Проверка i-й горизонтали
    ...
    // Проверка i-й вертикали
    ...
  }
  // Проверка одной диагонали
  ...
  // Проверка другой диагонали
  ...
  return false;
}
```

Python

```
def tryFinishLine(xo):
  for i in range(3):
    # Проверка i-й горизонтали
    ...
    # Проверка i-й вертикали
    ...
  # Проверка одной диагонали
  ...
  # Проверка другой диагонали
  ...
  return False
```

Функция `tryFinishLine` возвращает булево значение, чтобы уже в функции `computerMove` можно было проверить, удалось сделать ход или нет:

C++

```
// выигрышный ход
if (tryFinishLine('O'))
{
  return;
}
// не дать выиграть сопернику
if (tryFinishLine('X'))
{
  return;
}
```

Python

```
# выигрышный ход
if tryFinishLine('O'):
  return
# не дать выиграть сопернику
if tryFinishLine('X'):
  return
```

Сравнение только двух символов из трёх несколько сложнее, чем всех трёх. Тем более что после нахождения нужной комбинации в единственную пустую клетку найденной линии нужно будет поставить нолик.

### 5.3.7. Поиск завершения линии

Как же нам проверять линию на наличие двух одинаковых символов и одной пустой клетки? Всего-то нужно написать множество таких фрагментов кода:

C++

```
if (!field[i][0] && field[i][1] == xo
  && field[i][2] == xo)
{
  field[i][0] = '0';
  return true;
}
if (!field[i][1] && field[i][0] == xo
  && field[i][2] == xo)
{
  field[i][1] = '0';
  return true;
}
if (!field[i][2] && field[i][0] == xo
  && field[i][1] == xo)
{
  field[i][2] = '0';
  return true;
}
```

Python

```
if field[i][0] is None
  and field[i][1] == \
      field[i][2] == xo:
  field[i][0] = '0'
  return True
if field[i][1] is None
  and field[i][0] == \
      field[i][2] == xo:
  field[i][1] = '0'
  return True
if field[i][2] is None
  and field[i][0] == \
      field[i][1] == xo:
  field[i][2] = '0'
  return True
```

Получается довольно громоздко. А ведь это код только для горизонталей. Кроме него нужен ещё такой же набор ветвлений для вертикалей и для каждой из диагоналей. В подобных ситуациях копировать такие повторяющиеся (почти одинаковые) части — это плохая идея, потому что легко допустить ошибку, меняя скопированный код, так как эти фрагменты мало чем отличаются.

Как всегда, чтобы сделать однообразный громоздкий код лучше, нужно придумать функцию для повторяющихся частей. Здесь это группы по 3 ветвления. Так как каждая такая группа оперирует одной линией из трёх клеток, надо эту линию туда передать.

C++

```
int checkLine(char a, char b, char c,
↪  char xo)
{
  if (!a && b == xo && c == xo)
  {
    return 0;
  }
  if (!b && a == xo && c == xo)
  {
    return 1;
  }
  if (!c && a == xo && b == xo)
  {
    return 2;
  }
  return -1;
}
```

Python

```
def checkLine(a, b, c, xo):
  if a is None and b == c == xo:
    return 0
  if b is None and a == c == xo:
    return 1
  if c is None and a == b == xo:
    return 2
  return -1
```

Раз клеток всего три, то каждой из них сопоставляется один параметр функции. А четвёртый параметр — это символ, который мы хотим в этих клетках найти (то есть крестик или нолик). Возвращает функция номер пустой клетки (0, 1 или 2) из переданных трёх.

### 5.3.8. Проверяем все строки, столбцы и диагонали

Получается такая функция проверки «закрытия» линии. Она используется как для выигрышного хода (тогда ищет два нолика), так и для того чтобы помешать крестикам (тогда ищет место, где крестики вот-вот выиграют).

Кода получается всё равно много, но кто сказал, что искусственный интеллект — это легко.

Сначала напишем код для проверки горизонталей и вертикалей. Тут понадобится цикл, потому что их несколько:

C++

```
int t;
for (int i = 0 ; i < 3 ; ++i)
{
  // Проверка i-й горизонтали
  t = checkLine(field[i][0], field[i][1],
  ↪  field[i][2], xo);
  if (t >= 0)
  {
    field[i][t] = '0';
    return true;
  }
  // Проверка i-й вертикали
  t = checkLine(field[0][i], field[1][i],
  ↪  field[2][i], xo);
  if (t >= 0)
  {
    field[t][i] = '0';
    return true;
  }
}
```

Python

```
for i in range(3):
  # Проверка i-й горизонтали
  t = checkLine(field[i][0], field[i][1],
  ↪  field[i][2], xo)
  if t >= 0:
    field[i][t] = '0'
    return True
  # Проверка i-й вертикали
  t = checkLine(field[0][i], field[1][i],
  ↪  field[2][i], xo)
  if t >= 0:
    field[t][i] = '0'
    return True
```

А диагонали уже можно проверить с помощью всего пары вызовов `checkLine`, так как диагоналей всего две. С главной диагональю всё просто:

C++

```
t = checkLine(field[0][0], field[1][1],
↪  field[2][2], xo);
if (t >= 0)
{
  field[t][t] = '0';
  return true;
}
```

Python

```
t = checkLine(field[0][0], field[1][1],
↪  field[2][2], xo)
if t >= 0:
  field[t][t] = '0'
  return True
```

Со второстепенной диагональю нужна небольшая хитрость. Функция `checkLine` возвращает 0, 1 или 2 — номер пустой клетки-параметра. Нумерация строк для

клеток диагонали совпадает с этим числом. А столбцы идут в обратном порядке, так что найти их номера можно с помощью выражения $2 - t$.

C++

```
t = checkLine(field[0][2], field[1][1],
↪  field[2][0], xo);
if (t >= 0)
{
  field[t][2 - t] = '0';
  return true;
}
```

Python

```
t = checkLine(field[0][2], field[1][1],
↪  field[2][0], xo)
if t >= 0:
  field[t][2 - t] = '0'
  return True
```

Кажется, почти всё готово. Соберите все наши фрагменты кода и попробуйте поиграть. Когда вам удастся победить, подумайте, чего ещё не хватает программе, чтобы она всегда играла не хуже, чем вничью?

### 5.3.9. Всё для игры вничью

Наверняка вам удалось выяснить (или вспомнить), что мы не научили компьютер ставить нолик в угол, чтобы не проиграть. Это несложно, нужно всего лишь проверить каждый из углов. Так же, как мы проверяли центральную клетку.

C++

```
void computerMove()
{
  ...
  // ход в угол
  if (!field[0][0])
  {
    field[0][0] = '0';
    return;
  }
  if (!field[0][2])
  {
    field[0][2] = '0';
    return;
  }
  if (!field[2][2])
  {
    field[2][2] = '0';
    return;
  }
  if (!field[2][0])
  {
    field[2][0] = '0';
    return;
  }
  ...
}
```

Python

```
def computerMove():
  ...
  # ход в угол
  if field[0][0] is None:
    field[0][0] = '0'
    return
  if field[0][2] is None:
    field[0][2] = '0'
    return
  if field[2][2] is None:
    field[2][2] = '0'
    return
  if field[2][0] is None:
    field[2][0] = '0'
    return
  ...
```

Удалось сыграть с компьютером вничью? Но программа при этом зависла? Осталось сделать одну маленькую доработку — выход из программы, если больше нет свободных клеток.

Проверять, что все клетки заняты, нет никакого смысла, ведь на каждом ходу заполняется лишь одна. Так что добавьте либо счётчик ходов одного из игроков, либо счётчик занятых клеток, либо можно даже счётчик пустых клеток. И проверяйте внутри игрового цикла, чтобы завершить или сам цикл, или всю программу.

Когда закончите, сверьтесь с полным текстом программы на следующих страницах. Совпадёт ли ваше решение с тем, что используется там?

**Листинг 5.5. tictactoe.cpp**

```
#include <iostream>

char field[3][3];

int checkLine(char a, char b, char c, char xo)
{
  if (!a && b == xo && c == xo)
  {
    return 0;
  }
  if (!b && a == xo && c == xo)
  {
    return 1;
  }
  if (!c && a == xo && b == xo)
  {
    return 2;
  }
  return -1;
}

bool tryFinishLine(char xo)
{
  int t;
  for (int i = 0 ; i < 3 ; ++i)
  {
    // Проверка i-й горизонтали
    t = checkLine(field[i][0], field[i][1], field[i][2], xo);
```

```
    if (t >= 0)
    {
      field[i][t] = '0';
      return true;
    }
    // Проверка i-й вертикали
    t = checkLine(field[0][i], field[1][i], field[2][i], xo);
    if (t >= 0)
    {
      field[t][i] = '0';
      return true;
    }
  }
  // Проверка одной диагонали
  t = checkLine(field[0][0], field[1][1], field[2][2], xo);
  if (t >= 0)
  {
    field[t][t] = '0';
    return true;
  }
  // Проверка другой диагонали
  t = checkLine(field[0][2], field[1][1], field[2][0], xo);
  if (t >= 0)
  {
    field[t][2 - t] = '0';
    return true;
  }
  return false;
}

void computerMove()
{
  int i, j;
  // ход в центр
  if (!field[1][1])
  {
    field[1][1] = '0';
    return;
  }
  // выигрышный ход
  if (tryFinishLine('0'))
```

```
  {
    return;
  }
  // не дать выиграть сопернику
  if (tryFinishLine('X'))
  {
    return;
  }
  // ход в угол
  if (!field[0][0])
  {
    field[0][0] = '0';
    return;
  }
  if (!field[0][2])
  {
    field[0][2] = '0';
    return;
  }
  if (!field[2][2])
  {
    field[2][2] = '0';
    return;
  }
  if (!field[2][0])
  {
    field[2][0] = '0';
    return;
  }
  // все остальные ходы
  do
  {
    i = rand() % 3;
    j = rand() % 3;
  }
  while (field[i][j]);
  field[i][j] = '0';
}

bool checkWin()
{
```

```
  for (int i = 0 ; i < 3 ; ++i)
  {
    if (field[i][0] && field[i][0] == field[i][1] && field[i][1] ==
    ↪ field[i][2])
    {
      return true;
    }
    if (field[0][i] && field[0][i] == field[1][i] && field[1][i] ==
    ↪ field[2][i])
    {
      return true;
    }
  }
  return (field[0][0] && field[0][0] == field[1][1] && field[1][1] ==
  ↪ field[2][2])
    || (field[0][2] && field[0][2] == field[1][1] && field[1][1] ==
    ↪ field[2][0]);
}

void playerMove()
{
  int i, j, move;
  do
  {
    std::cout << "Введите ваш ход - номер клетки: ";
    std::cin >> move;
    i = (move - 1) / 3;
    j = (move - 1) % 3;
  }
  while (move <= 0 || move > 9
    || field[i][j]);
  field[i][j] = 'X';
}

void printField()
{
  for (int i = 0 ; i < 3 ; ++i)
  {
    for (int j = 0 ; j < 3 ; ++j)
    {
      if (field[i][j])
```

```
        std::cout << field[i][j] << " ";
      else
        std::cout << i * 3 + j + 1 << " ";
    }
    std::cout << "\n";
  }
  std::cout << "\n";
}

int main()
{
  int moves = 0;
  while (true)
  {
    printField();
    if (checkWin())
    {
      std::cout << "Я выиграл!\n";
      break;
    }
    playerMove();
    ++moves;
    printField();
    if (checkWin())
    {
      std::cout << "Вы победили!\n";
      break;
    }
    if (moves == 9)
    {
      std::cout << "Ничья!\n";
      break;
    }
    std::cout << "Мой ход:\n";
    computerMove();
    ++moves;
  }
}
```

**Листинг 5.6. tictactoe.py**

```
#!/usr/bin/python3
import random

field = [[None] * 3, [None] * 3, [None] * 3]

def checkLine(a, b, c, xo):
  if a is None and b == c == xo:
    return 0
  if b is None and a == c == xo:
    return 1
  if c is None and a == b == xo:
    return 2
  return -1

def tryFinishLine(xo):
  for i in range(3):
    # Проверка i-й горизонтали
    t = checkLine(field[i][0], field[i][1], field[i][2], xo)
    if t >= 0:
      field[i][t] = '0'
      return True
    # Проверка i-й вертикали
    t = checkLine(field[0][i], field[1][i], field[2][i], xo)
    if t >= 0:
      field[t][i] = '0'
      return True
  # Проверка одной диагонали
  t = checkLine(field[0][0], field[1][1], field[2][2], xo)
  if t >= 0:
    field[t][t] = '0'
    return True
  # Проверка другой диагонали
  t = checkLine(field[0][2], field[1][1], field[2][0], xo)
  if t >= 0:
    field[t][2 - t] = '0'
    return True
  return False

def computerMove():
  # ход в центр
```

```
  if field[1][1] is None:
    field[1][1] = 'O'
    return
  # выигрышный ход
  if tryFinishLine('O'):
    return
  # не дать выиграть сопернику
  if tryFinishLine('X'):
    return
  # ход в угол
  if field[0][0] is None:
    field[0][0] = 'O'
    return
  if field[0][2] is None:
    field[0][2] = 'O'
    return
  if field[2][2] is None:
    field[2][2] = 'O'
    return
  if field[2][0] is None:
    field[2][0] = 'O'
    return
  # все остальные ходы
  i = 0
  j = 0
  while field[i][j] is not None:
    i = random.randint(0, 2)
    j = random.randint(0, 2)
  field[i][j] = 'O'

def checkWin():
  for i in range(3):
    if field[i][0] is not None and field[i][0] == field[i][1] ==
    ↪ field[i][2]:
      return True
    if field[0][i] is not None and field[0][i] == field[1][i] ==
    ↪ field[2][i]:
      return True

  return (field[0][0] is not None and field[0][0] == field[1][1] ==
  ↪ field[2][2]) \
```

```
    or (field[0][2] is not None and field[0][2] == field[1][1] ==
    ↪ field[2][0])

def playerMove():
  i = -1
  j = -1
  while i < 0 or i >= 3 or j < 0 or j >= 3 or field[i][j] is not None:
    move = int(input('Введите ваш ход - номер клетки: '))
    i = (move - 1) // 3
    j = (move - 1) % 3
  field[i][j] = 'X'

def printField():
  for i in range(0, 3):
    for j in range(0, 3):
      cell = field[i][j] or i * 3 + j + 1
      print(f'{cell} ', end='')
    print()
  print()

moves = 0
while True:
  printField()
  if checkWin():
    print('Я выиграл!')
    break
  playerMove()
  moves += 1
  printField()
  if checkWin():
    print('Вы победили!')
    break
  if moves == 9:
    print('Ничья!')
    break
  print('Мой ход:')
  computerMove()
  moves += 1
```

Глава 6

# Полный перебор с помощью циклов

## 6.1. Числовые ребусы с буквами

Эта глава начинается не с игр, а с головоломок. В каком-то смысле это тоже игры, только вы соревнуетесь с их автором, а не с человеком (или компьютером), который находится рядом.

Числовые ребусы или криптарифмы — это арифметические выражения, в которых все цифры заменены буквами. Вот классический пример такого ребуса, опубликованный в журнале Strand Magazine в 1924 году:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Каждая буква обозначает одну цифру, а одинаковые цифры заменяются одинаковыми буквами. Такие ребусы можно решать логически, но мы вообще-то тут программируем, поэтому пока отложим рассуждения.

Чтобы разобраться, как с помощью программы решить ребус, сначала возьмём пример попроще:

$$TO + GO = OUT$$

Решать ребус будем полным перебором. Полный перебор — это поиск нужного решения среди всех возможных вариантов решений. Чтобы организовать перебор, нужно определить:

- множество переменных, из которых состоит решение;
- какие значения могут принимать переменные;
- какой алгоритм использовать для перебора;
- ограничения, накладываемые на всё множество.

### 6.1.1. Начинаем писать программу

В ребусах множество переменных соответствует тем буквам, которые мы хотим подобрать. Для второго ребуса это $T$, $O$, $G$, $U$. Цифр в ребусе 7, а переменных всего 4, потому что некоторые буквы повторяются. Будем дальше обозначать наши переменные такими же буквами, как в ребусе.

Так как буквы соответствуют цифрам, то они будут принимать значения от 0 до 9. Значит, в программе для них можно использовать целочисленные переменные.

C++

```
int main()
{
    int T, O, G, U;
}
```

Python

```
# в Python не нужно
# объявлять переменные :)
```

### 6.1.2. Добавляем код для перебора

Перебор заключается в том, что проверяются все возможные комбинации значений переменных. Например, для трёх переменных, которые принимают значения от 0 до 2, будет 27 вариантов:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 000 | 001 | 002 | 010 | 011 | 012 | 020 | 021 | 022 |
| 100 | 101 | 102 | 110 | 111 | 112 | 120 | 121 | 122 |
| 200 | 201 | 202 | 210 | 211 | 212 | 220 | 221 | 222 |

Есть множество способов программно найти все такие комбинации. В том случае когда переменных немного, а количество их зафиксировано, целесообразно использовать вложенные циклы. Позднее мы будем использовать другой вид перебора — рекурсивный перебор, для более сложных игр.

Например, для генерации комбинаций чисел, приведённых выше, можно написать такие циклы:

C++

```
for (int i = 0 ; i <= 2 ; ++i)
    for (int j = 0 ; j <= 2 ; ++j)
        for (int k = 0 ; k <= 2 ; ++k)
            std::cout << i << j << k
                      << "\n";
```

Python

```
for i in range(3):
    for j in range(3):
        for k in range(3):
            print(f'{i}{j}{k} ')
```

Перебор происходит так: $i$ в первом цикле пробегает значения от 0 до 2. Например, это можно представить как перемещение со строки на строку в списке чисел выше.

Для каждой строки выполняется следующий цикл, в котором $j$ пробегает значения от 0 до 2. А в каждой итерации этого цикла снова выполняется цикл для $k$ в диапазоне от 0 до 2. Эти две переменные отвечают за перемещение внутри строки. В последнем цикле переменные выводятся на экран, чтобы показать, что все варианты пройдены.

Примерно то же самое можно сделать и с переменными для ребуса. Всего будет 4 переменных со значениями от 0 до 9.

C++

```
for (T = 0 ; T <= 9 ; ++T)
  for (O = 0 ; O <= 9 ; ++O)
    for (G = 0 ; G <= 9 ; ++G)
      for (U = 0 ; U <= 9 ; ++U)
        std::cout << T << O << G
                  << U << "\n";
```

Python

```
for T in range(10):
  for O in range(10):
    for G in range(10):
      for U in range(10):
        print(f'{T}{O}{G}{U} ')
```

Попробуйте вставить этот код в свою программу и посмотреть, что она выведет.

Если вы используете неудачный шрифт в редакторе исходного кода, то переменную $O$ можно перепутать с цифрой 0. Поэтому в серьёзных программах такое имя переменной использовать не стоит.

### 6.1.3. Проверяем ограничения

Программа, которая получилась на этот момент, просто выведет все возможные четырёхзначные комбинации из цифр. Наша задача — выбрать из этих комбинаций ту, которая подходит в качестве решения.

Обратим внимание на выражение $TO + GO = OUT$. Так как в этом выражении три числа, а не просто цифры, можно сделать вывод, что на первом месте в каждом из чисел не могут стоять нули. Значит, перебор для $T$, $G$, $O$ надо начинать не с нуля, а с единицы.

Кроме того, цифры должны различаться. Для проверки этого добавим условие перед выводом «ответа»:

C++
```
for (T = 1 ; T <= 9 ; ++T)
  for (O = 1 ; O <= 9 ; ++O)
    for (G = 1 ; G <= 9 ; ++G)
      for (U = 0 ; U <= 9 ; ++U)
        if (T != O && T != G
          && T != U && O != G
          && O != U && G != U)
          std::cout << T << O << G
                    << U << "\n";
```

Python
```
for T in range(1, 10):
  for O in range(1, 10):
    for G in range(1, 10):
      for U in range(10):
        if T != O and T != G
          and T != U and O != G
          and O != U and G != U:
          print(f'{T}{O}{G}{U}')
```

Но всё равно вариантов слишком много. Осталось последнее условие — само равенство, составленное из букв-цифр. Если переписать ребус на язык программирования, то получится такое условие:

```
10*T + O + 10 * G + O == 100 * O + 10 * U + T.
```

⌨ | Вставьте последнее условие в код. Сколько теперь печатается вариантов?

Осталось добавить красивый вывод, чтобы ответ получался в виде арифметического выражения:

C++
```
std::cout << T << O << '+' << G << O <<
↪  '=' << O << U << T << '\n';
```

Python
```
print(f'{T}{O}+{G}{O}={O}{U}{T}')
```

И ещё одно улучшение. Условия, ограничивающие перебор, лучше проверять как можно раньше:

C++
```
int T, O, G, U;
for (T = 1 ; T <= 9 ; ++T)
  for (O = 1 ; O <= 9 ; ++O)
    if (T != O)
      for (G = 1 ; G <= 9 ; ++G)
        if (T != G && O != G)
          for (U = 0 ; U <= 9 ; ++U)
            if (T != U && O != U
              && G != U ...
```

Python
```
for T in range(1, 10):
  for O in range(1, 10):
    if T != O:
      for G in range(1, 10):
        if T != G && O != G:
          for U in range(10):
            if T != U and O != U
              and G != U ...
```

Ведь если $T = 1$ и $O = 1$, нет никакого смысла выбирать значения для $G$ и $U$, все эти варианты будут бесполезными, из-за того что две переменные совпадают. Поэтому условия, проверяющие уникальность значений, лучше вынести повыше.

### 6.1.4. Send more money!

Вернёмся к этому ребусу:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Здесь переменных будет уже 8: $S$, $E$, $N$, $D$, $M$, $O$, $R$, $Y$. Придётся написать 8 вложенных циклов, так что программа получится не очень красивая. Для нашей одноразовой задачи это подходит неплохо, но в реальной жизни чаще используется перебор с возвратом, к которому мы обратимся в другой главе.

Предлагаем вам написать эту программу самостоятельно.
Но если возникнут какие-то сложности, на следующих страницах вы найдёте готовый листинг.

### 6.1.5. Задания для самостоятельной работы

**1.** Упростите арифметические выражения в программе **sendmoremoney**. Подумайте, от каких переменных можно избавиться, так как их значения и так легко найти. Но не слишком углубляйтесь в рассуждения, ведь если довести их до конца, то программа не понадобится совсем.

**2.** Другие ребусы для самостоятельного решения можно найти в книгах [6], [5] или в Википедии.

**Листинг 6.1. sendmoremoney.cpp**

```
#include <iostream>

int main()
{
  // SEND + MORE = MONEY
  for (int S = 1 ; S <= 9 ; ++S)
  {
    for (int E = 0 ; E <= 9 ; ++E)
    {
      for (int N = 0 ; N <= 9 ; ++N)
      {
```

```
        for (int D = 0 ; D <= 9 ; ++D)
        {
          for (int M = 1 ; M <= 9 ; ++M)
          {
            for (int O = 0 ; O <= 9 ; ++O)
            {
              for (int R = 0 ; R <= 9 ; ++R)
              {
                for (int Y = 0 ; Y <= 9 ; ++Y)
                {
                  if (S != E && S != N && S != D
                    && S != M && S != O && S != R
                    && S != Y && E != N && E != D
                    && E != M && E != O && E != R
                    && E != Y && N != D && N != M
                    && N != O && N != R && N != Y
                    && D != M && D != O && D != R
                    && D != Y && M != O && M != R
                    && M != Y && O != R && O != Y
                    && R != Y
                    && S * 1000 + E * 100 + N * 10 + D
                    + M * 1000 + O * 100 + R * 10 + E
                    == M * 10000 + O * 1000 + N * 100
                    + E * 10 + Y)
                  {
                    std::cout << "SEND+MORE=MONEY\n";
                    std::cout << S << E << N << D << '+'
                      << M << O << R << E << '='
                      << M << O << N << E << Y << '\n';
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

**Листинг 6.2. sendmoremoney.py**

```
#!/usr/bin/python3

# SEND + MORE = MONEY
for S in range(1, 10):
  for E in range(10):
    for N in range(10):
      for D in range(10):
        for M in range(1, 10):
          for O in range(10):
            for R in range(10):
              for Y in range(10):
                if S != E and S != N and S != D \
                  and S != M and S != O and S != R \
                  and S != Y and E != N and E != D \
                  and E != M and E != O and E != R \
                  and E != Y and N != D and N != M \
                  and N != O and N != R and N != Y \
                  and D != M and D != O and D != R \
                  and D != Y and M != O and M != R \
                  and M != Y and O != R and O != Y \
                  and R != Y \
                  and S * 1000 + E * 100 + N * 10 + D \
                  + M * 1000 + O * 100 + R * 10 + E \
                  == M * 10000 + O * 1000 + N * 100 \
                  + E * 10 + Y:
                    print("SEND+MORE=MONEY")
                    print(f"{S}{E}{N}{D} + {M}{O}{R}{E} = {M}{O}{N}
                    ↪  {E}{Y}")
```

## 6.2. Игра «Быки и коровы»

Игра «Быки и коровы» чем-то похожа на игру «Отгадай число», которую мы уже разбирали. В ней один игрок также загадывает число, а другой пытается отгадать.

Но в этой игре число обязательно состоит из заранее известного количества цифр, а подсказки становятся более сложными: на каждую попытку отгадать число игрок получает сообщение вида «В числе $X$ Быков и $Y$ Коров». «Бык» — это цифра из попытки, которая есть и в загаданном числе, причём на той же позиции. «Корова» — это тоже цифра, которая есть в загаданном числе, но расположена на другой позиции. Таким образом, подсказка состоит из количества верно отгаданных цифр (быков), а также почти отгаданных (коров).

О том, сколько может быть цифр в загаданном числе, нужно условиться заранее. Есть вариант с тремя цифрами [1], но так играть слишком легко. Есть настольная игра «Mastermind» [11],где вместо цифр цветные шарики, причём цвета могут повторяться.

Но мы будем программировать другой классический вариант, где загаданное число состоит из четырёх различных цифр [3, 5, 7]. Собственно, именно этот вариант обычно и называется «Быки и коровы». Рассмотрим пример игры.

**Экран 1. Игра в «Быки и коровы»**

```
Введите последовательность из 4 цифр: 1234
Быки: 0 Коровы: 1
Введите последовательность из 4 цифр: 5678
Быки: 0 Коровы: 2
Введите последовательность из 4 цифр: 7819
Быки: 0 Коровы: 3
Введите последовательность из 4 цифр: 8701
Быки: 1 Коровы: 1
Введите последовательность из 4 цифр: 8749
Быки: 2 Коровы: 2
Введите последовательность из 4 цифр: 8947
Вы выиграли!
```

Последний ход в этой партии был 8947. Значит, такое число и было загадано. Сравним его с первым ходом — 1234. Из этой попытки в загаданном числе встречается только цифра 4, но она на третьей позиции, а не на четвёртой. Поэтому подсказкой после этой попытки было сообщение «Быки: 0 Коровы: 1».

Теперь рассмотрим четвёртый ход — 8701. В нём две цифры из секретного числа — 8 и 7. Цифра 8 стоит на нужном месте, значит, это «бык». 7 стоит на второй позиции вместо четвёртой, поэтому это «корова».

Всего можно попробовать 9999 разных чисел[1], поэтому, чтобы выиграть быстрее, нужно использовать подсказки ведущего. В нашем примере после хода 1234 становится ясно, что из этих цифр только одна есть в загаданном числе. Поэтому в следующих попытках не стоит использовать из этого набора больше одной цифры одновременно.

С правилами мы разобрались, теперь можно приступить к разработке алгоритма игры. Пусть сначала играть будет человек, а компьютер загадает ему число и даёт подсказки.

### 6.2.1. Отгадывает человек

Как мы уже выяснили, алгоритмы многих игр укладываются в простую схему: программа то получает от пользователя новые данные, то выводит что-то для него. И всё это происходит в цикле, пока игра не завершится.

Так же можно описать игру «Быки и коровы» (рис. 6.1). Сначала генерируется случайное число `secret`. Потом работает цикл, который завершается, если игрок отгадал комбинацию. В каждой итерации цикла игрок вводит в программу новую попытку `guess`. Программа в ответ выводит число «быков» и «коров» для этой попытки.

После того как алгоритм готов, можно придумывать программу. Сначала мы разобьём её на удобные для реализации части, а потом закодируем каждую из них.

### 6.2.2. Основные функции

Придумаем для шагов алгоритма названия, чтобы создать для каждого из них свою функцию:

- Сгенерировать `secret` — `generate`.
- Ввести `guess` — `inputGuess`.
- Вычислить число «быков» и «коров» — `countBullsAndCows`.

Программу удобнее писать «сверху вниз», то есть сначала определить её структуру, а потом заполнять нужными действиями. Поэтому создадим три функции, перечисленные выше. Так как мы не решили, как эти функции будут работать, то они пока пустые.

[1] На самом деле поменьше, их $10 \cdot 9 \cdot 8 \cdot 7 = 5040$, так как все цифры должны различаться.

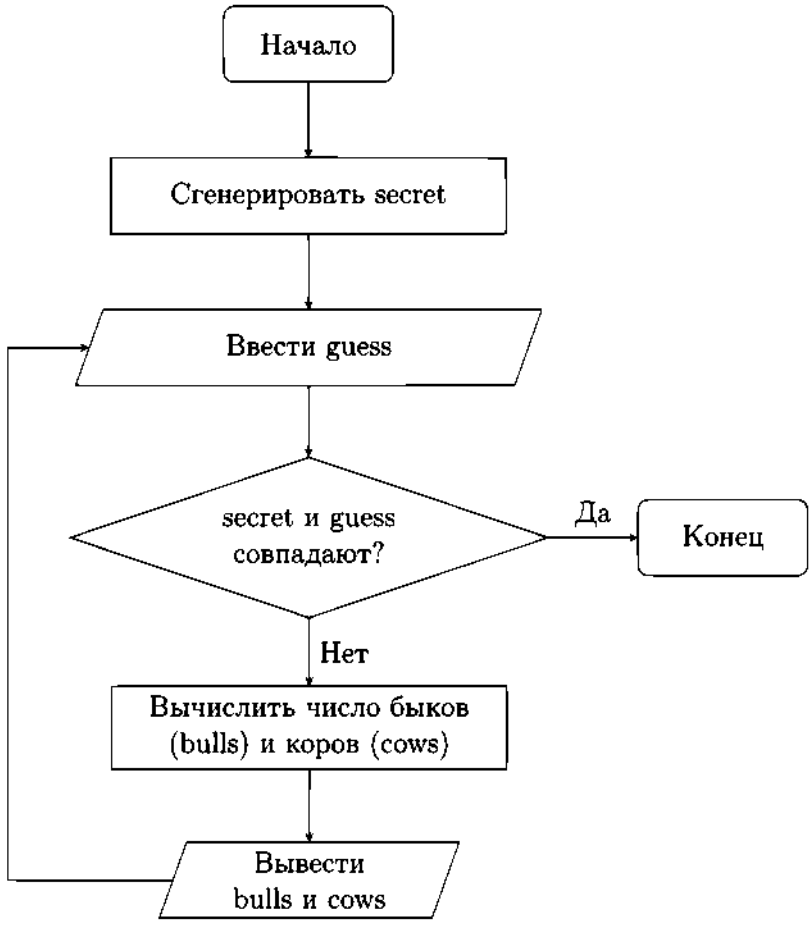



Рис. 6.1. Алгоритм игры «Быки и коровы»

**C++**

```
int generate()
{
    return 0;
}

int inputGuess()
{
    return 0;
}

void countBullsAndCows(int secret,
                       int guess)
{
}
```

**Python**

```
def generate():
    return 0

def inputGuess():
    return 0

def countBullsAndCows(secret, guess):
    pass
```

Функции `generate` и `inputGuess` состоят только из оператора `return 0`, потому что первая должна возвращать сгенерированное число, а вторая введённую попытку. В обоих случаях надо будет вставить вместо этого какой-то полезный код.

Конечно же, никто не заставляет реализовывать наш алгоритм, используя именно функции. Можно всё это реализовывать последовательно, в виде циклов и ветвлений, повторяя блок-схему. Но чем программа больше, тем удобнее становится её читать, когда она разбита на функции. А без чтения программ не обойтись, потому что их периодически нужно исправлять и усовершенствовать.

### 6.2.3. Основной цикл игры

Теперь мы можем описать основной алгоритм, как на блок-схеме, но уже используя язык программирования.

C++

```
int main()
{
  int secret = generate();
  int guess;
  while (true)
  {
    guess = inputGuess();
    if (guess == secret)
    {
       break;
    }
    countBullsAndCows(secret, guess);
  }
}
```

Python

```
secret = generate()
while True:
  guess = inputGuess()
  if secret == guess:
    break
  countBullsAndCows(secret, guess)
```

Заметьте, что действие «вывод быков и коров» спряталось в несуществующей пока функции `countBullsAndCows`.

Из этого и предыдущего фрагментов кода получается «работающая» программа. Её можно скомпилировать и запустить, но никакого видимого эффекта это не даст, потому что там нет ни ввода, ни вывода.

Зато программировать дальше будет очень легко — надо только поочерёдно заполнить все функции полезным содержимым, чтобы они делали то, на что указывает их название.

Соберите вместе фрагменты кода, которые мы уже написали. Не забудьте, что в начале программы надо разместить вспомогательные функции, а ниже их использование в основном цикле.

Потом скомпилируйте (если используете C++) и запустите программу. Если никаких ошибок не появилось, то это успех! Завершите программу комбинацией клавиш Ctrl+C.

### 6.2.4. Как хранить загаданное число

Алгоритм начинается с генерации значения `secret`. Во фрагментах кода выше предполагается, что это переменная имеет целочисленный тип. Это логично, потому что мы же храним в ней последовательность из нескольких цифр. Но насколько это удобно в работе?

Подумаем, что нам нужно делать с этой переменной. Во-первых, мы должны генерировать случайное число, а во-вторых, искать в нём цифры из другой последовательности. Искать цифры в разных местах нужно, потому что мы должны посчитать число «коров».

С генерацией разберёмся попозже, а сейчас рассмотрим подзадачу поиска цифр в числе. Чтобы узнать, совпадает ли цифра, которая хранится в переменной `d`, например, с третьей цифрой числа в переменной `secret`, нужно проверить следующее условие: `d == (secret / 10) % 10`. И, так как делать это потребуется для разных позиций, придётся написать цикл.

C++

```
int secret, d;
...
int x = secret;
bool found = false;
for (int i = 0 ; i < 4 ; ++i)
{
    if (d == x % 10)
    {
        found = true;
    }
    x /= 10;
}
```

Python

```
x = int(secret)
found = False
for i in range(4):
    if d == x % 10:
        found = True
    x = x // 10
```

В коде выше мы устанавливаем булеву переменную `found`, если цифра, хранящаяся в переменной `d`, была найдена в последовательности `secret`.

Другой вариант хранения загаданной комбинации — это строка символов. Тогда для проверки наличия в ней какой-либо цифры можно использовать встроенную функцию `find`.

C++

```
std::string secret;
char d;
...
found = secret.find(d)
        != std::string::npos;
```

Python

```
found = secret.find(d) != -1
```

Единственный недостаток при хранении загаданного числа в строковой переменной в том, что оно занимает больше памяти. Для нашей программы это несущественно, значит, будем использовать строки.

Куски кода отсюда пока копировать не нужно. Но раз уж мы решили использовать строки, то программу на C++ нужно немного поправить: вставить `#include <string>` в её начало и заменить типы возвращаемых значений для функций с `int` на `std::string`.
Для Python тоже можно заменить `return 0` на `return "0000"`. Но эту строку мы всё равно потом удалим, поэтому можно этого и не делать.

### 6.2.5. Функция generate — генерация случайного числа

Теперь разберёмся с генерацией числа. Можно попробовать просто получить четырёхзначное случайное число: `1000 + rand() % 9000`, а потом преобразовать его в строку. Но что делать, если в этом случае получается последовательность вроде 5662? Ведь одинаковые цифры использовать нельзя.

Правильный способ — это выбрать 4 случайные цифры из набора 0123456789. После каждого выбора использованную цифру нужно убирать, тогда все получаемые цифры будут разные.

Этот фрагмент кода уже можно вставить в функцию `generate`. Играть ещё не получится, ведь мы пока не написали никакой процедуры пользовательского ввода.

**C++**

```
std::string secret = "0123456789";
for (int i = 0 ; i < 4 ; ++i)
{
    int n = i + rand() % (10 - i);
    std::swap(res[i], res[n]);
}
res.resize(4);
```

В C++ выбрать случайный символ из строки (или массива) очень легко — надо сгенерировать индекс этого символа в диапазоне от нуля до размера исходного множества минус один.

Чтобы не возиться с удалением этого символа из множества, будем переставлять его в начало последовательности, а следующий генерируемый индекс начинать уже не с нуля, а с единицы, двойки, тройки.

В конце просто уменьшим размер исходной строки до четырёх, потому что четыре первые позиции уже будут занимать случайно выбранные символы.

**Python**

```
set = random.sample('0123456789', 4)
secret = ''.join(set)
```

В Python есть библиотечная функция `random.sample`, которая выбирает несколько различных элементов из множества. Первый её параметр — это множество элементов для выбора. Второй — число выбираемых элементов.

Возвращает функция список, поэтому потом его элементы склеиваются в одну строку функцией `join`.

Этот фрагмент уже можно вставить в нашу программу. Чтобы она запустилась, потребуется подключить библиотеки: `import random` для Python и `#include <time.h>` с `#include <stdlib.h>` для C++.
Правда, отгадать сгенерированное число пока не выйдет. Но вы можете временно добавить его вывод после вызова `generate`, чтобы убедиться, что всё работает.

### 6.2.6. Функция inputGuess — ввод числа игроком

Ввода числа или строки сам по себе несложен. Но нужно проверить, что игрок ввёл корректную последовательность:

- Длина строки ровно 4.
- В ней используются только цифры.
- Повторяющихся цифр в ней нет.

Первое условие проверить легко, вычислив длину введённой строки. Для второго и третьего потребуется цикл, чтобы проверять каждый символ. Этот цикл для красоты вынесен в отдельную функцию `check`. А ещё такая же функция нам понадобится в следующей программе.

C++

```
bool check(std::string s)
{
  for (int i = 0 ; i < 4 ; ++i)
  {
    if (s[i] < '0' || s[i] > '9'
      || s.find(s[i], i + 1)
         != std::string::npos)
    {
      return false;
    }
  }
  return true;
}

std::string inputGuess()
{
  std::string s;
  while (true)
  {
    std::cin >> s;
    if (s.size() == 4 && check(s))
    {
      return s;
    }
  }
}
```

Python

```
def check(s):
  for i in range(LEN):
    if s.find(s[i], i + 1) != -1:
      return False
  return True

def inputGuess():
  while True:
    s = input()
    if len(s) == LEN
      and s.isdigit()
      and check(s):
      return s
```

В этот раз фрагменты кода почти одинаковые. Например, в функции `find` в обоих языках есть второй параметр «откуда начинать поиск». Единственное

отличие — функция `isdigit` в Python, которая проверяет, что строка состоит из цифр. В C++ есть подобная функция для отдельных символов, но мы просто написали явное условие.

В этом фрагменте используется ввод с клавиатуры, которого не было в уже написанных функциях. Поэтому в программе на C++ нужно добавить подключение библиотеки ввода-вывода: `#include <iostream>`.

Теперь, когда есть и генерация числа, и ввод попытки, с программой можно даже поиграть. Подсказок не будет, так что это почти как угадать бросок кубика. Только вариантов будет $10 \cdot 9 \cdot 8 \cdot 7 = 5040$ вместо 6.

### 6.2.7. Функция countBullsAndCows — подсчёт «быков» и «коров»

Чтобы посчитать число «быков» и «коров» для пробного числа от игрока, нужно проверить каждую цифру этого числа. Если цифра совпадает с находящейся на той же позиции цифрой секретного числа, то увеличиваем счётчик «быков»

В противном случае нужно проверить, не является ли цифра «коровой». Это значит, что она должна быть где-то в секретной строке. Для этого снова можно использовать функцию `find`.

`find` найдёт цифру, даже если она стоит на той же позиции, значит, условие проверки «быков» надо поставить раньше, чтобы их не пропустить.

**C++**

```
int bulls = 0;
int cows = 0;
for (int i = 0 ; i < 4 ; ++i)
{
    if (guess[i] == secret[i])
        ++bulls;
    else if (secret.find(guess[i])
             != std::string::npos)
        ++cows;
}
std::cout << bulls << cows << "\n";
```

**Python**

```
bulls = 0
cows = 0
for i in range(4):
    if secret[i] == guess[i]:
        bulls += 1
    elif secret.find(guess[i]) != -1:
        cows += 1
print(f'{bulls} {cows}')
```

Теперь соберём все фрагменты выше в одну программу. Заодно добавим туда текстовые сообщения, чтобы игроку было понятно, что именно происходит.

Также длина загадываемого числа определена в отдельной (в C++ даже константной) переменной. Так программу легко переделать, чтобы она была попроще и загадывала 3 цифры, либо совсем сложной, если использовать 6.

Сколько ходов вам нужно, чтобы отгадывать число?

**Листинг 6.3. bullcowHuman.cpp**

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

const int LEN = 4;

std::string generate()
{
    std::string res = "0123456789";
    for (int i = 0 ; i < LEN ; ++i)
    {
        int n = i + rand() % (10 - i);
        std::swap(res[i], res[n]);
    }
    res.resize(LEN);
    return res;
}

bool check(std::string s)
{
    for (int i = 0 ; i < LEN ; ++i)
    {
        if (s[i] < '0' || s[i] > '9'
            || s.find(s[i], i + 1) != std::string::npos)
        {
            return false;
        }
    }
    return true;
}

std::string inputGuess()
{
    std::string s;
    while (true)
    {
        std::cout << "Введите последовательность из 4 цифр: ";
        std::cin >> s;
        if (s.size() == LEN && check(s))
```

```
        {
            return s;
        }
    }
}

void countBullsAndCows(const std::string &secret, const std::string
↪  &guess)
{
    int bulls = 0;
    int cows = 0;
    for (int i = 0 ; i < LEN ; ++i)
    {
        if (guess[i] == secret[i])
        {
            ++bulls;
        }
        else if (secret.find(guess[i]) != std::string::npos)
        {
            ++cows;
        }
    }
    std::cout << "Быки: " << bulls << " Коровы: " << cows << "\n";
}

int main()
{
    srand((int)time(NULL));
    std::string secret = generate();
    std::string guess;
    while (true)
    {
        guess = inputGuess();
        if (guess == secret)
        {
            std::cout << "Вы выиграли!\n";
            break;
        }
        countBullsAndCows(secret, guess);
    }
}
```

**Листинг 6.4. bullcowHuman.py**

```
#!/usr/bin/python3
import random

LEN = 4

def generate():
    return ''.join(random.sample('0123456789', LEN))

def check(s):
    for i in range(LEN):
        if s.find(s[i], i + 1) != -1:
            return False
    return True

def inputGuess():
    while True:
        s = input('Введите последовательность из 4 цифр: ')
        if len(s) == LEN and s.isdigit() and check(s):
            return s

def countBullsAndCows(secret, guess):
    bulls = 0
    cows = 0
    for i in range(LEN):
        if secret[i] == guess[i]:
            bulls += 1
        elif secret.find(guess[i]) != -1:
            cows += 1
    print(f'Быки: {bulls} Коровы: {cows}')

secret = generate()
while True:
    guess = inputGuess()
    if secret == guess:
        print('Вы выиграли!')
        break
    countBullsAndCows(secret, guess)
```

### 6.2.8. Отгадывает компьютер

Разгадывать загадки компьютера интересно, но сложно. Поменяемся с программой ролями, пусть теперь она отгадывает.

Но возникает одна проблема — научить программу играть придётся именно нам. Начнём с общего алгоритма (рис. 6.2).

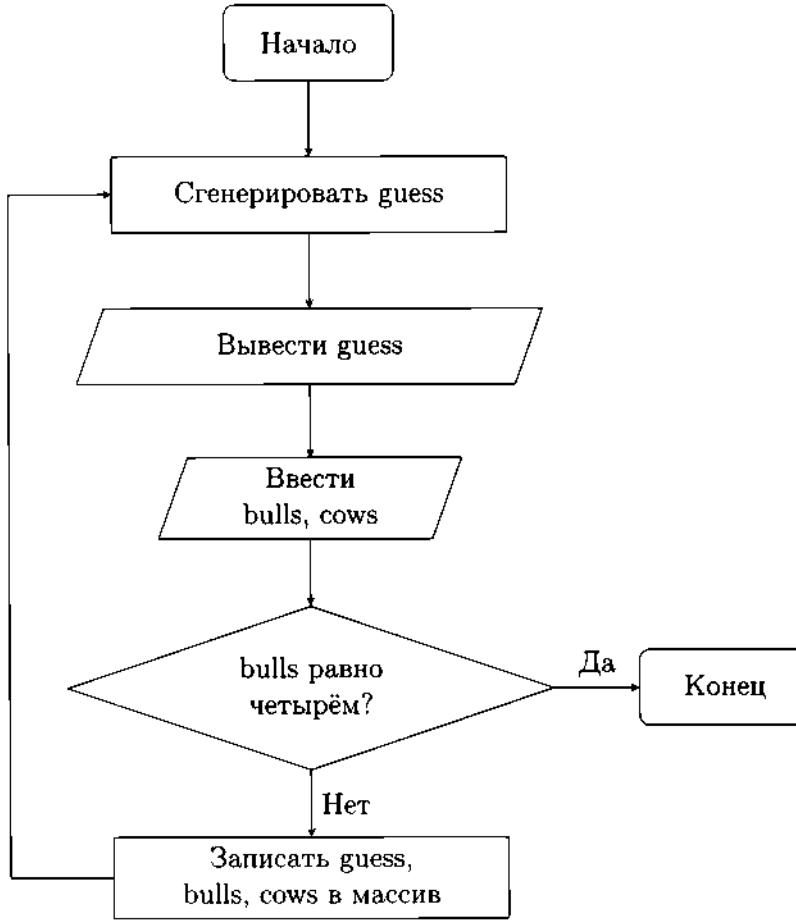



Рис. 6.2. Алгоритм игры «Быки и коровы», если отгадывает компьютер

### 6.2.9. Структура программы

Как и в прошлый раз, создадим несколько пустых функций. Правда, тут будет дополнительная сложность с тем, что вводиться должно не четырёхзначное число-отгадка, а два числа: количество «быков» и «коров».

C++

```
#include <string>
#include <iostream>

std::string generate()
{
  return "";
}

XXX readBullsCows()
{
}

void save(std::string guess, int bulls,
↪  int cows)
{
}
```

Python

```
def generate():
  return ""

def readBullsCows():
  return XXX

def save(guess, bulls, cows):
  pass
```

Также нам понадобится фрагмент кода, который эти (пока пустые) функции вызывает. В C++ этот фрагмент будет в функции `main`.

C++

```
int main()
{
  while (true)
  {
    std::string guess = generate();
    std::cout << guess << "\n";
    XXX = readBullsCows();
    if (bulls == 4)
    {
      break;
    }
    save(guess, bulls, cows);
  }
}
```

Python

```
while True:
  guess = generate()
  print(guess)
  XXX = readBullsCows()
  if b == 4:
    break
  save(guess, bulls, cows)
```

Наверное вы обратили внимание на новую функцию `save`, которой не было в прошлой программе. До сих пор всё было просто: получаем данные, анализируем, выводим реакцию. Сейчас реакция программы должна вычисляться на основе всех ранее полученных от игрока данных, ведь только по последней подсказке отгадать число не выйдет.

И тут возможны два подхода: хранить множество всех вариантов загаданного числа и вычёркивать оттуда неподходящие или хранить все прежние подсказки, на каждом ходу подбирая вариант, который им не противоречит.

На основе первого подхода можно реализовывать сложные алгоритмы поиска, которые минимизируют число попыток [3, 11]. Но мы находимся в главе «Полный перебор», поэтому алгоритм будет самый простой, использующий второй вариант. У этого варианта есть одно преимущество — так как с программой играет человек, то он может ошибаться. И было бы неплохо разрешить ему

исправлять ошибки. А сделать это можно, только если сохранилась история ходов.

### 6.2.10. Стуктуры данных

Вы не могли не заметить пропуски в предыдущем фрагменте. Это было предложение подумать, какой же тип использовать для хранения «быков» и «коров». Ведь на самом деле непонятно, как одновременно получить из функции два значения сразу: число «быков» и число «коров».

Как всегда, есть несколько вариантов. Первый — хранить эти числа в двух переменных. Но функция `readBullsCows` может возвращать только одно значение. Поэтому в C++ вместо возвращаемого значения удобнее передавать эти переменные по ссылке, чтобы функция могла их изменить, а в Python можно возвращать кортеж из двух значений. Отдельный тип структуры для хранения двух переменных мы не будем рассматривать, это избыточно для несложной программы.

Второй вариант — использовать одну переменную, но хранить в ней сразу два значения. Ведь и число быков, и число коров могут принимать значения от 0 до 4. В особо странном варианте игры — до 9. Значит, их можно скомбинировать в одно двузначное число `bulcow = bulls * 10 + cows`.

Использовать такой подход удобно, только если не требуется потом «быков» и «коров» разделять для обработки по отдельности. Попробуем для разнообразия этот способ, чтобы не загромождать программу.

Теперь замените в шаблоне выше две переменные `bulls` и `cows` на одну переменную `bulcow`. Как изменится условие выхода из цикла?

Какие ещё данные мы будем хранить? Помните функцию `save`, она должна запоминать подсказки пользователя. Поиск по ним не нужен, только последовательные обращения для проверки. Поэтому можно использовать списки, для них есть встроенная поддержка и в C++, и в Python.

В списке будут храниться пары значений: число-попытка и количество «быков» и «коров» в нём (мы же договорились хранить «быков» и «коров» в одной переменной).

C++

```
#include <list>
struct Bulcow {
    std::string guess;
    int bulcow;
};
std::list<Bulcow> guesses;
```

Python

```
guesses = []
```

Чтобы хранить два значения, пришлось объявить структуру с полями для них. Теперь у нас будет новый тип `Bulcow`, а переменные этого типа будут содержать сразу и строку и число.

Можно было бы использовать вместо собственного типа кортеж `std::tuple`. В нём тоже можно хранить несколько значений вместе. Но именованные поля структуры делают код более читабельным, чем безымянные поля кортежа.

На удивление короткий код получился. Секрет в динамической типизации Python. В теории можно будет добавлять в этот список любые значения, не только задуманные нами пары строка-число. И никакой компилятор нам не помешает сделать такую ошибку.

### 6.2.11. Узнаём у пользователя число «быков» и «коров»

Когда программа выводит свою попытку, пользователь, он же ведущий, должен ввести подсказку, то есть число «быков» и «коров». Мы уже решили, что возвращать функция будет одно число, в котором они оба закодированы.

Но запрашивать у пользователя можно как это одно число, так и «быков» и «коров» по отдельности. После этого нужно проверить корректность ввода и вернуть результат, если всё хорошо.

Если вводить одно число, то для проверки нужно будет отделить число коров от числа быков. А когда вводятся два числа, для формирования результата нужно будет из них сделать одно. Второй способ выглядит более наглядным и простым для кодирования, поэтому применим его.

**C++**

```
int readBullsCows()
{
  while (true)
  {
    int bulls, cows;
    std::cin >> bulls >> cows;
    if (bulls + cows >= 0
      && bulls + cows <= 4
      && bulls >=0 && cows >=0)
    {
      return bulls * 10 + cows;
    }
  }
}
```

**Python**

```
def readBullsCows():
  while True:
    bulls, cows = [int(x)
      for x in input().split()]
    if bulls + cows >= 0
      and bulls + cows <= 4
      and bulls >=0 and cows >=0:
      return bulls * 10 + cows
```

Основное отличие в двух фрагментах — в способе ввода чисел. В C++ ввести два числа очень легко с помощью оператора потокового ввода.

Код на Python начинается с магии. Мы хотим ввести числа в одной строке, поэтому вызов функции `input()` только один. Дальше с помощью функции `split()` эта строка делится на части, они перебираются с помощью оператора `for` и с помощью функции `int(x)` преобразуются в целочисленные значения.

В конце эти значения с помощью квадратных скобок помещаются в список. И оператор присваивания распределяет этот список между переменными `bulls` и `cows`.

Оставшаяся часть в обоих фрагментах одинаковая. Проверяется, что введены числа, удовлетворяющие правилам, а потом возвращается результат, если всё правильно.

Как убедиться, что всё правильно? В сумме число «быков» и «коров» не должно превышать 4, а каждое из этих чисел должно быть неотрицательным.

Так как мы решили, что в оставшейся программе быки и коровы будут храниться в одной переменной, нужно совместить их в одно значение. Быки — это десятки, коровы — единицы, поэтому получается простая формула `bulls * 10 + cows`.

Добавьте эту функцию в программу и попробуйте вводить разные варианты «быков» и «коров», правильные и неправильные. Есть ли ввод, при котором программа завершится?

### 6.2.12. Функция generate — поиск возможных ответов

Мы уже генерировали случайное четырёхзначное число с разными цифрами. В этот раз нам тоже надо это делать, но немного иначе.

Наша задача — найти такую попытку, которая не противоречит полученным ранее подсказкам. Если она попадёт в цель, то программа завершится. В противном случае мы получим дополнительную подсказку, которая исключит ещё несколько вариантов.

Будем перебирать все числа от 100 до 9999[1]. Проверка корректности комбинации цифр состоит из двух частей. Сначала нужно убедиться, что цифры в ней не повторяются, ведь мы же перебираем все числа подряд. После этого нужно проверить, не противоречит ли эта комбинация ранее полученным подсказкам.

[1] Можно было бы сократить этот диапазон до 123–9876, но полученное небольшое ускорение работы мы обменяем на более непонятный код. В настоящих программах в таких случаях пишут подробный комментарий, чтобы другие разработчики могли понять, что к чему.

Проверку уникальности цифр мы уже делали в прошлой программе, возьмём функцию **check** прямо оттуда. А вот второй проверки там не было. Как вообще определить, что наш новый вариант противоречит уже известной подсказке? Раз мы пытаемся отгадать секретное число, то выбранный в текущей итерации вариант вполне может им оказаться. Если это действительно будет секретное число, то мы можем вычислить, что бы ответил ведущий для любой из ранее сохранённых попыток.

Значит, мы можем сравнить то, что на самом деле сказал ведущий (ведь мы сохраняем все попытки), с тем, что он бы сказал, если бы сгенерированное нами число совпадало с задуманным. Если все ответы сходятся, то сгенерированное число может быть на самом деле задумано, поэтому его и надо вывести в качестве следующей попытки.

C++

```
for (int v = 100 ; v <= 9999 ; ++v)
{
  std::string s = std::to_string(v);
  if (v < 1000)
  {
    s = "0" + s;
  }
  if (!check(s))
  {
    continue;
  }
  bool ok = true;
  for (const Bulcow &g : guesses)
  {
    if (countBullsCows(s, g.guess)
        != g.bulcow)
    {
      ok = false;
    }
  }
  if (ok)
  {
    return s;
  }
}
```

Python

```
for v in range(100, 10000):
  s = f'{v:04d}'
  if not check(s):
    continue
  ok = True
  for g in guesses:
    if countBullsCows(s,
      g['guess']) != g['bulcow']:
      ok = False
  if ok:
    return s
return "9999"
```

### 6.2.13. Функция save — сохранение полученных ответов

Код, который перебирает сохранённые ранее подсказки, чтобы проверить, подходит ли под них новый вариант числа, мы уже написали. Теперь пришло время функции, которая дополняет этот список после получения очередной подсказки от ведущего.

**C++**

```
void save(std::string guess,
          int bulcow)
{
  guesses.push_back({guess, bulcow});
}
```

В списке **guesses** хранятся значения типа **Bulcow**, каждое из которых, в свою очередь, состоит из строки и числа. Функция **push_back** добавляет один такой сложный элемент в список. Фигурные скобки обозначают создание **Bulcow** из переменных **guess** и **bullcow**.

**Python**

```
def save(guess, bulcow):
  guesses.append({'guess': guess,
                  'bulcow': bulcow})
```

Списки в Python не ограничивают тип значений, которые будут в них записываться. За этим следит только программист. В нашей программе мы будем хранить там элементы типа «словарь», каждый из которых содержит по два значения: **guess** и **bulcow**.

Можно было бы просто использовать кортеж из двух безымянных значений, чтобы немного упростить код. Для больших же программ лучшим решением было бы создание собственного класса со свойствами **guess** и **bulcow**.

### 6.2.14. Функция countBullsCows — подсчёт «быков» и «коров»

Последняя функция, которую осталось написать, это **countBullsCows**. Её не было в исходной декомпозиции алгоритма, но при разработке функции **generate** показалось удобным выделить её, чтобы не перегружать тот код.

**C++**

```
int countBullsCows(std::string secret,
    std::string guess)
{
  int bulls = 0;
  int cows = 0;
  for (int i = 0 ; i < 4 ; ++i)
  {
    if (guess[i] == secret[i])
    {
      ++bulls;
    }
    else if (secret.find(guess[i])
      != std::string::npos)
    {
      ++cows;
    }
  }
  return bulls * 10 + cows;
}
```

**Python**

```
def countBullsCows(secret, guess):
  bulls = 0
  cows = 0
  for i in range(4):
    if secret[i] == guess[i]:
      bulls += 1
    elif secret.find(guess[i])
      != -1:
      cows += 1
  return bulls * 10 + cows
```

Последние несколько функций по отдельности не давали видимого эффекта. Но теперь можно наконец-то соединить всё вместе.

В C++-варианте функции `check` можно удалить ненужные проверки на то, что строка состоит из цифр.

И не забудьте протестировать игру чем-то вроде этого:

**Экран 2. Компьютер отгадывает число**

```
Моя попытка: 0123
Введите число быков и число коров: 1 0
Моя попытка: 0456
Введите число быков и число коров: 0 1
Моя попытка: 4178
Введите число быков и число коров: 1 0
Моя попытка: 5928
Введите число быков и число коров: 0 2
Моя попытка: 9573
Введите число быков и число коров: 4 0
Я отгадал!
```

### 6.2.15. Задания для самостоятельной работы

**1.** Доработайте первую программу, чтобы на первом месте загаданной комбинации не могло быть нуля, чтобы она была похожа на число, а не просто набор цифр.

**2.** Измените вторую программу, чтобы она не могла зайти в тупик. Такое бывает, если пользователь ошибается во вводе и указывает неверное число «быков» и «коров». В этой ситуации можно пройтись по прежним попыткам (они ведь сохранены), и уточнить у пользователя, верно ли он ввёл подсказки для них.

**3.** Объедините две программы, чтобы компьютер играл сам с собой. Сколько в среднем ему нужно попыток, чтобы отгадать число?

**Листинг 6.5. bullcowComputer.cpp**

```
#include <string>
#include <iostream>
#include <list>

struct Bulcow {
    std::string guess;
    int bulcow;
};
std::list<Bulcow> guesses;

int countBullsCows(std::string secret, std::string guess)
{
    int bulls = 0;
    int cows = 0;
    for (int i = 0 ; i < 4 ; ++i)
    {
        if (guess[i] == secret[i])
        {
            ++bulls;
        }
        else if (secret.find(guess[i]) != std::string::npos)
        {
            ++cows;
        }
    }
    return bulls * 10 + cows;
}

bool check(std::string s)
{
    for (int i = 0 ; i < 4 ; ++i)
    {
        if (s.find(s[i], i + 1) != std::string::npos)
        {
            return false;
        }
    }
    return true;
}
```

```
std::string generate()
{
    for (int v = 100 ; v <= 9999 ; ++v)
    {
        std::string s = std::to_string(v);
        if (v < 1000)
        {
            s = "0" + s;
        }
        if (!check(s))
        {
            continue;
        }
        bool ok = true;
        for (const Bulcow &g : guesses)
        {
            if (countBullsCows(s, g.guess) != g.bulcow)
            {
                ok = false;
            }
        }
        if (ok)
        {
            return s;
        }
    }
    return "9999";
}

int readBullsCows()
{
    while (true)
    {
        int bulls, cows;
        std::cout << "Введите число быков и число коров: ";
        std::cin >> bulls >> cows;
        if (bulls + cows >= 0
            && bulls + cows <= 4
            && bulls >=0 && cows >=0)
        {
            return bulls * 10 + cows;
```

```
        }
    }
}

void save(std::string guess, int bulcow)
{
    guesses.push_back({guess, bulcow});
}

int main()
{
    while (true)
    {
        std::string guess = generate();
        std::cout << "Моя попытка: " << guess << "\n";
        int bulcow = readBullsCows();
        if (bulcow == 40)
        {
            std::cout << "Я отгадал!\n";
            break;
        }
        save(guess, bulcow);
    }
}
```

**Листинг 6.6. bullcowComputer.py**

```
#!/usr/bin/python3
import sys

guesses = []

def countBullsCows(secret, guess):
    bulls = 0
    cows = 0
    for i in range(4):
        if secret[i] == guess[i]:
            bulls += 1
        elif secret.find(guess[i]) != -1:
            cows += 1
    return bulls * 10 + cows
```

```
def generate():
    for v in range(100, 10000):
        s = f'{v:04d}'
        ok = True
        for i in range(4):
            if s.find(s[i], i + 1) != -1:
                ok = False
        for g in guesses:
            if countBullsCows(s, g['guess']) \
                != g['bulcow']:
                ok = False
        if ok:
            return s
    return "9999"

def readBullsCows():
    while True:
        bulls, cows = [int(x)
            for x in input('Введите число быков и число коров:
            ↪ ').split()]
        if bulls + cows >= 0 \
            and bulls + cows <= 4 \
            and bulls >=0 and cows >=0:
            return bulls * 10 + cows

def save(guess, bulcow):
    guesses.append({'guess': guess, 'bulcow': bulcow})

while True:
    guess = generate()
    print(f'Моя попытка: {guess}')
    bulcow = readBullsCows()
    if bulcow == 40:
        print('Я отгадал!')
        break
    save(guess, bulcow)
```

# Глава 7

# Эвристические алгоритмы

## 7.1. Игра «Морской бой»

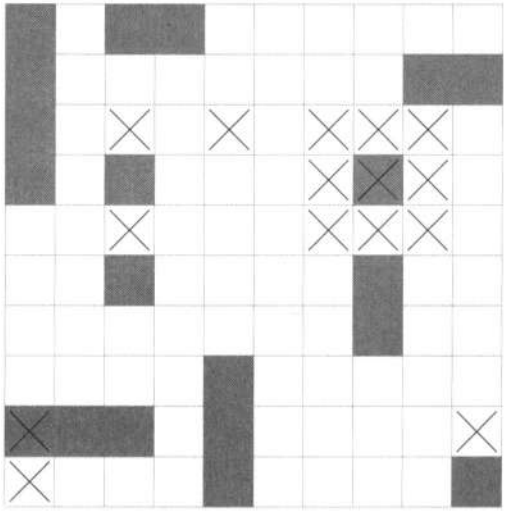
Игру «Морской бой» знают все. На полях 10 × 10 клеток два игрока размещают 10 кораблей втайне друг от друга. А затем пытаются потопить корабли соперника, поочерёдно делая «выстрелы» в одну из клеток вражеского поля.

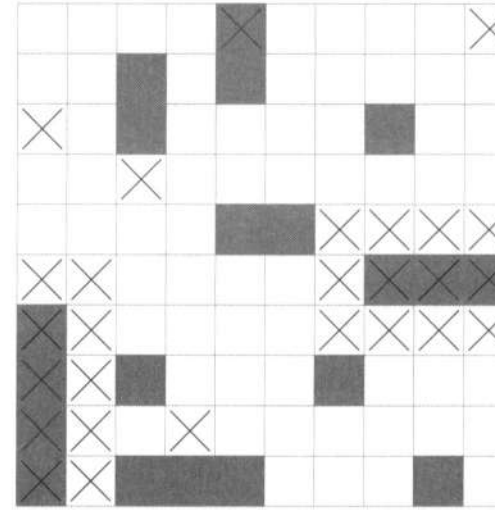
Корабли обычно очень незамысловатые: один четырёхпалубный линкор (размером 1 × 4 клетки поля), два трёхпалубных крейсера (1 × 3 клетки), три двухпалубных эсминца (1 × 2) и четыре однопалубных катера (1 × 1).

Корабли не должны касаться друг друга ни сторонами своих клеток, ни углами. Каждый ход состоит из «выстрела» — указания координат выбранной цели. Если было попадание, соперник отвечает «ранен» или «убит», в зависимости от того, остался ли поражённый корабль на плаву. В этом случае стрелявший получает право на повторный ход. Иначе соперник отвечает «мимо» и теперь уже сам получает право стрелять.

Побеждает тот, кто первым потопил все вражеские корабли. Конечно же, в этой игре очень многое зависит от удачи. Но и элементы стратегии в планировании ходов тоже есть, поэтому попробуем написать программу, с которой было бы интересно играть.

Стратегия нашей программы будет основываться на эвристиках, то есть на таких алгоритмах, которые выглядят логичными, но не обязательно дают оптимальный результат. Например, мы можем предположить, что люди часто ставят корабли по углам, и стараться сначала поразить именно эти клетки. Но это всё потом, а сейчас займёмся основой программы, чтобы затем перейти к стратегиям, эвристикам и искусственному интеллекту.





**Экран 1. Фрагмент партии в «Морской бой»**

```
Ваше поле:
   1 2 3 4 5 6 7 8 9 10
 1 0 0 0 0 . . . . x 0
 2 . x . x . . . . . .
 3 . . 0 . . . . 0 0 .
 4 . . 0 . . . . . . x
 5 . . 0 . 0 0 * . . .
 6 . . . . x . . . . .
 7 x . 0 . . . 0 0 . .
 8 . . . . . . . . . .
 9 . . 0 . . . . . 0 .
10 0 . . x . . . . 0 .
Мой ход: 3, 7
Убит!
Ваше поле:
   1 2 3 4 5 6 7 8 9 10
 1 0 0 0 0 . . . . x 0
 2 . x . x . . . . . .
 3 . . 0 . . . . 0 0 .
 4 . . 0 . . . . . . x
 5 . . 0 . 0 0 * . . .
 6 . x x x x . . . . .
 7 x x * x . . 0 0 . .
 8 . x x x . . . . . .
 9 . . 0 . . . . . 0 .
10 0 . . x . . . . 0 .
Мой ход: 7, 7
Ранен!
Ваше поле:
   1 2 3 4 5 6 7 8 9 10
 1 0 0 0 0 . . . . x 0
 2 . x . x . . . . . .
 3 . . 0 . . . . 0 0 .
 4 . . 0 . . . . . . x
 5 . . 0 . 0 0 * . . .
 6 . x x x x . . . . .
 7 x x * x . . * 0 . .
 8 . x x x . . . . . .
 9 . . 0 . . . . . 0 .
10 0 . . x . . . . 0 .
Мой ход: 10, 9
Ваше поле:
   1 2 3 4 5 6 7 8 9 10
 1 0 0 0 0 . . . . x 0
 2 . x . x . . . . . .
 3 . . 0 . . . . 0 0 .
 4 . . 0 . . . . . . x
 5 . . 0 . 0 0 * . . .
 6 . x x x x . . . . .
 7 x x * x . . * 0 . .
 8 . x x x . . . . . .
 9 . . 0 . . . . . 0 x
10 0 . . x . . . . 0 .
Моё поле:
   1 2 3 4 5 6 7 8 9 10
 1 . . . . . . . . . x
 2 . . . . . . . . . .
 3 . x x x x x . . . .
 4 . x * * * x . . . .
 5 . x x x x x . . . .
 6 . . . . . . . . . .
 7 . . . . . x . . . .
 8 . . . . . x . . .
 9 . . . . . . . . . .
10 x . . . . . . . . x
Куда будете стрелять (x, y)? 9
↪ 9
Ранен!
Моё поле:
   1 2 3 4 5 6 7 8 9 10
 1 . . . . . . . . . x
 2 . . . . . . . . . .
 3 . x x x x x . . . .
 4 . x * * * x . . . .
 5 . x x x x x . . . .
 6 . . . . . . . . . .
 7 . . . . . x . . . .
 8 . . . . . x . . .
 9 . . . . . . . . * .
10 x . . . . . . . . x
```

### 7.1.1. Игровая позиция

Какая игра без вывода поля на экран? Чтобы нарисовать картинку вроде той, что приведена выше, потребуется знать состояние каждой клетки поля.

Клетка может быть занята кораблём, а может быть не занята. Ещё состояние клетки меняется во время игры. По ней может быть не сделано выстрела, либо она уже может быть поражена. Поэтому (из-за комбинации этих двух признаков) у каждой клетки может быть всего 4 состояния.

Список кораблей с их координатами можно хранить в списке, массиве или чём-то ещё. Но, чтобы проверять попадания, придётся среди этих кораблей искать тот, который включает выбранные координаты. Это не очень удобно. Поэтому лучше сразу записывать положения кораблей в двумерном массиве, как это и делается при игре в реальности. Только там вместо массива используется листок в клеточку.

C++
```
const int N = 10;
int field[N][N] = {};
```

Python
```
N = 10
field = [[0] * N for _ in range(N)]
```

Получается квадратный двумерный массив из целых чисел. Вообще можно было бы сделать массив и из булевых переменных, но разные числа (а не только «ложь» или «истина») нам ещё наверняка пригодятся позднее.

Также при игре на бумаге после выстрела по какой-то клетке её вычёркивают. Это же надо нарисовать и на экране. А значит, нам будет удобно завести массив, где для клеток будет храниться признак «поражённости», то есть стреляли уже туда или нет.

C++
```
bool hits[N][N] = {};
```

Python
```
hits = [[False] * N for _ in range(N)]
```

Мы сделали два массива для одного поля. Но полей-то два, у игрока и у компьютера. Выходит, что массивов должно быть четыре. Чтобы упростить работу с ними (и передачу в разные функции в качестве параметров), объединим их в структуру:

C++
```
struct Player
{
  int field[N][N] = {};
  bool hits[N][N] = {};
};
```

Python
```
class Player:
  def __init__(self):
    self.field = [[0] * N for _ in
    ↪ range(N)]
    self.hits = [[False] * N for _ in
    ↪ range(N)]
```

Теперь будет легко написать функцию, которая выводит поле на экран. Живой корабль будем обозначать буквой `'0'`, подбитый корабль — символом `'*'`, а поражённую и непоражённую клетки — символами `'x'` и `'.'` соответственно.

C++

```
void printField(const Player &pl)
{
  for (int i = 0 ; i < N ; ++i)
  {
    for (int j = 0 ; j < N ; ++j)
    {
      std::cout << ' ';
      if (pl.hits[i][j])
      {
        if (pl.field[i][j])
        {
          std::cout << '*';
        }
        else
        {
          std::cout << 'x';
        }
      }
      else if (pl.field[i][j])
      {
        std::cout << '0';
      }
      else
      {
        std::cout << '.';
      }
    }
    std::cout << '\n';
  }
  std::cout << '\n';
}
```

Python

```
def printField(pl):
  for i in range(N):
    for j in range(N):
      print(' ', end='')
      if pl.hits[i][j]:
        if pl.field[i][j] > 0:
          print('*', end='')
        else:
          print('x', end='')
      elif pl.field[i][j] > 0:
          print('0', end='')
      else:
          print('.', end='')
    print()
  print()
```

## 7.1.2. Структура программы

Укрупнённая структура программы мало отличается от всех остальных пошаговых игр. Сначала инициализируем все структуры данных (для «Морского боя» в инициализацию также входит расстановка кораблей обоими игроками).

C++

```
#include <time.h>
#include <stdlib.h>
#include <iostream>

int main()
{
  // случайные числа наверняка
  ↪ понадобятся
  srand((int)time(NULL));
  // игровая позиция
  Player player, computer;
  // расставляем корабли
  ...
}
```

Python

```
import random

player = Player()
computer = Player()
# расставляем корабли
...
```

После этого начинается главный игровой цикл, в котором мы даём возможность игрокам ходить по очереди, пока один из них не победит.

C++

```
while (true)
{
  // ход компьютера
  printField(player);
  ...
  // ход человека
  printField(computer);
  ...
}
```

Python

```
while True:
  # ход компьютера
  printField(player)
  ...
  # ход человека
  printField(computer)
  ...
```

Мы выводим каждое из игровых полей непосредственно перед ходами, чтобы было удобно выбирать координаты цели (или наблюдать за тем, как компьютер их выбирает).

### 7.1.3. Стрельба по кораблям

Расстановка кораблей — непростое дело. Надо много всего проверять, чтобы не нарушить правила. Гораздо проще сделать функцию для стрельбы по кораблям. Ведь что для этого нужно? Всего лишь записать «истину» в одну ячейку в массиве `hits`.

C++

```
void shoot(Player &pl, int x, int y)
{
  if (!pl.hits[y][x])
  {
    pl.hits[y][x] = true;
  }
}
```

Python

```
def shoot(pl, x, y):
  if not pl.hits[y][x]:
    pl.hits[y][x] = True
```

Теперь можно добавить вызов этой функции в главный игровой цикл и расстреливать корабли противника:

C++

```
// ход человека
printField(computer);
do
{
  std::cout << "Куда будете стрелять (x,
  ↪  y)? ";
  std::cin >> x >> y;
  --x;
  --y;
}
while (x < 0 || y < 0 || x >= N || y >= N
↪  || computer.hits[y][x]);
shoot(computer, x, y);
```

Python

```
# ход человека
printField(computer)
while True:
  str = input('Куда будете стрелять (x,
  ↪  y)? ')
  x, y = str.split()
  x = int(x) - 1
  y = int(y) - 1
  if x >= 0 and y >= 0 and x < N and y <
  ↪  N and not computer.hits[y][x]:
    break
shoot(computer, x, y)
```

Уже можно играть! Стреляйте по клеткам и наблюдайте, как меняется поле компьютера. Можете даже сначала записать в массив `field` какие-нибудь ненулевые значения, чтобы на поле были и «корабли».

Заметили, что выбирать клетку неудобно? Чтобы понять, где какой столбец и где какая строка, нужно пересчитывать их. Давайте добавим нумерацию в функцию вывода игрового поля:

C++

```
#include <iomanip>

void printField(const Player &pl)
{
  std::cout << "  ";
  for (int j = 0 ; j < N ; ++j)
  {
    std::cout << ' ' << j + 1;
  }
  std::cout << '\n';
  for (int i = 0 ; i < N ; ++i)
  {
    std::cout << std::setw(2) << i + 1;
    ...
  }
  std::cout << '\n';
}
```

Python

```
def printField(pl):
  print('  ', end='')
  for j in range(N):
    print(f' {j + 1}', end='')
  print()
  for i in range(N):
    print(f'{i + 1:>2}', end='')
    ...
  print()
```

Модификатор `std::setw(2)` дополняет пробелами до 2 символов следующее выводимое значение. Благодаря ему все номера строк (и 1, и 10) занимают по два символа, и поле выглядит ровным.

Символы `:>2` рядом с выводимым значением означают «выравнивание по правому краю и дополнение пробелами до двух символов».

Но, чтобы было куда стрелять, сначала нужно расставить корабли. Для этого игрок должен ввести позиции своих кораблей, а компьютер как-то расположить свои.

### 7.1.4. Расстановка кораблей

Расположить корабль на игровом поле не так-то просто. Нужно знать его размер, проследить, что он не выходит за границы поля, не пересекается с другими кораблями и даже не касается их.

Даже перечисление этих условий выглядит громоздко. Разумно это всё оформить в виде функции, а не реализовывать каждый раз. Тем более что корабли будут выставлять оба соперника.

Функция будет возвращать `true`, если корабль помещается в нужном месте.

C++
```
bool checkShip(const Player &pl, int x,
↪  int y, int len, bool v)
{
  // все проверки
  return true;
}
```

Python
```
def checkShip(pl, x, y, len, v):
  # все проверки
  return True
```

Сначала нужно проверить, не выходит ли корабль за границы поля. Это может произойти, если его левая верхняя клетка либо правая нижняя находится за пределами поля. Средние клетки проверять отдельно не нужно, потому что высунуться за границы они не смогут, если края помещаются.

Левую верхнюю клетку корабля найти легко — она имеет координаты $(x, y)$. А как найти правую нижнюю? Корабль — это прямоугольник размером $1 \times len$, расположенный горизонтально или вертикально. Это описывается параметром $v$, который равен `true`, если корабль нужно поставить вертикально. В таком случае правая нижняя его клетка будет иметь координаты $(x, y + len - 1)$. Иначе он расположен горизонтально, а правая нижняя клетка будет в $(x + len - 1, y)$.

Теперь мы знаем, как проверить, что корабль не вылез за пределы поля. Вот условия для функции `checkShip`:

C++
```
if (x < 0 || y < 0)
{
  return false;
}
int endx = v ? x : x + len - 1;
int endy = v ? y + len - 1 : y;
if (endx >= N || endy >= N)
{
  return false;
}
```

Python
```
if x < 0 or y < 0:
  return False

endx = x if v else x + len - 1
endy = y + len - 1 if v else y
if endx >= N or endy >= N:
  return False
```

Проверять, что левая верхняя клетка лежит за правой границей поля, нет смысла, потому что тогда и правая нижняя клетка окажется там же. То же самое верно и для второго условия.

Теперь нужно узнать, не пересекается ли корабль с другими кораблями. Координаты краёв у нас уже есть, нужно просто перебрать все клетки между ними:

C++
```
for (int i = 0 ; i < len ; ++i)
{
  int xx = v ? x : x + i;
  int yy = v ? y + i : y;
  if (pl.field[yy][xx])
  {
    return false;
  }
}
```

Python
```
for i in range(0, len):
  xx = x if v else x + i
  yy = y + i if v else y
  if pl.field[yy][xx] > 0:
    return False
```

Что ещё осталось? Проверить, что корабли не касаются друг друга. Но как определить касание? Например, если соседние с кораблём клетки попадают на другой корабль. То, что клетка попадает на другой корабль, мы только что проверяли. Осталось прогнать через эту проверку все соседние клетки, а не только клетки корабля.

Для этого можно расширить проверяемый прямоугольник. Чтобы он был размером не $1 \times len$, а $3 \times (len + 2)$. Только координаты левого верхнего угла этого прямоугольника будут $(x - 1, y - 1)$.

C++

```
for (int i = -1 ; i <= len ; ++i)
{
  for (int j = -1 ; j <= 1 ; ++j)
  {
    int xx = v ? x + j : x + i;
    int yy = v ? y + i : y + j;
    if (xx >= 0 && xx < N && yy >= 0 &&
    ↪  yy < N)
    {
      if (pl.field[yy][xx])
      {
        return false;
      }
    }
  }
}
```

Python

```
for i in range(-1, len + 1):
  for j in range(-1, 2):
    xx = x + j if v else x + i
    yy = y + i if v else y + j
    if xx >= 0 and xx < N and yy >= 0 and
    ↪  yy < N:
      if pl.field[yy][xx] > 0:
        return False
```

Клетки прямоугольника одним циклом не обойти, поэтому теперь циклов два. И ещё добавилась проверка попадания координат клетки в границы поля, потому что корабль может стоять с краю, а в этом случае у него некоторых клеток по периметру просто нет.

### 7.1.5. Расстановка кораблей: заполнение клеток

Когда мы убедились, что место для корабля есть, можно его размещать. Для этого нужно будет заполнить ячейки массива `Player.field` ненулевыми значениями. В этой функции уже проверять ничего не нужно, просто обойти клетки, из которых состоит корабль.

C++

```
void placeShip(Player &pl, int x, int y,
↪  int len, bool v)
{
  for (int i = 0 ; i < len ; ++i)
  {
    int xx = v ? x : x + i;
    int yy = v ? y + i : y;
    pl.field[yy][xx] = 1;
  }
}
```

Python

```
def placeShip(pl, x, y, type, v):
  len = shipLen[type]
  for i in range(len):
    xx = x if v else x + i
    yy = y + i if v else y
    pl.field[yy][xx] = type + 1
```

Теперь можно разместить на поле вражеские корабли и расстреливать их.

C++
```
placeShip(computer, 3, 2, 4, true);
placeShip(computer, 7, 7, 1, false);
```

Python
```
placeShip(computer, 3, 2, 4, True)
placeShip(computer, 7, 7, 1, False)
```

Протестируйте, как работает вывод поля и стрельба, когда корабли есть на поле.

Но при стрельбе чего-то не хватает. Не выдаётся никаких сообщений. Попали мы или нет, ещё можно определить по состоянию игрового поля. А вот как понять, потоплен ли корабль? В «живой» партии соперник отвечает нам «Убит» или «Ранен», в зависимости от результатов попадания. Придётся доделать функцию стрельбы, чтобы программа делала так же.

### 7.1.6. Доработка функции стрельбы

Добавить вывод сообщения о попадании очень легко:

C++
```
void shoot(Player &pl, int x, int y)
{
  if (!pl.hits[y][x])
  {
    pl.hits[y][x] = true;
    if (pl.field[y][x])
    {
      std::cout << "Ранен!\n";
    }
    else
    {
      std::cout << "Мимо!\n";
    }
  }
}
```

Python
```
def shoot(pl, x, y):
  if not pl.hits[y][x]:
    pl.hits[y][x] = True
    if pl.field[y][x] > 0:
      print('Ранен!')
    else:
      print('Мимо!')
```

Тут всегда выводится «Ранен». А как понять, потоплен ли корабль? Можно проверять все окрестности подбитой клетки, чтобы убедиться, что она не связана с неподбитыми. Но такой код будет довольно сложным.

Лучше добавить в игровую позицию немного данных о кораблях — сколько у каждого из них осталось непотопленных клеток.

C++
```
// число кораблей у одного игрока
const int S = 10;

struct Player
{
    ...
    int ships[S] = {};
};
```

Python
```
# число кораблей у одного игрока
S = 10

class Player:
  def __init__(self):
    ...
    self.ships = [0] * S
```

Но как-то надо ещё определить, к какому кораблю относится подбитая клетка. Вот тут и пригодится возможность записывать любое целое число в описание каждой клетки. Ноль будет значить, что клетка пустая (как и было), а ненулевое значение — это номер корабля (от 1 до $S$).

Теперь при попадании можно просто уменьшать соответствующую ячейку массива `ships` на единицу. И если там остался ноль, корабль потоплен.

C++

```
void shoot(Player &pl, int x, int y)
{
  if (!pl.hits[y][x])
  {
    pl.hits[y][x] = true;
    if (pl.field[y][x])
    {
      if (!--pl.ships[pl.field[y][x] -
      ↪ 1])
      {
        std::cout << "Убит!\n";
      }
      else
      {
        std::cout << "Ранен!\n";
      }
    }
    else
    {
      std::cout << "Мимо!\n";
    }
  }
}
```

Python

```
def shoot(pl, x, y):
  if not pl.hits[y][x]:
    pl.hits[y][x] = True
    if pl.field[y][x] > 0:
      pl.ships[pl.field[y][x] - 1] -= 1
      if pl.ships[pl.field[y][x] - 1] ==
      ↪ 0:
        print('Убит!')
      else:
        print('Ранен!')
    else:
      print('Мимо!')
```

Но ведь функция размещения корабля записывает в `Player.field` только единицы. Придётся доработать и её. Теперь вместо длины будем передавать в функцию номер корабля во флотилии, а длину получим из заранее заполненного массива.

C++

```
const int shipLen[S] = {4, 3, 3, 2, 2, 2,
↪ 1, 1, 1, 1};

void placeShip(Player &pl, int x, int y,
↪ int type, bool v)
{
  int len = shipLen[type];
  for (int i = 0 ; i < len ; ++i)
  {
    int xx = v ? x : x + i;
    int yy = v ? y + i : y;
    pl.field[yy][xx] = type + 1;
  }
  pl.ships[type] = len;
}
```

Python

```
shipLen = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]

def placeShip(pl, x, y, type, v):
  len = shipLen[type]
  for i in range(len):
    xx = x if v else x + i
    yy = y + i if v else y
    pl.field[yy][xx] = type + 1
  pl.ships[type] = len
```

С последними доработками стрелять по кораблям стало намного интереснее. Не забудьте протестировать новый вариант.

### 7.1.7. Расстановка кораблей игрока

Теперь всё готово к тому, чтобы игрок мог расставить свои корабли. Самое главное, что есть функция проверки корректности положения, так что ошибиться у человека не выйдет.

Если заставлять игрока подтверждать каждый выстрел компьютера (попал он или нет), игра интереснее не станет, а будет очень утомительной. К тому же тогда человеку понадобятся карандаш и бумага. Так что пользователям нашей программы придётся поверить, что она не жульничает, когда они вводят позиции своих кораблей.

Как обычно, программа будет запрашивать координаты поочерёдно для каждого корабля. И ещё для кораблей, состоящих больше чем из одной клетки, придётся ввести ориентацию — вертикальная или горизонтальная.

C++

```
for (int s = 0 ; s < S ; ++s)
{
  printField(player);
  int x, y;
  char c;
  bool v;
  do
  {
    std::cout << "Введите позицию для
    ↪  корабля " << s + 1
      << " (" << shipLen[s] <<
      ↪  "-палубный) (x, y"
      << (shipLen[s] > 1 ? ", v" : "")
      << "): ";
    std::cin >> x >> y;
    if (shipLen[s] > 1)
    {
      std::cin >> c;
    }
    --x;
    --y;
    v = c == 'v' || c == 'V';
  }
  while (!checkShip(player, x, y,
  ↪  shipLen[s], v));
  placeShip(player, x, y, s, v);
}
```

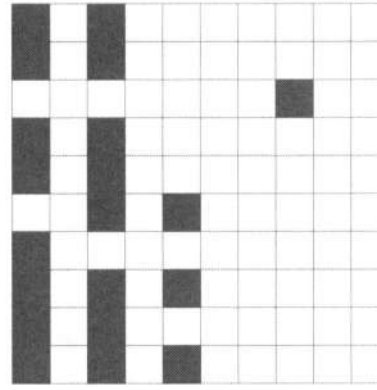
Python

```
for s in range(S):
  printField(player)
  while True:
    str = input(f'Введите позицию для
    ↪  корабля {s + 1}
    ↪  ({shipLen[s]}-палубный) (x, y{",
    ↪  v" if shipLen[s] > 1 else ""}):
    ↪  ')
    if shipLen[s] > 1:
      x, y, c = str.split()
    else:
      x, y = str.split()
    x = int(x) - 1
    y = int(y) - 1
    v = c == 'v' or c == 'V'
    if checkShip(player, x, y,
    ↪  shipLen[s], v):
      break
  placeShip(player, x, y, s, v)
```

### 7.1.8. Расстановка кораблей компьютером

Чтобы с компьютером было интересно играть, не будем использовать плотную расстановку основной части кораблей, как на рисунке ниже, чтобы противник

долго-долго искал оставшиеся однопалубные корабли на почти пустом поле (подробнее об этом можно почитать в [3]). Лучше просто разместим корабли случайно.



Подобно тому как это уже было в других играх, будем генерировать для каждого корабля случайные координаты (и не забудем про ориентацию), которые, правда придётся ещё проверить на корректность с помощью написанной ранее функции `checkShip`.

C++

```
for (int s = 0 ; s < S ; ++s)
{
  int x, y;
  bool v;
  do
  {
    x = rand() % N;
    y = rand() % N;
    v = rand() % 2;
  }
  while (!checkShip(computer, x, y,
  ↪  shipLen[s], v));
  placeShip(computer, x, y, s, v);
}
```

Python

```
for s in range(S):
  while True:
    x = random.randint(0, N - 1)
    y = random.randint(0, N - 1)
    v = random.choice([True, False])
    if checkShip(computer, x, y,
    ↪  shipLen[s], v):
      break
  placeShip(computer, x, y, s, v)
```

Код получился короче, чем для игрока, потому что не надо выводить никаких сообщений.

### 7.1.9. Снова ход игрока

По правилам, если в корабль попали, стреляющий имеет право на повторный ход. А в нашей программе такого ещё нет. Но как в игровом цикле узнать, было ли попадание, чтобы игрок мог выстрелить снова, когда за стрельбу у нас отвечает функция `shoot`?

Конечно же, придётся заставить её возвращать признак того, было ли попадание.

C++

```
bool shoot(Player &pl, int x, int y)
{
  if (!pl.hits[y][x])
  {
    pl.hits[y][x] = true;
    if (pl.field[y][x])
    {
      ...
      return true;
    }
    ...
  }
  return false;
}
```

Python

```
def shoot(pl, x, y):
  if not pl.hits[y][x]:
    pl.hits[y][x] = True
    if pl.field[y][x] > 0:
      ...
      return True
    else:
      print('Мимо!')
  return False
```

И с учётом этого возвращаемого значения можно доработать цикл, отвечающий за один ход игрока. Теперь за этот ход игрок сможет выстрелить несколько раз, если будет всё время попадать в цель.

C++

```
while (true)
{
  std::cout << "Моё поле:\n";
  printField(computer);
  do
  {
    std::cout << "Куда будете стрелять
    ↪  (x, y)? ";
    std::cin >> x >> y;
    --x;
    --y;
  }
  while (x < 0 || y < 0 || x >= N || y >=
  ↪  N || computer.hits[y][x]);
  if (!shoot(computer, x, y))
  {
    std::cout << "Моё поле:\n";
    printField(computer);
    break;
  }
}
```

Python

```
while True:
  print('Моё поле:')
  printField(computer)
  while True:
    str = input('Куда будете стрелять (x,
    ↪  y)? ')
    x, y = str.split()
    x = int(x) - 1
    y = int(y) - 1
    if x >= 0 and y >= 0 and x < N and y
    ↪  < N and not computer.hits[y][x]:
      break
  if shoot(computer, x, y) == Missed:
    print('Моё поле:')
    printField(computer)
    break
```

### 7.1.10. Случайный ход компьютера

Ход компьютера совсем немного отличается от хода игрока. Например, если выбор клетки для выстрела работает корректно, то проверять, что координаты попадают в границы поля, уже не нужно.

В остальном всё точно так же. Выводится поле, компьютер выбирает куда стрелять, вычёркивается клетка. Если было попадание, всё повторяется.

Чтобы проверить всю программу, сначала реализуем случайный выбор клетки поля, а уже потом перейдём к сложному «интеллекту». Тогда проверку того, что ход не повторяет один из прежних ходов, придётся оставить.

C++

```
while (true)
{
  std::cout << "Ваше поле:\n";
  printField(player);
  do
  {
    x = rand() % N;
    y = rand() % N;
  }
  while (player.hits[y][x]);
  std::cout << "Мой ход: " << x << ", "
  ↪  << y << "\n";
  if (!shoot(player, x, y))
  {
    std::cout << "Ваше поле:\n";
    printField(player);
    break;
  }
}
```

Python

```
while alive(player):
  print('Ваше поле:')
  printField(player)
  while True:
    x = random.randint(0, N - 1)
    y = random.randint(0, N - 1)
    if not player.hits[y][x]:
      break
  print(f'Мой ход: {x + 1}, {y + 1}')
  if not shoot(player, x, y):
    print('Ваше поле:')
    printField(player)
    break
```

### 7.1.11. Условие окончания игры

Заметили, что оба цикла для хода игроков могут работать бесконечно? В частности, если все корабли потоплены, придётся дальше стрелять в пустые клетки. А если же все клетки закончатся, даже не удастся ввести корректные координаты, чтобы сделать ход.

Условие окончания игры — это потопление последнего оставшегося корабля. Чтобы об этом узнать, можно проверить состояние массива `ships`, где для каждого из кораблей записано число «живых» клеток. И чтобы узнать, осталась ли на поле хоть одна, можно все эти числа сложить.

Выглядит не очень удобно, но если вынести этот код в функцию, то работать будет неплохо. Ведь масштаб игры невелик, и нужно всего-то проверить 10 чисел.

C++

```
bool alive(const Player &pl)
{
  for (auto s : pl.ships)
  {
    if (s)
    {
      return true;
    }
  }
  return false;
}
```

Python

```
def alive(pl):
  for s in pl.ships:
  if s > 0:
    return True
  return False
```

Теперь условие в обоих циклах `while (true)` нужно заменить на вызов функции `alive`. Только в одном случае передать туда состояние кораблей игрока, а во втором — компьютера.

Сможет ли игра завершиться теперь? Конечно же нет, ведь мы не позаботились о главном игровом цикле. Он-то по-прежнему бесконечный.

В заголовке цикла проверять условие окончания не выйдет, потому что внутри него игроки стреляют один за другим в одной итерации. И если один уже потопил все корабли, другой сможет ещё делать (успешные) ходы, пока дело не дойдёт до новой итерации главного цикла.

Поэтому проверки условия завершения игры нужно вставлять сразу после каждого из циклов, занимающихся стрельбой. Заодно и правильное сообщение можно будет вывести.

**C++**

```
// главный игровой цикл
while (true)
{
  ...
  // ход компьютера
  while (alive(player))
    ...
  if (!alive(player))
  {
    std::cout << "Я победил!\n";
    break;
  }
  // ход человека
  while (alive(computer))
    ...
  if (!alive(computer))
  {
    std::cout << "Вы выиграли!\n";
    break;
  }
}
```

**Python**

```
# главный игровой цикл
while True:
  # ход компьютера
  while alive(player):
    ...
  if not alive(player):
    print('Я победил!\n')
    break
  # ход человека
  while alive(computer):
    ...
  if not alive(computer):
    print('Вы выиграли!\n')
    break
```

⌨ | Ну теперь-то можно наконец поиграть! Или всё-таки чего-то не хватает?

### 7.1.12. Добавляем «туман войны»

Сейчас игровой процесс довольно странный, ведь когда игрок стреляет по кораблям компьютера, он видит их расположение на поле. Получается, корабли показывать нельзя.

С другой стороны, корабли показывать нужно, когда выводится поле самого игрока. Или, если он проиграл, выводится поле компьютера, чтобы показать, где на самом деле спрятана флотилия.

Получается, что признак того, показываются корабли или нет, зависит не от игрового поля, а от ситуации, в которой это поле выводится. Значит, будет удобно добавить в параметры функции вывода поля соответствующий флаг. Если он установлен, корабли показываются, а иначе скрываются.

C++

```
void printField(const Player &pl, bool
↪ hide)
{
  ...
  for (int i = 0 ; i < N ; ++i)
  {
    std::cout << std::setw(2) << i + 1;
    for (int j = 0 ; j < N ; ++j)
    {
      std::cout << ' ';
      if (pl.hits[i][j])
      {
        ...
      }
      else if (!hide && pl.field[i][j])
      {
        std::cout << '0';
      }
      else
      {
        std::cout << '.';
      }
    }
    std::cout << '\n';
  }
  std::cout << '\n';
}
```

Python

```
def printField(pl, hide):
  ...
  for i in range(N):
    print(f'{i + 1:>2}', end='')
    for j in range(N):
      print(' ', end='')
      if pl.hits[i][j]:
        ...
      elif not hide and pl.field[i][j] >
      ↪ 0:
        print('0', end='')
      else:
        print('.', end='')
    print()
  print()
```

Найдите в коде все вызовы функции **printField** и выберите, какое значение флага **hide** передавать в каждый вызов. Не забывайте, что надо не позволить игроку жульничать.

### 7.1.13. Вычёркиваем клетки вокруг потопленных кораблей

Пока что компьютер стреляет только по случайно выбранным клеткам. И иногда такие клетки могут оказаться рядом с уже потопленными кораблями.

Все мы знаем, что это неэффективно, и при игре на бумаге сразу вычёркиваем такие клетки. Но наша программа пока так не умеет, поэтому играть с ней не очень интересно.

Добавим код для вычёркивания в функцию **shoot**. Она используется для обоих игроков, так что и компьютер станет играть лучше, и человеку станет удобнее смотреть на поле соперника, где ненужные клетки сами вычёркиваются.

Но как это сделать? Чтобы найти все соседние клетки корабля, нужно знать, где находится сам корабль. А мы ведь не храним отдельно его координаты.

Можно, конечно, находить непрерывную область из поражённых клеток, когда выводится сообщение «Убит». Но есть способ проще.

Не зря же мы записывали в массиве `field` уникальный номер каждого корабля для всех его клеток. Получается, что можно просто обойти весь массив, сравнивая этот номер с числами из просматриваемых клеток.

Конечно в больших и сложных играх это было бы неэффективно, но для нашего маленького поля это хорошее упрощение кода.

Как только нужная клетка находится, мы будем вычёркивать всех её соседей. Пусть даже какие-то из них будут вычеркнуты дважды, на результат это не повлияет.

C++

```
...
std::cout << "Убит!\n";
for (int i = 0 ; i < N ; ++i)
{
  for (int j = 0 ; j < N ; ++j)
  {
    if (pl.field[i][j] == pl.field[y][x])
    {
      for (int ii = i - 1 ; ii <= i + 1 ;
      ↪  ++ii)
      {
        for (int jj = j - 1 ; jj <= j + 1
        ↪  ; ++jj)
        {
          if (ii >= 0 && ii < N
           && jj >= 0 && jj < N)
          {
            pl.hits[ii][jj] = true;
          }
        }
      }
    }
  }
}
...
```

Python

```
...
print('Убит!')
for i in range(N):
  for j in range(N):
    if pl.field[i][j] == pl.field[y][x]:
      for ii in range(i - 1, i + 2):
        for jj in range(j - 1, j + 2):
          if ii >= 0 and ii < N and jj >=
          ↪  0 and jj < N:
            pl.hits[ii][jj] = True
...
```

Четыре вложенных цикла — это вам не шутки. Но зато так играть намного удобнее. Правда, компьютер всё равно проигрывает, придётся учить его дальше.

### 7.1.14. Добивание раненого корабля

Главный недостаток игры компьютера в том, что он всегда делает случайный ход. Получается, что он в полной мере не использует вычёркиваемые «автоматически» клетки вокруг поражённых кораблей, потому что не пытается их целенаправленно топить.

Во-первых, чтобы «добивать» корабль, нужно знать итог последнего выстрела. Сейчас функция `shoot` сообщает только о попадании. А чтобы узнать, затонул

ли в результате корабль, или нет, нужно расширить диапазон возвращаемых значений, булевого типа уже недостаточно.

C++

```
enum Shoot
{
    Missed,
    Hit,
    Sunk,
};

Shoot shoot(Player &pl, int x, int y)
{
  if (!pl.hits[y][x])
  {
    pl.hits[y][x] = true;
    if (pl.field[y][x])
    {
      if (!--pl.ships[pl.field[y][x] -
      ↪ 1])
      {
        std::cout << "Убит!\n";
        ...
        return Sunk;
      }
      else
      {
        std::cout << "Ранен!\n";
        return Hit;
      }
    }
    else
    {
      std::cout << "Мимо!\n";
    }
  }
  return Missed;
}
```

Python

```
Missed = 0
Hit = 1
Sunk = 2

def shoot(pl, x, y):
  if not pl.hits[y][x]:
    pl.hits[y][x] = True
    if pl.field[y][x] > 0:
      pl.ships[pl.field[y][x] - 1] -= 1
      if pl.ships[pl.field[y][x] - 1] ==
      ↪ 0:
        print('Убит!')
        ...
        return Sunk
      else:
        print('Ранен!')
        return Hit
    else:
      print('Мимо!')
  return Missed
```

Теперь надо что-то сделать с возвращаемым результатом. Как узнать, что на поле есть раненый корабль? Запомнить его координаты во время стрельбы, если функция `shoot` вернёт значение `Hit`.

C++

```
int shipX = -1;
int shipY;
```

Python

```
shipX = -1
shipY = -1
```

Переменные `shipX` и `shipY` хранят координаты последнего попадания. Объявить эти переменные нужно до главного игрового цикла, потому что иначе не получится продолжить «добивание» корабля после промаха. После хода игрока снова наступит ход компьютера, а эти переменные будут хранить место, куда надо продолжить стрелять.

Обновляются эти координаты только тогда, когда происходит первое попадание в корабль. Если попадание не первое, делать этого не нужно, потому что от первой найденной клетки корабля мы будем искать следующие. Так как корабли бывают только вертикальные или горизонтальные, достаточно проверять соседние клетки во всех направлениях.

C++

```
// ход компьютера
while (alive(player))
{
  std::cout << "Ваше поле:\n";
  printField(player, false);
  if (shipX == -1)
  {
    do
    {
      x = rand() % N;
      y = rand() % N;
    }
    while (player.hits[y][x]);
  }
  else
  {
    ...
  }
  std::cout << "Мой ход: " << x + 1 << ",
  ↪  " << y + 1 << "\n";
  Shoot s = shoot(player, x, y);
  if (s == Hit && shipX < 0)
  {
    shipX = x;
    shipY = y;
  }
  else if (s == Sunk)
  {
    shipX = -1;
  }
  if (s == Missed)
  {
    std::cout << "Ваше поле:\n";
    printField(player, false);
    break;
  }
}
```

Python

```
# ход компьютера
while alive(player):
  print('Ваше поле:')
  printField(player, False)
  if shipX == -1:
    while True:
      x = random.randint(0, N - 1)
      y = random.randint(0, N - 1)
      if not player.hits[y][x]:
        break
  else:
    ...
  print(f'Мой ход: {x + 1}, {y + 1}')
  s = shoot(player, x, y)
  if s == Hit and shipX < 0:
    shipX = x
    shipY = y
  elif s == Sunk:
    shipX = -1
  elif s == Missed:
    print('Ваше поле:')
    printField(player, False)
    break
```

Создадим четыре похожих цикла, которые двигаются в каждом из направлений. В этих циклах компьютер немного заглядывает в расстановку кораблей игрока. Но происходит это только для поражённых клеток, поэтому правила не нарушаются. Если же в одном из направлений обнаруживается клетка, которая ещё не была поражена, туда и надо стрелять. Ну и, конечно, каждый цикл останавливается при выходе за границы поля, либо если найдена пустая поражённая клетка.

Лишних проверок тут нет, потому что раз компьютер «добивает» корабль (из-за того что функция `shoot` вернула значение `Hit`), значит, что-то на поле ещё осталось, и мы эту клетку обязательно обнаружим.

**C++**

```
x = -1;
for (int j = shipX - 1 ; j >= 0 ; --j)
{
  if (!player.hits[shipY][j])
  {
    x = j;
    y = shipY;
    break;
  }
  else if (!player.field[shipY][j])
  {
    break;
  }
}
if (x < 0)
{
  for (int j = shipX + 1 ; j < N ; ++j)
  {
    if (!player.hits[shipY][j])
    {
      x = j;
      y = shipY;
      break;
    }
    else if (!player.field[shipY][j])
    {
      break;
    }
  }
}
// Аналогичный код для поиска по
↪ вертикали
...
```

**Python**

```
x = -1
for j in range(shipX - 1, -1, -1):
  if not player.hits[shipY][j]:
    x = j
    y = shipY
    break
  elif player.field[shipY][j] == 0:
    break
if x < 0:
  for j in range(shipX + 1, N):
    if not player.hits[shipY][j]:
      x = j
      y = shipY
      break
    elif player.field[shipY][j] == 0:
      break
# Аналогичный код для поиска по вертикали
...
```

Теперь понятно, почему стартовые координаты не обновляются при каждом попадании. Ведь тогда от каждой новой клетки мы начали бы снова искать влево и вправо, даже если уже раньше находили клетку корабля, двигаясь по вертикали.

Доделайте этот фрагмент, добавив поиск нужной клетки по вертикали. Теперь-то вы играете с компьютером на равных? Или он всё ещё слабее?

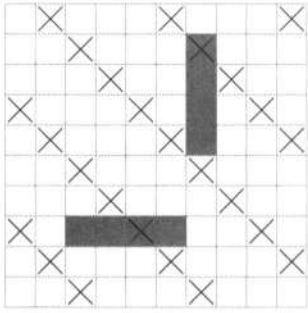
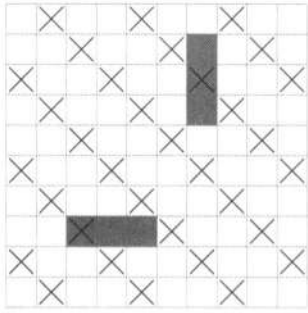
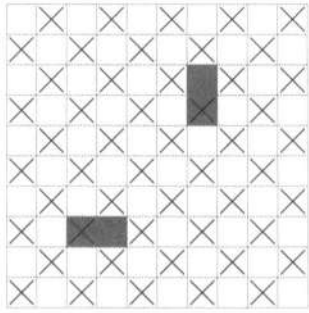
### 7.1.15. Эвристики для игры компьютера

Вроде бы компьютер уже играет довольно неплохо. Но ещё есть, что улучшить.

В книге [3] приводится понятие «минимальный тестовый залп», чтобы не только поразить все возможные корабли определённого размера, но и найти их точное положение.

В реальной игре такие залпы избыточны. Например, если мы ищем корабль максимального размера, который существует в единственном экземпляре, то в случае попадания в него в одном месте поля в других местах его уже искать не нужно.

Но на поле есть не только один корабль максимального размера, но и корабли поменьше, вплоть до $1 \times 2$ (ну и очевидно, что для успешного нахождения всех однопалубных кораблей, надо стрелять в каждую клетку).

Посмотрим на этот набор картинок. Очевидно, что если расстрелять все клетки с первой картинки, четырёхпалубный корабль скрыться не сможет. Аналогично, на второй картинке не скроется и трёхпалубный корабль. А поражение таких кораблей очень выгодно, ведь можно сразу вычеркнуть с поля очень много клеток.

Поэтому один из вариантов действий — стрелять по клеткам из этих сеток (слева направо), пока ещё остаются корабли большого размера.

С другой стороны, если четырёхпалубный корабль уже потоплен благодаря выстрелам по клеткам с первой картинки, нам придётся стрелять по клеткам со второй. А они почти не пересекаются, значительную часть работы придётся проделать заново. Но зато с первой сеткой хорошо пересекается третья.

Выходит, что идеального решения этой задачи нет, для любой стратегии есть пример, где она работает плохо. Поэтому сейчас мы запрограммируем стратегию, которая поочерёдно использует все сетки выстрелов, а если вам она не очень понравится, можно будет уже доработать её по-своему.

И, чтобы это сделать, надо понять, как определить принадлежность клетки одной из сеток и в каком порядке выбирать клетки из неё. Самый простой способ — это оставить случайный выбор клетки, как раньше, но проверять принадлежность координат правильным диагоналям. Это можно сделать, сравнив остаток от деления суммы координат на длину корабля с нулём.

Вторая проблема — где взять длину корабля. Тут программе снова придётся заглянуть в секретные структуры данных игрока, ведь в массиве `ships` есть сведения о том, потоплен корабль или нет. Информацию об этом можно получить

и в процессе стрельбы, но зачем усложнять? Ведь если мы используем ships только для проверки факта существования корабля, то это не жульничество.

Итак, случайный ход компьютера тогда изменится следующим образом:

C++

```
int len = 0;
for (int i = 0 ; !len && i < S ; ++i)
{
  if (player.ships[i])
  {
    len = shipLen[i];
  }
}
do
{
  x = rand() % N;
  y = rand() % N;
}
while (player.hits[y][x]
       || (x + y) % len != 0);
```

Python

```
len = 0
for i in range(S):
  if player.ships[i] > 0:
    len = shipLen[i]
    break
while True:
  x = random.randint(0, N - 1)
  y = random.randint(0, N - 1)
  if not player.hits[y][x] and (x + y) %
  ↪ len == 0:
    break
```

Переменная len не останется равной нулю, потому что стрельбу компьютер начинает только в том случае, когда остался хотя бы один корабль.

Ну а теперь вам удаётся обыграть программу? Если да, то придумайте ещё несколько улучшений стратегии. Например, иногда нет смысла стрелять в какую-то клетку, если большой корабль в той области в принципе не поместится.

### 7.1.16. Задания для самостоятельной работы

**1.** Добивание корабля компьютером всегда работает в одинаковом порядке: сначала горизонтально, а потом вертикально. Если игрок ставит все корабли вертикально, компьютер будет всегда тратить лишние ходы на добивание. Сделайте так, чтобы начальное направление выбиралось случайным образом.

**2.** Добавьте названия кораблям, чтобы при их расстановке и уничтожении выводился не просто размер корабля, но и название. Ну и заодно можно выводить состав флотилии, чтобы удобнее было смотреть, какие ещё корабли остались на плаву.

**Листинг 7.1. battleship.cpp**

```
#include <iostream>
#include <iomanip>
#include <time.h>
#include <stdlib.h>

const int N = 10;
const int S = 10;
const int shipLen[S] = {4, 3, 3, 2, 2, 2, 1, 1, 1, 1};

enum Shoot
{
    Missed,
    Hit,
    Sunk,
};

struct Player
{
    int field[N][N] = {};
    bool hits[N][N] = {};
    int ships[S] = {};
};

bool alive(const Player &pl)
{
    for (auto s : pl.ships)
    {
        if (s)
        {
            return true;
        }
    }
    return false;
}

void printField(const Player &pl, bool hide)
{
    std::cout << "  ";
    for (int j = 0 ; j < N ; ++j)
    {
```

```
        std::cout << ' ' << j + 1;
    }
    std::cout << '\n';
    for (int i = 0 ; i < N ; ++i)
    {
        std::cout << std::setw(2) << i + 1;
        for (int j = 0 ; j < N ; ++j)
        {
            std::cout << ' ';
            if (pl.hits[i][j])
            {
                if (pl.field[i][j])
                {
                    std::cout << '*';
                }
                else
                {
                    std::cout << 'x';
                }
            }
            else if (!hide && pl.field[i][j])
            {
                std::cout << '0';
            }
            else
            {
                std::cout << '.';
            }
        }
        std::cout << '\n';
    }
    std::cout << '\n';
}

bool checkShip(const Player &pl, int x, int y, int len, bool v)
{
    if (x < 0 || y < 0)
    {
        return false;
    }
    int endx = v ? x : x + len - 1;
```

```
    int endy = v ? y + len - 1 : y;
    if (endx >= N || endy >= N)
    {
        return false;
    }
    for (int i = -1 ; i <= len ; ++i)
    {
        for (int j = -1 ; j <= 1 ; ++j)
        {
            int xx = v ? x + j : x + i;
            int yy = v ? y + i : y + j;
            if (xx >= 0 && xx < N && yy >= 0 && yy < N)
            {
                if (pl.field[yy][xx])
                {
                    return false;
                }
            }
        }
    }
    return true;
}

void placeShip(Player &pl, int x, int y, int type, bool v)
{
    int len = shipLen[type];
    for (int i = 0 ; i < len ; ++i)
    {
        int xx = v ? x : x + i;
        int yy = v ? y + i : y;
        pl.field[yy][xx] = type + 1;
    }
    pl.ships[type] = len;
}

Shoot shoot(Player &pl, int x, int y)
{
    if (!pl.hits[y][x])
    {
        pl.hits[y][x] = true;
        if (pl.field[y][x])
```

```
        {
            if (!--pl.ships[pl.field[y][x] - 1])
            {
                std::cout << "Убит! \n";

                for (int i = 0 ; i < N ; ++i)
                {
                    for (int j = 0 ; j < N ; ++j)
                    {
                        if (pl.field[i][j] == pl.field[y][x])
                        {
                            for (int ii = i - 1 ; ii <= i + 1 ; ++ii)
                            {
                                for (int jj = j - 1 ; jj <= j + 1 ;
                                ↪ ++jj)
                                {
                                    if (ii >= 0 && ii < N && jj >= 0
                                    ↪ && jj < N)
                                    {
                                        pl.hits[ii][jj] = true;
                                    }
                                }
                            }
                        }
                    }
                }
                return Sunk;
            }
            else
            {
                std::cout << "Ранен! \n";
                return Hit;
            }
        }
        else
        {
            std::cout << "Мимо! \n";
        }
    }
    return Missed;
}
```

```
int main()
{
    srand((int)time(NULL));

    Player player, computer;
    int shipX = -1, shipY;
    int x, y;
    // расстановка кораблей игрока
    for (int s = 0 ; s < S ; ++s)
    {
        printField(player, false);
        char c;
        bool v;
        do
        {
            std::cout << "Введите позицию для корабля " << s + 1
                << " (" << shipLen[s] << "-палубный) (x, y"
                << (shipLen[s] > 1 ? ", v" : "") << "): ";
            std::cin >> x >> y;
            if (shipLen[s] > 1)
            {
                std::cin >> c;
            }
            --x;
            --y;
            v = c == 'v' || c == 'V';
        }
        while (!checkShip(player, x, y, shipLen[s], v));
        placeShip(player, x, y, s, v);
    }
    // расстановка кораблей компьютера
    for (int s = 0 ; s < S ; ++s)
    {
        bool v;
        do
        {
            x = rand() % N;
            y = rand() % N;
            v = rand() % 2;
        }
```

```
        while (!checkShip(computer, x, y, shipLen[s], v));
        placeShip(computer, x, y, s, v);
    }
    while (true)
    {
        // ход компьютера
        while (alive(player))
        {
            std::cout << "Ваше поле:\n";
            printField(player, false);
            if (shipX == -1)
            {
                int len = 0;
                for (int i = 0 ; !len && i < S ; ++i)
                {
                    if (player.ships[i])
                    {
                        len = shipLen[i];
                    }
                }
                do
                {
                    x = rand() % N;
                    y = rand() % N;
                }
                while (player.hits[y][x] || (x + y) % len != 0);
            }
            else
            {
                x = -1;
                for (int j = shipX - 1 ; j >= 0 ; --j)
                {
                    if (!player.hits[shipY][j])
                    {
                        x = j;
                        y = shipY;
                        break;
                    }
                    else if (!player.field[shipY][j])
                    {
                        break;
```

```
    }
}
if (x < 0)
{
    for (int j = shipX + 1 ; j < N ; ++j)
    {
        if (!player.hits[shipY][j])
        {
            x = j;
            y = shipY;
            break;
        }
        else if (!player.field[shipY][j])
        {
            break;
        }
    }
}
if (x < 0)
{
    for (int i = shipY - 1 ; i >= 0 ; --i)
    {
        if (!player.hits[i][shipX])
        {
            x = shipX;
            y = i;
            break;
        }
        else if (!player.field[i][shipX])
        {
            break;
        }
    }
}
if (x < 0)
{
    for (int i = shipY + 1 ; i < N ; ++i)
    {
        if (!player.hits[i][shipX])
        {
            x = shipX;
```

```
                    y = i;
                    break;
                }
                else if (!player.field[i][shipX])
                {
                    break;
                }
            }
        }
    }
    std::cout << "Мой ход: " << x + 1 << ", " << y + 1 <<
    ↪ "\n";
    Shoot s = shoot(player, x, y);
    if (s == Hit && shipX < 0)
    {
        shipX = x;
        shipY = y;
    }
    else if (s == Sunk)
    {
        shipX = -1;
    }
    else if (s == Missed)
    {
        std::cout << "Ваше поле:\n";
        printField(player, false);
        break;
    }
}
if (!alive(player))
{
    std::cout << "Ваше поле:\n";
    printField(player, false);
    std::cout << "Я победил!\nВот мои корабли:\n";
    printField(computer, false);
    break;
}
// ход человека
while (alive(computer))
{
    std::cout << "Моё поле:\n";
```

```
            printField(computer, true);
            do
            {
                std::cout << "Куда будете стрелять (x, y)? ";
                std::cin >> x >> y;
                --x;
                --y;
            }
            while (x < 0 || y < 0 || x >= N || y >= N ||
            ↪  computer.hits[y][x]);
            if (shoot(computer, x, y) == Missed)
            {
                std::cout << "Моё поле:\n";
                printField(computer, true);
                break;
            }
        }
        if (!alive(computer))
        {
            std::cout << "Моё поле:\n";
            printField(computer, false);
            std::cout << "Вы выиграли!\n";
            break;
        }
    }
}
```

**Листинг 7.2. battleship.py**

```
#!/usr/bin/python3
import random

N = 10
S = 10
shipLen = [4, 3, 3, 2, 2, 2, 1, 1, 1, 1]
Missed = 0
Hit = 1
Sunk = 2

class Player:
    def __init__(self):
```

```
        self.field = [[0] * N for _ in range(N)]
        self.hits = [[False] * N for _ in range(N)]
        self.ships = [0] * S

def alive(pl):
    for s in pl.ships:
        if s > 0:
            return True
    return False

def printField(pl, hide):
    print('  ', end='')
    for j in range(N):
        print(f' {j + 1}', end='')
    print()
    for i in range(N):
        print(f'{i + 1:>2}', end='')
        for j in range(N):
            print(' ', end='')
            if pl.hits[i][j]:
                if pl.field[i][j] > 0:
                    print('*', end='')
                else:
                    print('x', end='')
            elif not hide and pl.field[i][j] > 0:
                print('0', end='')
            else:
                print('.', end='')
        print()
    print()

def checkShip(pl, x, y, len, v):
    if x < 0 or y < 0:
        return False

    endx = x if v else x + len - 1
    endy = y + len - 1 if v else y
    if endx >= N or endy >= N:
        return False

    for i in range(-1, len + 1):
```

```
        for j in range(-1, 2):
            xx = x + j if v else x + i
            yy = y + i if v else y + j
            if xx >= 0 and xx < N and yy >= 0 and yy < N:
                if pl.field[yy][xx] > 0:
                    return False
    return True

def placeShip(pl, x, y, type, v):
    len = shipLen[type]
    for i in range(len):
        xx = x if v else x + i
        yy = y + i if v else y
        pl.field[yy][xx] = type + 1
    pl.ships[type] = len

def shoot(pl, x, y):
    if not pl.hits[y][x]:
        pl.hits[y][x] = True
        if pl.field[y][x] > 0:
            pl.ships[pl.field[y][x] - 1] -= 1
            if pl.ships[pl.field[y][x] - 1] == 0:
                print('Убит!')

                for i in range(N):
                    for j in range(N):
                        if pl.field[i][j] == pl.field[y][x]:
                            for ii in range(i - 1, i + 2):
                                for jj in range(j - 1, j + 2):
                                    if ii >= 0 and ii < N and jj >= 0
                                    ↪ and jj < N:
                                        pl.hits[ii][jj] = True
                return Sunk
            else:
                print('Ранен!')
                return Hit
        else:
            print('Мимо!')
    return Missed
```

```
player = Player()
computer = Player()
shipX = -1
shipY = -1
# расстановка кораблей игрока
for s in range(S):
    printField(player, False)
    while True:
        str = input(f'Введите позицию для корабля {s + 1}
        ↪ ({shipLen[s]}-палубный) (x, y{", v" if shipLen[s] > 1
        ↪ else ""}): ')
        if shipLen[s] > 1:
            x, y, c = str.split()
        else:
            x, y = str.split()
        x = int(x) - 1
        y = int(y) - 1
        v = c == 'v' or c == 'V'
        if checkShip(player, x, y, shipLen[s], v):
            break
    placeShip(player, x, y, s, v)

# расстановка кораблей компьютера
for s in range(S):
    while True:
        x = random.randint(0, N - 1)
        y = random.randint(0, N - 1)
        v = random.choice([True, False])
        if checkShip(computer, x, y, shipLen[s], v):
            break
    placeShip(computer, x, y, s, v)

# главный игровой цикл
while True:
    # ход компьютера
    while alive(player):
        print('Ваше поле:')
        printField(player, False)
        if shipX == -1:
            len = 0
            for i in range(S):
```

```
            if player.ships[i] > 0:
                len = shipLen[i]
                break
        while True:
            x = random.randint(0, N - 1)
            y = random.randint(0, N - 1)
            if not player.hits[y][x] and (x + y) % len == 0:
                break
    else:
        x = -1
        for j in range(shipX - 1, -1, -1):
            if not player.hits[shipY][j]:
                x = j
                y = shipY
                break
            elif player.field[shipY][j] == 0:
                break
        if x < 0:
            for j in range(shipX + 1, N):
                if not player.hits[shipY][j]:
                    x = j
                    y = shipY
                    break
                elif player.field[shipY][j] == 0:
                    break
        if x < 0:
            for i in range(shipY - 1, -1, -1):
                if not player.hits[i][shipX]:
                    x = shipX
                    y = i
                    break
                elif player.field[i][shipX] == 0:
                    break
        if x < 0:
            for i in range(shipY + 1, N):
                if not player.hits[i][shipX]:
                    x = shipX
                    y = i
                    break
                elif player.field[i][shipX] == 0:
                    break
```

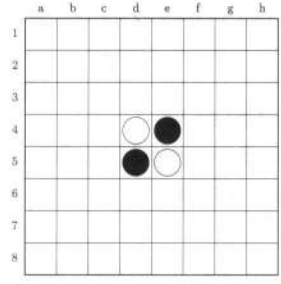
```
        print(f'Мой ход: {x + 1}, {y + 1}')
        s = shoot(player, x, y)
        if s == Hit and shipX < 0:
            shipX = x
            shipY = y
        elif s == Sunk:
            shipX = -1
        elif s == Missed:
            print('Ваше поле:')
            printField(player, False)
            break

    if not alive(player):
        print('Ваше поле:')
        printField(player, False)
        print('Я победил! \nВот мои корабли:')
        printField(computer, False)
        break

    # ход человека
    while alive(computer):
        print('Моё поле:')
        printField(computer, True)
        while True:
            str = input('Куда будете стрелять (x, y)? ')
            x, y = str.split()
            x = int(x) - 1
            y = int(y) - 1
            if x >= 0 and y >= 0 and x < N and y < N and not
            ↪ computer.hits[y][x]:
                break
        if shoot(computer, x, y) == Missed:
            print('Моё поле:')
            printField(computer, True)
            break

    if not alive(computer):
        print('Моё поле:')
        printField(computer, False)
        print('Вы выиграли!')
        break
```

## 7.2. Игра «Реверси»

Игра «Реверси» немного напоминает шашки – квадратная доска, на которой расположены одинаковые фишки двух цветов, чёрного и белого. А ещё игроки ходят поочерёдно.

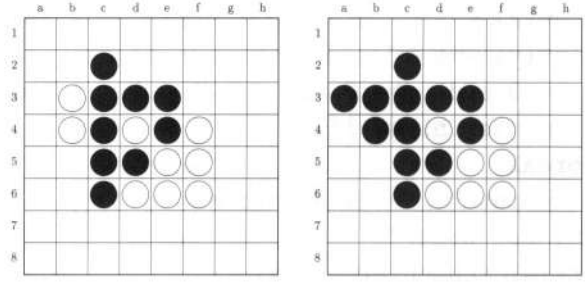
На этом сходство заканчивается и начинаются различия. В «Реверси» при каждом ходе на доске появляется одна фишка игрока, а некоторые фишки противника меняют цвет на противоположный. Цель игры – получить как можно больше фишек своего цвета в конце игры.

Начинается игра, когда у каждого из игроков по две фишки на поле:




Первыми ходят «белые». Каждый ход заключается в том, чтобы найти прилегающую к фишкам противника пустую клетку и поставить туда свою фишку. При этом игрок меняет цвет тех фишек противника, что оказались «заперты» между новой фишкой и одной из своих прежних.

Вот, например, чёрные делают ход на клетку $a3$. В результате две белые фишки на $b3$ и $b4$ оказываются зажатыми между чёрными и из-за этого меняют цвет:




Иногда бывает, что игрок не может сделать ход. Тогда право хода получает соперник.

Стратегию игры в «Реверси» придумать не так-то просто, поэтому создадим такую программу, которая вместо сложных размышлений будет действовать в соответствии с эвристикой. Эвристику придумать тоже сложно, но просмотром дерева игры и поиском оптимального хода мы займёмся только в следующих главах.

**Экран 1. Фрагмент партии в «Реверси»**

```
Куда будете ходить? g5
  a b c d e f g h
1 . . . . . . . .
2 . . . . . . . .
3 . . O O O # . #
4 . . O O O O # .
5 . . O O O O O .
6 . . . . O # # #
7 . . . . . . . .
8 . . . . . . . .
У вас 13 фишек, у меня 6 фишек

Мой ход c6
  a b c d e f g h
1 . . . . . . . .
2 . . . . . . . .
3 . . O O O # . #
4 . . O O # O # .
5 . . O # O O O .
6 . . # . O # # #
7 . . . . . . . .
8 . . . . . . . .
У вас 11 фишек, у меня 9 фишек

Куда будете ходить? f7
  a b c d e f g h
1 . . . . . . . .
2 . . . . . . . .
3 . . O O O # . #
4 . . O O # O # .
5 . . O # O O O .
6 . . # . O O # #
7 . . . . . O . .
8 . . . . . . . .
У вас 13 фишек, у меня 8 фишек

Мой ход f8
  a b c d e f g h
1 . . . . . . . .
2 . . . . . . . .
3 . . O O O # . #
4 . . O O # # # .
5 . . O # O # O .
6 . . # . O # # #
7 . . . . . # . .
8 . . . . . # . .
У вас 9 фишек, у меня 13 фишек

Куда будете ходить? h4
  a b c d e f g h
1 . . . . . . . .
2 . . . . . . . .
3 . . O O O # . #
4 . . O O O O O O
5 . . O # O # O .
6 . . # . O # # #
7 . . . . . # . .
8 . . . . . # . .
У вас 13 фишек, у меня 10
↪ фишек

Мой ход h5
  a b c d e f g h
1 . . . . . . . .
2 . . . . . . . .
3 . . O O O # . #
4 . . O O O O # #
5 . . O # O # # #
6 . . # . O # # #
7 . . . . . # . .
8 . . . . . # . .
У вас 10 фишек, у меня 14
↪ фишек
```

### 7.2.1. Игровая позиция

Реверси — игра с открытой информацией. Всё что есть у игроков — это общая доска с фишками. Цвет и расположение этих фишек известны обоим игрокам.

Поэтому игровая позиция — это то, что сейчас происходит на доске: где расположены фишки и какого они цвета.

Как обычно, в таких играх удобно создать двумерный массив для описания состояния клеток. Одна ячейка массива — одна клетка.

C++

```
const char EMPTY = '.';
const char BLACK = '#';
const char WHITE = 'O';

const int N = 8;

char board[N][N];
```

Python

```
EMPTY = '.'
BLACK = '#'
WHITE = 'O'

N = 8

board = [[EMPTY] * N for _ in range(N)]
```

Сразу выберем символы, которые будут храниться в массиве-описании доски. Всего будет три вида символов: пустая клетка `'.'`, фишка первого игрока `'O'` и фишка второго игрока `'#'`.

Соответственно, изначально надо заполнить массив точками и поставить четыре стартовые фишки:

C++

```
int main()
{
  for (int i = 0 ; i < N ; ++i)
  {
    for (int j = 0 ; j < N ; ++j)
    {
      board[i][j] = EMPTY;
    }
  }
  board[N / 2 - 1][N / 2 - 1] = WHITE;
  board[N / 2][N / 2] = WHITE;
  board[N / 2][N / 2 - 1] = BLACK;
  board[N / 2 - 1][N / 2] = BLACK;
  ...
}
```

Python

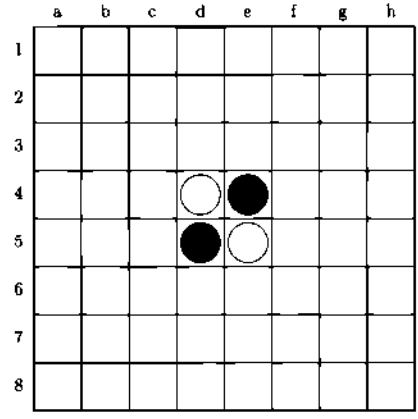
```
# пустыми значениями массив уже
↪ инициализирован при создании

board[N // 2 - 1][N // 2 - 1] = WHITE
board[N // 2][N // 2] = WHITE
board[N // 2][N // 2 - 1] = BLACK
board[N // 2 - 1][N // 2] = BLACK
```

Координаты центра доски, где стоят стартовые фишки, мы описываем как выражения с использованием $N$, то есть размера доски. Так её можно уменьшать и увеличивать, если захочется.

### 7.2.2. Вывод позиции на экран

Выбор символов, удобных для человека, для кодирования состояний отдельных клеток удобен и при отладке, и при выводе игровой позиции на экран.

Но кроме позиции нужно вывести также ещё и координатные оси, чтобы было удобно выбирать ход. Для реверси принято использовать координаты в формате «латинская буква»«цифра», причём буква задаёт позицию клетки по горизонтали, а цифра по вертикали:




**C++**

```
#include <iostream>
#include <iomanip>

void printBoard()
{
  std::cout << ' ';
  for (int j = 0 ; j < N ; ++j)
  {
    std::cout << ' ' << (char)('a' + j);
  }
  std::cout << '\n';
  for (int i = 0 ; i < N ; ++i)
  {
    std::cout << i + 1;
    for (int j = 0 ; j < N ; ++j)
    {
      std::cout << ' ' << board[i][j];
    }
    std::cout << '\n';
  }
  std::cout << "\n";
}
```

Чтобы получить буквы a, b, c, d, ..., мы прибавляем к коду символа `'a'` номер столбца, а затем приводим получившееся число к типу `char`. Очень удобно, что это просто один из целых типов и можно проводить с ним вычисления.

**Python**

```
def printBoard():
  print(' ', end='')
  for j in range(N):
    print(f' {chr(ord("a") + j)}',
    ↪ end='')
  print()
  for i in range(N):
    print(f'{i + 1}', end='')
    for j in range(N):
      print(f' {board[i][j]}', end='')
    print()
  print()
```

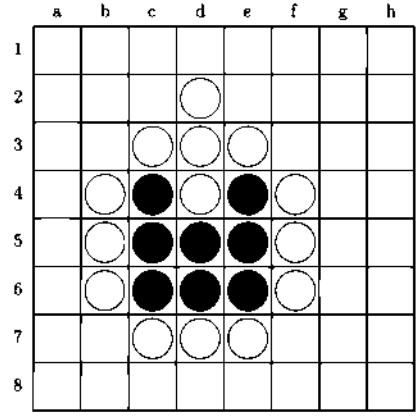
В Python к символу число прибавить нельзя, поэтому сначала мы получаем код символа с помощью функции `ord`, а после инкремента из кода обратно делаем символ с помощью функции `chr`.

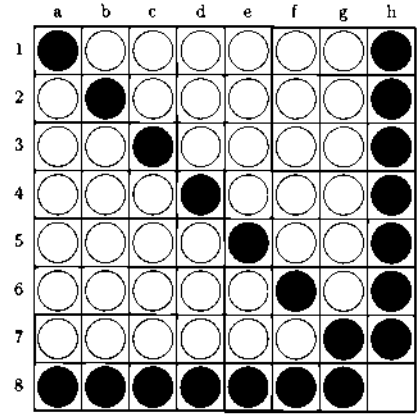
### 7.2.3. Главный игровой цикл

Игроки делают ходы поочерёдно. Поэтому в главном игровом цикле можно делать два действия: ход белых и ход чёрных. И повторять их, пока у игроков ещё есть доступные ходы.

Но как узнать, есть ли ход у игрока? Просто проверить, что хватает пустых клеток, недостаточно. Ведь иногда в пустые клетки не может походить один из игроков, а иногда и оба.

Здесь на первом рисунке нет хода у белых, а на втором — и у белых, и у чёрных:




Может, эти позиции и выдуманные, но ведь мы не можем строго обосновать, что нечто подобное не возникнет в реальной партии. Поэтому мы обязаны учесть такую возможность в программе.

C++

```
int main()
{
  ...
  while (hasMove(WHITE) ||
  ↪  hasMove(BLACK))
  {
    // ход игрока
    printBoard();
    ...
    // ход компьютера
    printBoard();
    ...
  }
  std::cout << "Игра окончена\n";
  printBoard();
}
```

Python

```
...
while hasMove(WHITE) or hasMove(BLACK):
  # ход игрока
  printBoard()
  ...
  # ход компьютера
  printBoard()
  ...

print('Игра окончена')
printBoard()
```

Функция `hasMove` должна проверять, есть ли доступные ходы для соответствующего игрока. Так как ход заключается в том, что фишка выставляется на пустую клетку, эта функция должна проверить их все.

C++

```
bool hasMove(char player)
{
  for (int i = 0 ; i < N ; ++i)
  {
    for (int j = 0 ; j < N ; ++j)
    {
      if (checkMove(player, j, i))
      {
        return true;
      }
    }
  }
  return false;
}
```

Python

```
def hasMove(player):
  for i in range(N):
    for j in range(N):
      if checkMove(player, j, i):
        return True
  return False
```

И опять новая функция! checkMove уже проверяет допустимость хода в конкретную клетку. Почему бы этот код не написать прямо здесь? Да потому что он пригодится ещё при проверке допустимости хода игрока (мало ли что он там ввёл с клавиатуры).

### 7.2.4. Проверка допустимости хода

При каких условиях ход будет допустимым? Ну, во-первых, если координаты попадают в игровое поле. Во-вторых, если выбранная клетка не занята. А в-третьих, если при этом удастся перевернуть какие-нибудь фишки соперника.

Первые два условия закодировать легко:

C++

```
int checkMove(char player, int x, int y)
{
  if (x < 0 || x >= N || y < 0 || y >= N
    || board[y][x] != EMPTY)
  {
    return 0;
  }
  ...
}
```

Python

```
def checkMove(player, x, y):
  if x < 0 or x >= N \
    or y < 0 or y >= N \
    or board[y][x] != EMPTY:
    return 0
```

Функция возвращает целое число — количество фишек, переворачиваемых в случае успеха. Это нам пригодится при написании «искусственного интеллекта» для программы.

Поэтому, когда checkMove будет искать возможность «захватить» чужие фишки, она заодно их посчитает. Чтобы сделать это, нужно перебрать «лучи», ведущие во всех восьми направлениях от выбранной клетки.

А для этого функция проверяет все клетки рядом с заданной. А точнее, все смещения соседних клеток. Они лежат в диапазоне от $-1$ до 1 по обеим координатам. Для простоты перебора используются циклы по dx и dy, которые и

обозначают смещения. Но внутри циклов приходится пропустить вариант, когда оба смещения равны нулю, потому что такой «луч» никуда не ведёт.

C++

```
int res = 0;
for (int dx = -1 ; dx <= 1 ; ++dx)
{
  for (int dy = -1 ; dy <= 1 ; ++dy)
  {
    if (!dx && !dy)
    {
      continue;
    }
    res += checkDir(player, x, y, dx,
    ↪  dy);
  }
}
return res;
```

Python

```
res = 0
for dx in range(-1, 2):
  for dy in range(-1, 2):
    if dx == 0 and dy == 0:
      continue
    res += checkDir(player, x, y, dx, dy)
return res
```

Зная смещение соседней клетки, можно многократно прибавлять его в цикле к координатам, чтобы двигаться по лучу. Эту сложную проверку мы вынесем в функцию `checkDir`, так как она тоже понадобится не один раз. Возвращает эта функция число фишек, которые можно перевернуть для заданного направления.

C++

```
int checkDir(char player, int x, int y,
↪  int dx, int dy)
{
  int res = 0;
  int xx = x + dx;
  int yy = y + dy;
  while (xx >= 0 && xx < N && yy >= 0 &&
  ↪  yy < N)
  {
    if (board[yy][xx] == player)
    {
      return res;
    }
    else if (board[yy][xx] == EMPTY)
    {
      return 0;
    }
    ++res;
    xx += dx;
    yy += dy;
  }
  return 0;
}
```

Python

```
def checkDir(player, x, y, dx, dy):
  res = 0
  xx = x + dx
  yy = y + dy
  while xx >= 0 and xx < N and yy >= 0
  ↪  and yy < N:
    if board[yy][xx] == player:
      return res
    elif board[yy][xx] == EMPTY:
      return 0
    res += 1
    xx += dx
    yy += dy
  return 0
```

Подсчёт происходит в цикле, который останавливается, если:

- луч вышел за пределы поля;
- встретилась собственная фишка (такая же, как параметр `player`);
- встретилась пустая клетка.

В первом и третьем случае функция вернёт значение 0, потому что для «захвата» нужно, чтобы собственные фишки были с обеих сторон.

А вот если собственная фишка всё же нашлась, нужно вернуть посчитанные в процессе перемещения по лучу чужие фишки. Это происходит в конце тела цикла, когда проверка исключила, что клетка поля пустая или занята собственной фишкой. Следовательно, фишка там чужая, и её нужно посчитать.

### 7.2.5. Ход игрока

Ход игрока, с одной стороны, сделать просто — всего лишь спросить, куда ставить фишку, а потом разместить её. А, с другой стороны, сначала нужно убедиться, что ход вообще возможен. Потом, что игрок ввёл корректные координаты. А после этого не только разместить новую фишку, но и «перевернуть» фишки соперника.

Для проверки возможности хода у нас уже есть функция `hasMove`. И так удачно совпало, что она вызывает функцию `checkMove`, которая как раз сейчас нужна, чтобы узнать, правильно ли игрок указал координаты. С помощью этих функций можно написать почти готовый алгоритм для хода игрока:

**C++**

```
int x, y;
printBoard();
if (!hasMove(WHITE))
{
  std::cout << "Вам некуда ходить\n";
}
else
{
  do
  {
    std::cout << "Куда будете ходить? ";
    char c;
    std::cin >> c;
    x = c - 'a';
    std::cin >> c;
    y = c - '1';
  }
  while (!checkMove(WHITE, x, y));
  makeMove(WHITE, x, y);
}
```

**Python**

```
printBoard()
if not hasMove(WHITE):
  print('Вам некуда ходить')
else:
  x = -1
  y = -1
  while not checkMove(WHITE, x, y):
    s = input('Куда будете ходить? ')
    x = ord(s[0]) - ord('a')
    y = int(s[1]) - 1
  makeMove(WHITE, x, y)
```

Как видите, здесь осталась нереализованной только функция `makeMove`, выставляющая фишку на указанную позицию.

### 7.2.6. Переворачивание фишек

Давайте уже допишем функцию `makeMove`, отвечающую за «переворачивание» фишек, чтобы можно было поиграть.

Так как корректность хода уже проверена раньше, здесь остаётся только выполнить манипуляции с фишками: поставить новую в выбранную клетку, а затем перевернуть старые.

Для переворачивания фишек снова будем проверять все направления, как это было в функции `checkMove`. И дальше нужно идти в выбранном направлении, переворачивая фишки соперника.

Но что произойдёт, если там будут только они, а в конце ряда своей фишки не окажется? Значит, сначала нужно проверить, что в этом направлении всё хорошо. А для этого как раз можно использовать написанную ранее функцию `checkDir`. Она возвращает число фишек, которые можно перевернуть. И если там будет хотя бы одна, значит, цикл переворачивания можно запускать.

Для единообразия вынесем этот цикл обработки одного направления в отдельную функцию. Хотя она и не будет повторно использоваться, код, вроде как, становится красивее благодаря этому.

**C++**

```
void makeMove(char player, int x, int y)
{
  board[y][x] = player;
  for (int dx = -1 ; dx <= 1 ; ++dx)
  {
    for (int dy = -1 ; dy <= 1 ; ++dy)
    {
      if (!dx && !dy)
      {
        continue;
      }
      if (checkDir(player, x, y, dx, dy))
      {
        swapDir(player, x, y, dx, dy);
      }
    }
  }
}
```

**Python**

```
def makeMove(player, x, y):
  board[y][x] = player
  for dx in range(-1, 2):
    for dy in range(-1, 2):
      if dx == 0 and dy == 0:
        continue
      if checkDir(player, x, y, dx, dy):
        swapDir(player, x, y, dx, dy)
```

Осталось добавить функцию `swapDir` для «переворачивания» фишек:

**C++**

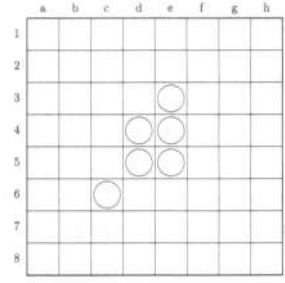
```
void swapDir(char player, int x, int y,
→  int dx, int dy)
{
  x += dx;
  y += dy;
  while (board[y][x] != player)
  {
    board[y][x] = player;
    x += dx;
    y += dy;
  }
}
```

**Python**

```
def swapDir(player, x, y, dx, dy):
  x += dx
  y += dy
  while board[y][x] != player:
    board[y][x] = player
    x += dx
    y += dy
```

Функция `swapDir` уже ничего не проверяет, так как за неё это сделала функция `checkDir`. Она просто в цикле заполняет всё фишками игрока, пока не встретится его же фишка. А собственная фишка заведомо встретится, до неё стоят только фишки соперника и нет никаких пустых клеток.

Можете попробовать поиграть, чтобы доиграться до какой-то такой позиции (ведь соперник ответить не может):




Заодно и узнаете, правильно ли работает функция `hasMove`.

### 7.2.7. Ход компьютера

Произвольный ход для компьютера сделать легко. Ведь функцию `checkMove` мы уже написали, поэтому можно выбирать случайную клетку и проверять её с помощью этой функции.

Но можно поступить немного умнее. Примерно так, как действуют начинающие игроки. Они выбирают такой ход, который переворачивает больше всего фишек соперника.

Это запрограммировать легко, потому что `checkMove` уже возвращает число переворачиваемых фишек. Следовательно, нужно перебрать все клетки поля, вызывая эту функцию для каждой из них:

**C++**

```
printBoard();
int best = 0;
for (int i = 0 ; i < N ; ++i)
{
  for (int j = 0 ; j < N ; ++j)
  {
    int c = checkMove(BLACK, j, i);
    if (c > best)
    {
      x = j;
      y = i;
      best = c;
    }
  }
}
```

**Python**

```
printBoard()
best = 0
for i in range(N):
  for j in range(N):
    c = checkMove(BLACK, j, i)
    if c > best:
      x = j
      y = i
      best = c
```

После этого для хода берётся лучший найденный результат. При этом функция hasMove здесь не понадобится, потому что мы и так перебираем все возможные ходы. И если ни один не найден, то ход нужно пропустить.

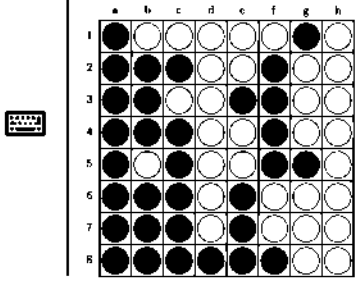
**C++**

```
if (best > 0)
{
  std::cout << "Мой ход " << (char)('a' +
  ↪  x) << y + 1 << "\n";
  makeMove(BLACK, x, y);
}
else
{
  std::cout << "Мне некуда ходить\n";
}
```

**Python**

```
if best > 0:
  print(f'Мой ход {chr(ord("a") + x)}{y
  ↪  + 1}')
  makeMove(BLACK, x, y)
else:
  print('Мне некуда ходить')
```

Ну всё, уже настоящая игра. Легко ли понять в конце, кто победил?



## 7.2.8. Окончание игры

Когда партия заканчивается, можно взглянуть на доску и понять, кто же победил. Но для этого придётся пересчитать все фишки, особенно когда их почти поровну.

Конечно же, считать вручную неинтересно. Поручим эту задачу компьютеру.

**C++**

```
std::cout << "Игра окончена\n";
printBoard();
int white = count(WHITE);
int black = count(BLACK);
if (white > black)
{
  std::cout << "Вы победили!\n";
}
else if (black > white)
{
  std::cout << "Я выиграл!\n";
}
else
{
  std::cout << "Ничья\n";
}
```

**Python**

```
print('Игра окончена')
printBoard()
white = count(WHITE)
black = count(BLACK)
if white > black:
  print('Вы победили!')
elif black > white:
  print('Я выиграл!')
else:
  print('Ничья')
```

Тут опять появилась новая функция `count`, которая подсчитывает клетки выбранного цвета. То есть те, где уже стоят фишки. С ней всё просто — обходим все клетки и проверяем, находится ли там нужная фишка.

C++

```
int count(char player)
{
  int res = 0;
  for (int i = 0 ; i < N ; ++i)
  {
    for (int j = 0 ; j < N ; ++j)
    {
      if (board[i][j] == player)
      {
        ++res;
      }
    }
  }
  return res;
}
```

Python

```
def count(player):
  res = 0
  for i in range(N):
    for j in range(N):
      if board[i][j] == player:
        res += 1
  return res
```



Доработайте функцию вывода доски `printBoard`, добавив туда текущий расклад — у кого сколько фишек. Так можно будет сразу понимать, кто выигрывает.

### 7.2.9. Задаём приоритет клеткам

Сейчас компьютер играет не лучшим образом. Например, он может, не сопротивляясь, «отдать» угловую клетку. А из литературы (например, [3]) мы знаем, что это довольно выгодная позиция, потому что фишку в такой клетке никак не перевернуть. И она очень хорошо помогает захватывать края поля. А занятые края очень удобны для захвата клеток в центре.

Вот у нас и получилась эвристика. Чтобы победить, нужно захватывать углы и края игрового поля.

Теперь придумаем, как эти приоритеты реализовать. Самый простой способ — это добавить весовые коэффициенты для всех клеток. Так, если компьютер делает ход в угол, то «выгодность» хода, которую мы до этого измеряли в фишках, должна быть выше, чем если бы ход делался в другое место.

Этого можно достичь, домножив число переворачиваемых фишек на этот коэффициент. Например, для углов он будет 10, для клеток на краю поля — 5, для центра поля 3, а для клеток рядом с угловой или крайней — 1. Последний вариант — это способ не дать игроку занять угол. Ведь если компьютер не ставит фишку рядом с углом, то игрок угол захватить не сможет.

Запишем все эти коэффициенты в таблицу, а при выборе хода будем домножать количество фишек на число из таблицы:

**C++**

```
int cost[N][N] = {
  {10, 1, 5, 5, 5, 5, 1, 10},
  {1, 1, 1, 1, 1, 1, 1, 1},
  {5, 1, 3, 3, 3, 3, 1, 5},
  {5, 1, 3, 3, 3, 3, 1, 5},
  {5, 1, 3, 3, 3, 3, 1, 5},
  {5, 1, 3, 3, 3, 3, 1, 5},
  {1, 1, 1, 1, 1, 1, 1, 1},
  {10, 1, 5, 5, 5, 5, 1, 10},
};
...
// и доработаем выбор клетки компьютером
int c = checkMove(BLACK, j, i) *
↪  cost[i][j];
```

**Python**

```
cost = [
  [10, 1, 5, 5, 5, 5, 1, 10],
  [1, 1, 1, 1, 1, 1, 1, 1],
  [5, 1, 3, 3, 3, 3, 1, 5],
  [5, 1, 3, 3, 3, 3, 1, 5],
  [5, 1, 3, 3, 3, 3, 1, 5],
  [5, 1, 3, 3, 3, 3, 1, 5],
  [1, 1, 1, 1, 1, 1, 1, 1],
  [10, 1, 5, 5, 5, 5, 1, 10],
]
...
# и доработаем выбор клетки компьютером
c = checkMove(BLACK, j, i) * cost[i][j]
```



Стал ли компьютер играть сильнее? Попробуйте доработать таблицу стоимости клеток, если его ходы кажутся вам неудачными. Или даже придумайте совсем другую эвристику с более сложной логикой.

**Листинг 7.3. reversi.cpp**

```
#include <iostream>

const char EMPTY = '.';
const char BLACK = '#';
const char WHITE = 'O';
const int N = 8;
char board[N][N];
int cost[N][N] = {
    {10, 1, 5, 5, 5, 5, 1, 10},
    {1, 1, 1, 1, 1, 1, 1, 1},
    {5, 1, 3, 3, 3, 3, 1, 5},
    {5, 1, 3, 3, 3, 3, 1, 5},
    {5, 1, 3, 3, 3, 3, 1, 5},
    {5, 1, 3, 3, 3, 3, 1, 5},
    {1, 1, 1, 1, 1, 1, 1, 1},
    {10, 1, 5, 5, 5, 5, 1, 10},
};

int count(char player)
{
    int res = 0;
    for (int i = 0 ; i < N ; ++i)
    {
        for (int j = 0 ; j < N ; ++j)
```

```
        {
            if (board[i][j] == player)
            {
                ++res;
            }
        }
    }
    return res;
}

void printBoard()
{
    std::cout << ' ';
    for (int j = 0 ; j < N ; ++j)
    {
        std::cout << ' ' << (char)('a' + j);
    }
    std::cout << '\n';
    for (int i = 0 ; i < N ; ++i)
    {
        std::cout << i + 1;
        for (int j = 0 ; j < N ; ++j)
        {
            std::cout << ' ' << board[i][j];
        }
        std::cout << '\n';
    }
    std::cout << "У вас " << count(WHITE) << " фишек, у меня " <<
    ↪ count(BLACK) << " фишек\n\n";
}

int checkDir(char player, int x, int y, int dx, int dy)
{
    int res = 0;
    int xx = x + dx, yy = y + dy;
    while (xx >= 0 && xx < N && yy >= 0 && yy < N)
    {
        if (board[yy][xx] == player)
        {
            return res;
        }
```

```
        else if (board[yy][xx] == EMPTY)
        {
            return 0;
        }
        ++res;
        xx += dx;
        yy += dy;
    }
    return 0;
}

int checkMove(char player, int x, int y)
{
    if (x < 0 || x >= N || y < 0 || y >= N || board[y][x] != EMPTY)
    {
        return 0;
    }
    int res = 0;
    for (int dx = -1 ; dx <= 1 ; ++dx)
    {
        for (int dy = -1 ; dy <= 1 ; ++dy)
        {
            if (!dx && !dy)
            {
                continue;
            }
            res += checkDir(player, x, y, dx, dy);
        }
    }
    return res;
}

void swapDir(char player, int x, int y, int dx, int dy)
{
    x += dx;
    y += dy;
    while (board[y][x] != player)
    {
        board[y][x] = player;
        x += dx;
        y += dy;
```

```
    }
}

void makeMove(char player, int x, int y)
{
    board[y][x] = player;
    for (int dx = -1 ; dx <= 1 ; ++dx)
    {
        for (int dy = -1 ; dy <= 1 ; ++dy)
        {
            if (!dx && !dy)
            {
                continue;
            }
            if (checkDir(player, x, y, dx, dy))
            {
                swapDir(player, x, y, dx, dy);
            }
        }
    }
}

bool hasMove(char player)
{
    for (int i = 0 ; i < N ; ++i)
    {
        for (int j = 0 ; j < N ; ++j)
        {
            if (checkMove(player, j, i))
            {
                return true;
            }
        }
    }
    return false;
}

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

```
        for (int j = 0 ; j < N ; ++j)
        {
            board[i][j] = EMPTY;
        }
    }
    board[N / 2 - 1][N / 2 - 1] = WHITE;
    board[N / 2][N / 2] = WHITE;
    board[N / 2][N / 2 - 1] = BLACK;
    board[N / 2 - 1][N / 2] = BLACK;
    while (hasMove(WHITE) || hasMove(BLACK))
    {
        int x, y;
        printBoard();
        if (!hasMove(WHITE))
        {
            std::cout << "Вам некуда ходить\n";
        }
        else
        {
            do
            {
                std::cout << "Куда будете ходить? ";
                char c;
                std::cin >> c;
                x = c - 'a';
                std::cin >> c;
                y = c - '1';
            }
            while (!checkMove(WHITE, x, y));
            makeMove(WHITE, x, y);
        }
        printBoard();
        int best = 0;
        for (int i = 0 ; i < N ; ++i)
        {
            for (int j = 0 ; j < N ; ++j)
            {
                int c = checkMove(BLACK, j, i) * cost[i][j];
                if (c > best)
                {
                    x = j;
```

```
                    y = i;
                    best = c;
                }
            }
        }
        if (best > 0)
        {
            std::cout << "Мой ход " << (char)('a' + x) << y + 1 <<
            ↪  "\n";
            makeMove(BLACK, x, y);
        }
        else
        {
            std::cout << "Мне некуда ходить\n";
        }
    }
    std::cout << "Игра окончена\n";
    printBoard();
    int white = count(WHITE);
    int black = count(BLACK);
    if (white > black)
    {
        std::cout << "Вы победили!\n";
    }
    else if (black > white)
    {
        std::cout << "Я выиграл!\n";
    }
    else
    {
        std::cout << "Ничья\n";
    }
}
```

**Листинг 7.4. reversi.py**

```
#!/usr/bin/python3
EMPTY = '.'
BLACK = '#'
WHITE = 'O'
N = 8
board = [[EMPTY] * N for _ in range(N)]
```

```
cost = [
    [10, 1, 5, 5, 5, 5, 1, 10],
    [1, 1, 1, 1, 1, 1, 1, 1],
    [5, 1, 3, 3, 3, 3, 1, 5],
    [5, 1, 3, 3, 3, 3, 1, 5],
    [5, 1, 3, 3, 3, 3, 1, 5],
    [5, 1, 3, 3, 3, 3, 1, 5],
    [1, 1, 1, 1, 1, 1, 1, 1],
    [10, 1, 5, 5, 5, 5, 1, 10],
]

def count(player):
    res = 0
    for i in range(N):
        for j in range(N):
            if board[i][j] == player:
                res += 1
    return res

def printBoard():
    print('  ', end='')
    for j in range(N):
        print(f' {chr(ord("a") + j)} ', end='')
    print()
    for i in range(N):
        print(f'{i + 1} ', end='')
        for j in range(N):
            print(f' {board[i][j]} ', end='')
        print()
    print(f'У вас {count(WHITE)} фишек, у меня {count(BLACK)}
    ↪ фишек\n')

def checkDir(player, x, y, dx, dy):
    res = 0
    xx = x + dx
    yy = y + dy
    while xx >= 0 and xx < N and yy >= 0 and yy < N:
        if board[yy][xx] == player:
            return res
        elif board[yy][xx] == EMPTY:
            return 0
```

```
        res += 1
        xx += dx
        yy += dy
    return 0

def checkMove(player, x, y):
    if x < 0 or x >= N or y < 0 or y >= N or board[y][x] != EMPTY:
        return 0
    res = 0
    for dx in range(-1, 2):
        for dy in range(-1, 2):
            if dx == 0 and dy == 0:
                continue
            res += checkDir(player, x, y, dx, dy)
    return res

def swapDir(player, x, y, dx, dy):
    x += dx
    y += dy
    while board[y][x] != player:
        board[y][x] = player
        x += dx
        y += dy

def makeMove(player, x, y):
    board[y][x] = player
    for dx in range(-1, 2):
        for dy in range(-1, 2):
            if dx == 0 and dy == 0:
                continue
            if checkDir(player, x, y, dx, dy):
                swapDir(player, x, y, dx, dy)

def hasMove(player):
    for i in range(N):
        for j in range(N):
            if checkMove(player, j, i):
                return True
    return False
```

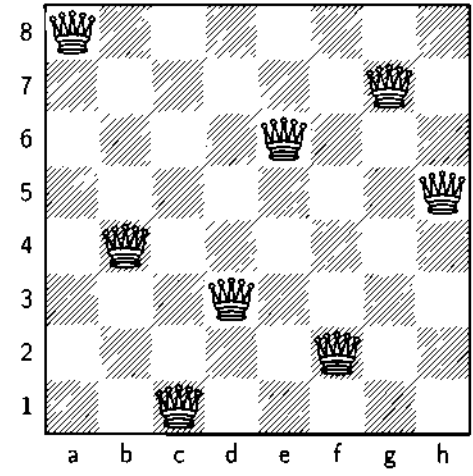
```
board[N // 2 - 1][N // 2 - 1] = WHITE
board[N // 2][N // 2] = WHITE
board[N // 2][N // 2 - 1] = BLACK
board[N // 2 - 1][N // 2] = BLACK
while hasMove(WHITE) or hasMove(BLACK):
    printBoard()
    if not hasMove(WHITE):
        print('Вам некуда ходить')
    else:
        x = -1
        y = -1
        while not checkMove(WHITE, x, y):
            s = input('Куда будете ходить? ')
            x = ord(s[0]) - ord('a')
            y = int(s[1]) - 1
        makeMove(WHITE, x, y)
    printBoard()
    best = 0
    for i in range(N):
        for j in range(N):
            c = checkMove(BLACK, j, i) * cost[i][j]
            if c > best:
                x = j
                y = i
                best = c
    if best > 0:
        print(f'Мой ход {chr(ord("a") + x)}{y + 1}')
        makeMove(BLACK, x, y)
    else:
        print('Мне некуда ходить')
print('Игра окончена')
printBoard()
white = count(WHITE)
black = count(BLACK)
if white > black:
    print('Вы победили!')
elif black > white:
    print('Я выиграл!')
else:
    print('Ничья')
```

Глава 8

# Рекурсивные алгоритмы в играх

## 8.1. Головоломка «8 ферзей»

В этой головоломке нужно расставить 8 ферзей на шахматной доске, чтобы они не атаковали друг друга. Например, вот так:




Какое-то произвольное решение задачи, а не все расстановки ферзей, найти довольно легко. Для этого можно использовать алгоритм из книги [8]. Там же есть и ссылки на другие работы по этой теме. Но мы будем искать все решения, для того чтобы научиться новому методу перебора вариантов. Ведь это потом пригодится при создании более сложных игр.

### 8.1.1. Игровая позиция

Так как в этой игре нужно расставить 8 ферзей, то игровая позиция – это их положение на доске. Каждый ферзь стоит на отдельной клетке, имеющей две координаты – строку и столбец. Поэтому можно для хранения координат использовать два массива:

C++
```
int x[8];
int y[8];
```

Python
```
x = [0] * 8
y = [0] * 8
```

Но в нашей программе это будет немного избыточным. Снова взглянем на картинку с расстановкой ферзей — каждый из них занимает собственную строку (и собственный столбец). То есть можно договориться, что в программе будут выполняться условия `x[0] = 0`, `x[1] = 1`, ..., `x[7] = 7`. Ведь нет никакого смысла пытаться разместить двух ферзей в одном столбце.

Тогда хранить эту координату не нужно, а во всех вычислениях и логических выражениях принимать `x[i] = i`. И описание игровой позиции в программе станет выглядеть так:

C++
```
int queens[8];
```

Python
```
queens = [0] * 8
```

А позиция с картинки выше кодируется такими значениями:

C++
```
int queens[8] = {7, 3, 0, 2, 5, 1, 6, 4};
```

Python
```
queens = [7, 3, 0, 2, 5, 1, 6, 4]
```

Они начинаются с 0, а не с 1, потому что такие координаты удобнее использовать. Это мы увидим, когда дойдём до вывода расстановки на экран. Да и циклы проще получаются.

### 8.1.2. Общий алгоритм поиска решений

Принцип перебора возможных решений почти не отличается от решения ребусов. Сначала находим потенциальное решение (там это был набор соответствий буква-цифра, а здесь положения ферзей на поле). Потом проверяем, подходит это решение или нет (сходится ли ребус или бьют ли ферзи друг друга). Если решение подошло, ни один из ферзей не находится под боем другого, то выводим шахматную доску на экран.

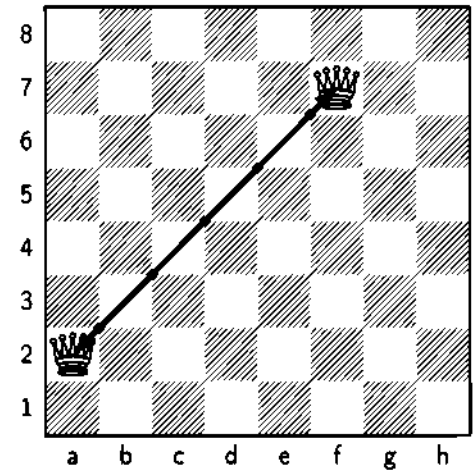
Представим, что ферзи уже расставлены. Как узнать, что ни один из них не атакован другим? Сформулировать такое условие не очень просто. Но гораздо проще проверить обратное условие — что какой-то из ферзей атакован другим ферзём. Тогда если оно не выполняется, ни один из ферзей не атакован. То есть достаточно найти пару ферзей на одной линии. И этого будет достаточно, чтобы понять, что такая расстановка не подходит. Для проверки переберём все возможные пары ферзей:

C++
```
for (int q1 = 0 ; q1 < 8 ; ++q1)
  for (int q2 = q1 + 1 ; q2 < 8 ; q2++)
    // атакует ли ферзь q1 ферзя q2?
```

Python
```
for q1 in range(8):
  for q2 in range(q1 + 1, 8):
    # атакует ли ферзь q1 ферзя q2?
```

Здесь q1 и q2 — порядковые номера ферзей, которых мы проверяем. Причём для q1 перебираются все номера от 0 до 7, а для q2 только те номера, которые больше q1. Почему так? Потому что проверять пары вроде $(5, 3)$ (где $q1 > q2$) нет никакого смысла, ведь раньше мы и так проверим аналогичную пару $(3, 5)$. А так как ферзи могут ходить в любую сторону, то если первый бьёт второго, то и второй одновременно бьёт первого. Значит, дважды проверять одну и ту же пару смысла нет.

Как на картинке ниже ферзь на $f7$ может побить ферзя на $a2$, и наоборот:




### 8.1.3. Проверка, что один ферзь атакует другого

В цикле из предыдущего раздела чего-то не хватает. Там есть комментарий, но нет непосредственно кода, проверяющего, что ферзи находятся на одной линии. Как это можно сделать?

Два ферзя находятся на одной линии (то есть атакуют друг друга) при выполнении одного из условий:

- Они оба стоят на одной горизонтали. То есть $y_1 = y_2$.
- Ферзи стоят на одной вертикали. Или $x_1 = x_2$.
- Ферзи на одной диагонали. Самое сложное условие. Как его перевести на математический язык? Попробуем двигать ферзя по диагонали клетка за клеткой. Легко заметить, что и $x$, и $y$ каждый раз меняются на 1. Значит, когда один ферзь дойдёт до другого, обе координаты изменятся на одну и ту же величину. Только непонятно, в какую сторону. Ведь если ферзь двигается вправо, то $x$ увеличивается, а если влево, то уменьшается.

  Получается, что если второму ферзю нужно пройти $d$ клеток по диагонали, чтобы «съесть» первого, то или $x_1 = x_2 + d, y_1 = y_2 + d$, или $x_1 = x_2 - d, y_1 = y_2 + d$, или $x_1 = x_2 + d, y_1 = y_2 - d$, или $x_1 = x_2 - d, y_1 = y_2 - d$.

Как-то сложно. Давайте попробуем это упростить. Во-первых, мы договорились, что все ферзи заведомо располагаются на разных вертикалях. Следовательно, второе условие, $x_1 = x_2$, не будет выполняться ни для какой пары.

Во вторых, мы смотрим на такие пары ферзей, у которых номер первого ферзя меньше номера второго. А номера — это их координаты по горизонтали. То

есть $x_1 < x_2$, следовательно, при проверке диагоналей $x_1 = x_2 + d$ не может выполниться никогда.

Остаются $x_1 = x_2 - d, y_1 = y_2 + d$ и $x_1 = x_2 - d, y_1 = y_2 - d$. Как же нам угадать эту переменную $d$? Неужели писать цикл? Вовсе нет. Нужно всего лишь преобразовать условия: $d = x_2 - x_1, d = y_1 - y_2$ и $d = x_2 - x_1, d = y_2 - y_1$.

Теперь видно, что разница между координатами по $x$ и по $y$ совпадает, а сама эта величина $d$ нам не так уж и нужна. Следовательно, два ферзя стоят на одной диагонали, если выполняется любое из условий: $x_2 - x_1 = y_1 - y_2$ или $x_2 - x_1 = y_2 - y_1$.

Соберём все эти проверки в функцию для проверки пары ферзей. В функцию передаются их координаты, и она возвращает `true`, если оба ферзя в безопасности.

C++

```
bool check(int x1, int y1, int x2, int
↪  y2)
{
  if (y1 == y2)
  {
    return false;
  }
  if (x2 - x1 == y2 - y1
   || x2 - x1 == y1 - y2)
  {
    return false;
  }
  return true;
}
```

Python

```
def check(x1, y1, x2, y2):
  if y1 == y2:
    return False
  if x2 - x1 == y2 - y1 or x2 - x1 == y1
  ↪  - y2:
    return False
  return True
```

### 8.1.4. Перебор с помощью циклов

У нас есть проверка расположения двух ферзей, есть какой-то цикл, но как это поможет найти решение задачи? Когда мы разгадывали числовые ребусы, один цикл соответствовал одной букве. Так проверялись все возможные варианты соответствия букв цифрам.

Теперь вместо букв 8 ферзей, для которых нужно выбрать клетки поля. Тогда для перебора их позиций понадобится 8 вложенных циклов, по аналогии с тем как мы делали по одному циклу на каждую букву криптарифма.

Уменьшим задачу до 4 ферзей, чтобы код был покороче. Восемь вложенных циклов всё-таки не шутка.

C++

```
for (int a = 0 ; a < 4 ; ++a)
  for (int b = 0 ; b < 4 ; ++b)
    for (int c = 0 ; c < 4 ; ++c)
      for (int d = 0 ; d < 4 ; ++d)
```

Python

```
for a in range(4):
  for b in range(4):
    for c in range(4):
      for d in range(4):
```

Здесь `a, b, c, d` — это координаты ферзей от первого до четвёртого. То есть это номера строк, которые они занимают. А номера столбцов совпадают с номерами ферзей, как мы уже договорились раньше.

Вставим в тело внутреннего цикла код, чтобы вывести шахматную доску с ферзями на экран. Если бы каждый ферзь занимал отдельную строку, а не столбец, можно было бы сильно упростить вывод. А так приходится сохранять поле в памяти и только потом выводить.

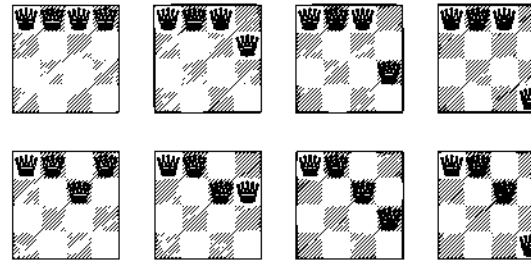
C++

```
std::string field[4] = {"....", "....",
↪  "....", "...."};
field[a][0] = 'Q';
field[b][1] = 'Q';
field[c][2] = 'Q';
field[d][3] = 'Q';
for (int i = 0 ; i < 4 ; ++i)
{
  std::cout << field[i] << "\n";
}
std::cout << "\n";
```

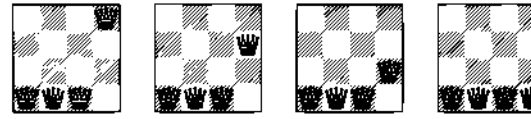
Python

```
field = [['.'] * 4, ['.'] * 4, ['.'] *
↪  4, ['.'] * 4]
field[a][0] = 'Q';
field[b][1] = 'Q';
field[c][2] = 'Q';
field[d][3] = 'Q';
for i in range(4):
  print(''.join(field[i]))
print()
```

Попробуйте запустить такой вариант программы. Должен получиться вывод всех-всех расположений ферзей, вроде таких:



И так далее, до:



### 8.1.5. Отсеиваем некорректные варианты

Очевидно, не всё, что находит программа, нужно выводить на экран, ведь в большинстве решений ферзи находятся под ударом друг друга.

Чтобы такие решения отсечь, будем использовать функцию `check`, которую мы придумали раньше. На входе этой функции координаты двух ферзей, а на выходе «истина», если позиция безопасна, то есть ферзи не атакуют друг друга.

Функция проверяет пару ферзей, а у нас их четыре. Поэтому нужно проверить все пары: 1–2, 1–3, 1–4, 2–3 и 2–4. Ферзи имеют координаты $(0, a)$, $(1, b)$, $(2, c)$ и $(3, d)$, поэтому проверки получатся такими:

C++

```
check(0, a, 1, b)
check(0, a, 2, c)
check(1, b, 2, c)
check(0, a, 3, d)
check(1, b, 3, d)
check(2, c, 3, d)
```

Python

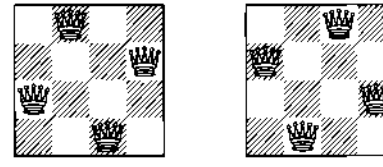
```
check(0, a, 1, b)
check(0, a, 2, c)
check(1, b, 2, c)
check(0, a, 3, d)
check(1, b, 3, d)
check(2, c, 3, d)
```

Но это только вызовы функций, а не весь код для проверки. Нужно объединить вызовы с помощью «логического и», ведь позиция будет корректным решением, если все условия выполняются.

Вставьте эти проверки в программу, чтобы решение выводилось только в случае, когда они все истинны.

Будет ли программа эффективнее, если корректность положения пары ферзей `a-b` проверять сразу внутри цикла по `b`?

В итоге должно получиться всего два решения. Если у вас так и вышло, пойдём дальше и попробуем решить задачу уже для восьми ферзей.



### 8.1.6. Что такое рекурсия

Чтобы решить полный вариант головоломки, можно было бы и восемь циклов написать. Компилятор это позволит. Но в общем случае это не лучший способ. Например, что мы будем делать, если размер доски (и число ферзей) вводится пользователем? Можно, конечно, написать все варианты циклов (вложенностью от 4 до, скажем, 10) и выбирать их с помощью оператора `if`. Но вы только представьте, как будет выглядеть эта программа. И как в ней искать ошибки и опечатки.

В подобных ситуациях, когда нужно перебирать неизвестное число параметров либо параметры с неизвестной размерностью, пригодится рекурсия.

Рекурсия — это обращение функции к самой себе. То есть внутри цикла перебора значений параметра будет вызов этой же функции, но для перебора уже следующего параметра. Примерно так:

C++
```
void find()
{
  for (int i = 0 ; i < 8 ; ++i)
    find();
}
```

Python
```
def find():
  for i in range(8):
    find()
```

Что будет делать этот код, если функцию вызвать? Завершаться с ошибкой переполнения стека, потому что он не доделан. Нужно ещё кое-что узнать.

Стек — это последовательность объектов (переменных, байтов, чего угодно), которая пополняется только с одного конца. И с того же конца элементы покидают последовательность. Например, если в стек добавить три значения 1, 6, 2, то покидать стек они будут в порядке 2, 6, 1.

Зачем это нужно? А для того, чтобы функции в программе могли вызывать друг друга. Когда происходит вызов, в стек процессора помещаются все переменные для новой функции. А когда она заканчивает работу, эти переменные из стека удаляются, и выполнение продолжается в вызывающей функции, будто бы ничего и не произошло, потому что на вершине стека будут как раз принадлежащие ей переменные.

Вот очередной пример кода:

C++
```
#include <iostream>

int f()
{
  int a = 1;
  return a;
}

int g()
{
  int b = 2;
  b += f();
  return b;
}

int main()
{
  int c = 3;
  c += g();
  std::cout << c;
}
```

Python
```
def f():
  a = 1
  return a

def g():
  b = 2;
  b += f();
  return b

def main():
  c = 3
  c += g()
  print(c)

main()
```

Состояние стека в этой программе будет меняться так:

| | | | | | |
|---|---|---|---|---|---|
| Функция f | | | a = 1 | | |
| Функция g | | b = 2 | b = 2 | b = 3 | |
| Функция main | c = 3 | c = 3 | c = 3 | c = 3 | c = 6 |

Сначала выполняется `main`, в которой есть переменная `c`. `main` вызывает функцию `g`, в стеке появляется её переменная `b`. Дальше вызывается функция `f`, для которой создаётся переменная `a`.

Когда функция `f` заканчивает работу, процессор возвращается к выполнению функции `g` и меняет там переменную `b`. Потом так же меняется и переменная `c`.

Сейчас мы вызывали три разные функции. Но вернёмся к предыдущему фрагменту, где функция вызывала сама себя. Там происходило примерно то же самое. `find` требовала места в стеке под свою переменную `i`, а затем снова вызывала саму себя. При этом создавалась новая переменная `i`, ведь вызов-то новый. И так эти переменные накапливались, пока стек не переполнялся. Память-то в компьютере не бесконечная.

Попробуйте определить, сколько таких рекурсивных вызовов происходит, прежде чем программа завершается с ошибкой. Для этого можно использовать дополнительные команды вывода на экран.

### 8.1.7. Выход из рекурсии

Мы уже выяснили, что функция `find` будет создавать переменные `i` до бесконечности. По задумке каждая такая переменная соответствует одному ферзю и описывает его координату по вертикали. Но так как ферзей всего 8, то делать больше 8 вложенных вызовов не нужно.

Чтобы ограничить рекурсию, можно передавать в функцию параметр, соответствующий номеру рассматриваемого ферзя. И когда он достигнет 8, функция может завершиться, а не продолжать поиск дальше.

C++

```
void find(int q)
{
  if (q == 8)
    return;
  for (int i = 0 ; i < 8 ; ++i)
    find(q + 1);
}
```

Python

```
def find(q):
  if q == 8:
    return
  for i in range(8):
    find(q + 1)
```

Если теперь запустить этот код с помощью вызова `find(0)`, то мы получим... ничего. Но это всё же лучше, чем аварийное завершение.

### 8.1.8. Рекурсивный перебор всех решений

Итак, в функции `find` что-то происходит. Или ничего не происходит. Как это проверить? Нужно вывести «решение», когда положение всех 8 ферзей определено. Но как это сделать, ведь позиции ферзей хранятся в локальных переменных, а обратиться к чужим локальным переменным нельзя?

Один из вариантов — записать найденных ферзей в глобальную переменную-массив queens, которую мы придумали в самом начале для хранения позиции головоломки:

C++

```
void find(int q)
{
  if (q == 8)
    return;
  for (int i = 0 ; i < 8 ; ++i)
  {
    queens[q] = i;
    find(q + 1);
  }
}
```

Python

```
def find(q):
  if q == 8:
    return
  for i in range(8):
    queens[q] = i
    find(q + 1)
```

Теперь добавим в код вывод решения. В то место, где позиции всех ферзей определены, то есть под условием q == 8:

C++

```
for (int i = 0 ; i < 8 ; ++i)
{
  std::string line(8, '.');
  line[queens[i]] = 'Q';
  std::cout << line << '\n';
}
std::cout << '\n';
```

Python

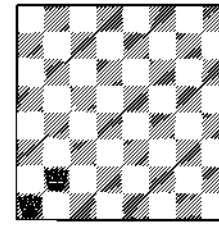
```
for i in range(N):
  pos = queens[i]
  line = '.' * pos + 'Q' + '.' * (N -
  ↪ pos - 1)
  print(line)
print()
```

Здесь мы не стали хранить всё поле в памяти, как в прошлый раз, а просто выводим каждого ферзя на своей строке вместо столбца. То есть доска повёрнута на 90°. Так как мы ищем все решения задачи, то все варианты поворотов всё равно будут найдены. А, значит, можно сразу выводить неправильно, это повлияет только на порядок ответов.

### 8.1.9. И снова отсеивание некорректных вариантов

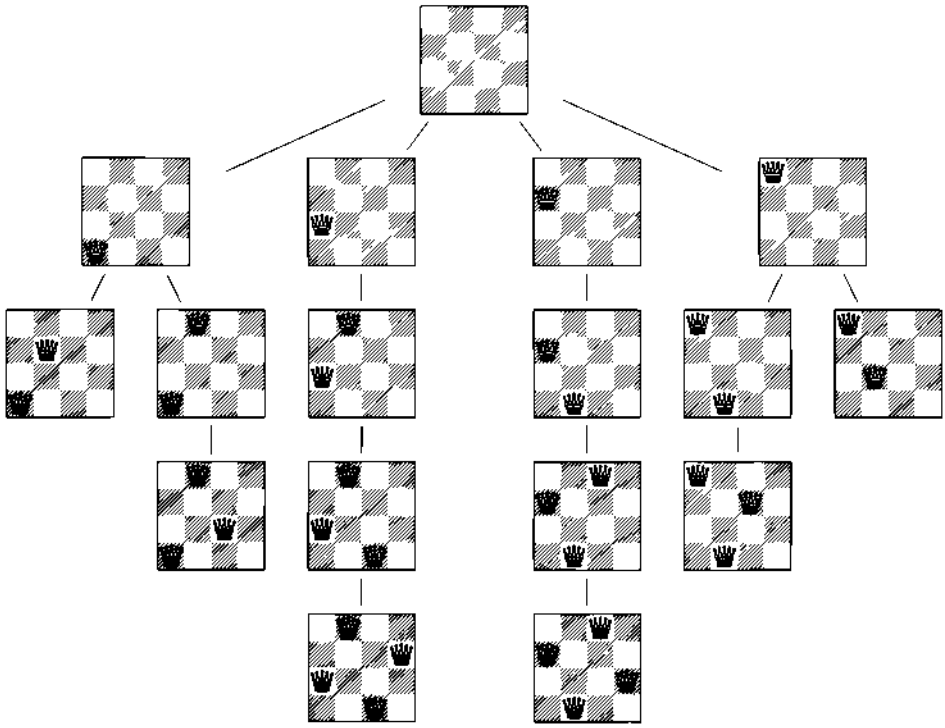
С проверкой корректности позиции (то есть предполагаемого решения) мы уже сталкивались, когда искали решение мини-задачи про ферзей, используя циклы. Нужно всего лишь проверить корректность расположения на доске каждой пары ферзей.

Но тут будет один нюанс. Посмотрим на рисунок:



Предположим, первые два ферзя расположились именно таким образом. Есть ли смысл пытаться расставить остальных? Конечно же нет, мы только потратим время, а решения всё равно не найдём.

Конечно, нам бы хотелось, чтобы перебор не выводил все позиции подряд, а искал только корректные решения. Для случая 4 ферзей дерево рекурсивных вызовов могло бы выглядеть так:



Как видите, здесь ветки перебора, ведущие к некорректным позициям, обрываются сразу, и программе не нужно тратить время впустую.

Следовательно, проверять корректность размещения очередного ферзя надо сразу, а не когда все фигуры уже расставлены, перед выводом решения. Проверять сразу можно в цикле, где функция вызывается рекурсивно. Если новый ферзь не атакует никакого из уже размещённых, то можно его оставить на этой позиции и попытаться расставить остальных. В противном случае продолжать анализировать этот вариант нет смысла.

Координаты размещённых ферзей хранятся в массиве `queens`. Только корректными из них будут ячейки от `0` до `q-1`. Потому что функция работает с ферзём под номером `q`, ферзи с меньшими номерами расставлены в вызывающих функциях, а с бóльшими ещё нет, их позиции определятся в более глубоких вызовах.

C++

```
for (int i = 0 ; i < N ; ++i)
{
  bool ok = true;
  for (int k = 0 ; ok && k < q ; ++k)
  {
    ok = check(k, queens[k], q, i);
  }
  if (ok)
  {
    queens[q] = i;
    find(q + 1);
  }
}
```

Python

```
for i in range(N):
  ok = True
  for k in range(q):
    ok = check(k, queens[k], q, i)
    if not ok:
    break
  if ok:
    queens[q] = i
    find(q + 1)
```

Вот так будет работать рекурсия, заполняя массив queens:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| find(0) | | | | | | | | |
| find(1) | 0 | | | | | | | |
| find(2) | 0 | 2 | | | | | | |
| find(3) | 0 | 2 | 4 | | | | | |
| find(4) | 0 | 2 | 4 | 1 | | | | |
| find(5) | 0 | 2 | 4 | 1 | 3 | | | |
| find(6) | 0 | 2 | 4 | 1 | 3 | 7 | | |
| find(7) | 0 | 2 | 4 | 1 | 3 | 7 | 5 | зашли в тупик, возвращаемся назад |
| find(5) | 0 | 2 | 4 | 1 | 7 | | | |
| find(4) | 0 | 2 | 4 | 6 | | | | |
| find(5) | 0 | 2 | 4 | 6 | 1 | | | |
| find(6) | 0 | 2 | 4 | 6 | 1 | 3 | | |
| find(7) | 0 | 2 | 4 | 6 | 1 | 3 | 5 | |
| find(8) | 0 | 2 | 4 | 6 | 1 | 3 | 5 | 7 | первое решение найдено |

Когда вызов функции `find(q)` завершается, ячейку `q` в массиве никто не очищает. В таблице такие ячейки показаны пустыми, потому что их значения не используются в проверке позиции очередного ферзя.

Теперь нужно собрать все фрагменты кода вместе. Сколько решений выдаёт программа? Корректны ли они?

**Листинг 8.1. 8queensLoop.cpp**

```
#include <iostream>

bool check(int x1, int y1, int x2, int y2)
{
  if (y1 == y2)
  {
    return false;
```

```
  }
  if (x2 - x1 == y2 - y1 || x2 - x1 == y1 - y2)
  {
    return false;
  }
  return true;
}

int main()
{
  for (int a = 0 ; a < 4 ; ++a)
  {
    for (int b = 0 ; b < 4 ; ++b)
    {
      if (!check(0, a, 1, b))
      {
        continue;
      }
      for (int c = 0 ; c < 4 ; ++c)
      {
        if (!check(0, a, 2, c) || !check(1, b, 2, c))
        {
          continue;
        }
        for (int d = 0 ; d < 4 ; ++d)
        {
          if (!check(0, a, 3, d) || !check(1, b, 3, d) || !check(2,
          ↪  c, 3, d))
          {
            continue;
          }
          std::string field[4] = {"....", "....", "....", "...."};
          field[a][0] = 'Q';
          field[b][1] = 'Q';
          field[c][2] = 'Q';
          field[d][3] = 'Q';
          for (int i = 0 ; i < 4 ; ++i)
          {
            std::cout << field[i] << "\n";
          }
          std::cout << "\n";
```

```
        }
      }
    }
  }
}
```

**Листинг 8.2. 8queensLoop.py**

```
#!/usr/bin/python3

def check(x1, y1, x2, y2):
  if y1 == y2:
    return False
  if x2 - x1 == y2 - y1 or x2 - x1 == y1 - y2:
    return False
  return True

for a in range(4):
  for b in range(4):
    if not check(0, a, 1, b):
      continue
    for c in range(4):
      if not check(0, a, 2, c) or not check(1, b, 2, c):
        continue
      for d in range(4):
        if not check(0, a, 3, d) or not check(1, b, 3, d) or not
        ↪  check(2, c, 3, d):
          continue
        field = [['.'] * 4, ['.'] * 4, ['.'] * 4, ['.'] * 4]
        field[a][0] = 'Q';
        field[b][1] = 'Q';
        field[c][2] = 'Q';
        field[d][3] = 'Q';
        for i in range(4):
          print(''.join(field[i]))
        print()
```

**Листинг 8.3. 8queens.cpp**

```
#include <iostream>
#include <string>
```

```
const int N = 8;

int queens[N];

bool check(int x1, int y1, int x2, int y2)
{
  if (y1 == y2)
  {
    return false;
  }
  if (x2 - x1 == y2 - y1 || x2 - x1 == y1 - y2)
  {
    return false;
  }
  return true;
}

void find(int q)
{
  if (q == N)
  {
    for (int i = 0 ; i < N ; ++i)
    {
      std::string line(N, '.');
      line[queens[i]] = 'Q';
      std::cout << line << '\n';
    }
    std::cout << '\n';
    return;
  }
  for (int i = 0 ; i < N ; ++i)
  {
    bool ok = true;
    for (int k = 0 ; ok && k < q ; ++k)
    {
      ok = check(k, queens[k], q, i);
    }
    if (ok)
    {
      queens[q] = i;
      find(q + 1);
```

```
    }
  }
}

int main()
{
  find(0);
}
```

**Листинг 8.4. 8queens.py**

```
#!/usr/bin/python3

N = 8
queens = [0] * N

def check(x1, y1, x2, y2):
  if y1 == y2:
    return False
  if x2 - x1 == y2 - y1 or x2 - x1 == y1 - y2:
    return False
  return True

def find(q):
  if q == N:
    for i in range(N):
      pos = queens[i]
      line = '.' * pos + 'Q' + '.' * (N - pos - 1)
      print(line)
    print()
    return
  for i in range(N):
    ok = True
    for k in range(q):
      ok = check(k, queens[k], q, i)
      if not ok:
        break
    if ok:
      queens[q] = i
      find(q + 1)
find(0)
```

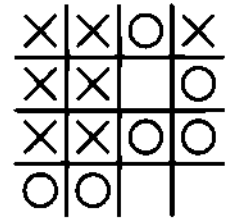
## 8.2. Игра «Крестики-нолики» 4×4

Обычные крестики-нолики мало кому интересны. Там легко разобраться, как играть, чтобы выиграть у слабого соперника или не проиграть сильному.

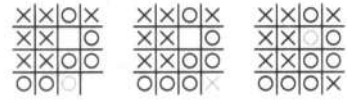
Чтобы игра стала сложнее, можно не только увеличить размер поля, но и изменить правила. Одним из вариантов таких правил мы сейчас и займёмся. В нём игроки ставят крестики и нолики как обычно. «Крестики» выигрывают, если образовалась линия из четырёх одинаковых символов. Причём неважно, из крестиков эта линия состоит или из ноликов.

Единственное ограничение, необходимое для баланса игры, это то, что линия, построенная на второстепенной диагонали, не считается. То есть учитываются только вертикали, горизонтали и главная диагональ (слева направо и сверху вниз).

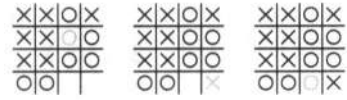
Чтобы лучше понять правила, рассмотрим пример позиции. Сейчас очередь «ноликов» ходить:



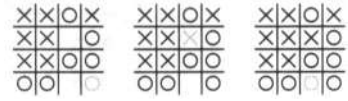
Вроде бы тут напрашивается линия из четырёх ноликов. Но в этой игре любая линия — это поражение для ноликов, а не выигрыш, как в обычном варианте игры. И они обязательно построят линию, куда бы ни сделали ход. Хоть сюда:



Или сюда:



Или даже так:



Выходит, что игра не так проста, как обычные крестики-нолики, которые мы уже разбирали в одной из предыдущих глав. Тут даже сходу не просматривается стратегия, которую можно было бы реализовать в виде несложных условий.

В то же время число состояний игры ограничено сверху значением $3^{16} = 43046721$ На самом деле их меньше, потому что не все комбинации крестиков, ноликов и пустых клеток могут возникнуть в реальной партии (например, на поле не могут быть одни нолики без крестиков).

Такое небольшое число состояний позволяет нам провести полный перебор. Можно проверить все состояния заранее (и построить таблицу выигрышных ходов), но лучше потренироваться в написании рекурсивных алгоритмов. Они ещё не раз понадобятся нам в дальнейшем.

**Экран 1. Фрагмент партии в усложнённые крестики-нолики**

```
Мой ход:
  a b c d
1 O . . .
2 . X . .
3 . . . .
4 . . . .

Куда будете ходить? c2
  a b c d
1 O . . .
2 . X X .
3 . . . .
4 . . . .

Мой ход:
  a b c d
1 O . . .
2 . X X O
3 . . . .
4 . . . .

Куда будете ходить? c3
  a b c d
1 O . . .
2 . X X O
3 . . X .
4 . . . .
```

```
Мой ход:
  a b c d
1 O . . .
2 . X X O
3 . O X .
4 . . . .

Куда будете ходить? c1
  a b c d
1 O . X .
2 . X X O
3 . O X .
4 . . . .

Мой ход:
  a b c d
1 O . X .
2 . X X O
3 . O X .
4 . . O .

Куда будете ходить? b1
  a b c d
1 O X X .
2 . X X O
3 . O X .
4 . . O .
```

### 8.2.1. Игровая позиция

Игровую позицию в крестиках-ноликах мы уже разбирали для поля 3 × 3. Она кодировалась с помощью массива 3 × 3. Одна ячейка массива — одна клетка поля.

То же самое можно сделать и сейчас, только массив будет 4 × 4. По ходу игры в ячейки будут записываться символы `'X'` или `'O'`.

C++

```
const int N = 4;
char field[N][N];
```

Python

```
N = 4
field = [[None] * N for _ in range(N)]
```

### 8.2.2. Главный игровой цикл

Ход игры не должен сильно отличаться от прежней версии: выводим поле, один игрок делает ход. Потом снова выводим поле, и другой игрок делает ход.

Если просто адаптировать прежний игровой цикл под новую размерность, получится так:

C++

```
#include <iostream>

int main()
{
  int moves = 0;
  while (true)
  {
    printField();
    if (checkWin())
    {
      std::cout << "Я выиграл!\n";
      break;
    }
    playerMove();
    ++moves;
    printField();
    if (checkWin())
    {
      std::cout << "Вы победили!\n";
      break;
    }
    if (moves == N * N)
    {
      std::cout << "Ничья!\n";
      break;
    }
    std::cout << "Мой ход:\n";
    computerMove();
    ++moves;
  }
}
```

Python

```
moves = 0
while True:
  printField()
  if checkWin():
    print('Я выиграл!')
    break
  playerMove()
  moves += 1
  printField()
  if checkWin():
    print('Вы победили!')
    break
  if moves == N * N:
    print('Ничья!')
    break
  print('Мой ход:')
  computerMove()
  moves += 1
```

Сразу видны несколько несоответствий. Во-первых, в новом варианте на поле чётное число клеток. Поэтому игра закончится после хода второго игрока, а не первого. А значит, и заполненность поля нужно проверять в другой фазе игрового цикла. Например, в самом начале. Это произойдёт перед ходом первого игрока, то есть в самом начале игры (вхолостую), а потом после каждого хода «ноликов».

И сразу же поменяем сообщение с ничьей на победу компьютера. Он у нас играет за ноликов, а по новым правилам они побеждают, если никакой линии не образовалось.

C++

```
while (true)
{
  printField();
  if (moves == N * N)
  {
    std::cout << "Я победил!\n";
    break;
  }
  ...
  computerMove();
  ++moves;
}
```

Python

```
while True:
  printField()
  if moves == N * N:
    print('Я победил!')
    break
  ...
  computerMove()
  moves += 1
```

Но нужно проверять и наличие выстроенных линий на поле. Ведь если хотя бы одна появилась, то крестики выиграли.

C++

```
while (true)
{
  printField();
  if (checkLines())
  {
    std::cout << "Вы выиграли!\n";
    break;
  }
  if (moves == N * N)
  {
    std::cout << "Я победил!\n";
    break;
  }
  playerMove();
  ++moves;
  printField();
  if (checkLines())
  {
    std::cout << "Вы выиграли!\n";
    break;
  }
  std::cout << "Мой ход:\n";
  computerMove();
  ++moves;
}
```

Python

```
moves = 0
while True:
  printField()
  if checkLines():
    print('Вы выиграли!')
    break
  if moves == N * N:
    print('Я победил!')
    break
  playerMove()
  moves += 1
  printField()
  if checkLines():
    print('Вы выиграли!')
    break
  print('Мой ход:')
  computerMove()
  moves += 1
```

Мы заменили проверки с вызовом `checkWin` на вызовы `checkLines` — новой функции, возвращающей `true`, если на поле появилась линия из четырёх одинаковых символов. В этом случае выводится сообщение о победе «крестиков».

Вот и готовая программа. Всего-то и осталось, что написать функции `printField` (вывод игрового поля), `checkLines` (проверка линий из одинаковых символов), `playerMove` (ход игрока) и `computerMove` (ход компьютера).

### 8.2.3. Вывод игрового поля

В прежних крестиках-ноликах клетки нумеровались от 1 до 9. Игрок при выборе хода просто вводил нужное число. Сейчас же клеток 16, и если выводить такие же номера в пустых клетках, то будет не очень красиво, из-за того что некоторые из них двузначные.

Вместо этого можно применить тот же способ, что был в игре «Реверси» — столбцы обозначаются буквами, а строки цифрами. Тогда игрок будет при выборе клетки для хода вводить два символа – букву и цифру.

Но пока дело до ввода не дошло, обозначения строк и столбцов нужно уже выводить в функции `printField`, чтобы потом к ней не возвращаться.

C++

```
void printField()
{
  std::cout << ' ';
  for (int j = 0; j < N; ++j)
  {
    std::cout << ' ' << (char)('a' + j);
  }
  std::cout << '\n';
  for (int i = 0 ; i < N ; ++i)
  {
    std::cout << i + 1;
    for (int j = 0 ; j < N ; ++j)
    {
      if (field[i][j])
      {
        std::cout << ' ' << field[i][j];
      }
      else
      {
        std::cout << " .";
      }
    }
    std::cout << '\n';
  }
  std::cout << '\n';
}
```

Python

```
def printField():
  print(' ', end='')
  for j in range(N):
    print(f' {chr(ord("a") + j)}',
    ↪ end='')
  print()
  for i in range(N):
    print(f'{i + 1}', end='')
    for j in range(N):
      if field[i][j] is not None:
        print(f' {field[i][j]}',
        ↪ end='')
      else:
        print(' .', end='')
      print()
  print()
```

### 8.2.4. Ход игрока

За обработку хода игрока отвечает функция **playerMove**. Она уже не такая, как была в старых крестика-ноликах, из-за того что теперь выбираемая клетка кодируется двумя символами, а не одним.

Но зато она почти не отличается от аналога в игре «Реверси». Разница лишь в том, что здесь не нужно переворачивать фишки. Просто символ *'X'* записывается в нужную ячейку массива.

**C++**

```
void playerMove()
{
  int x, y;
  do
  {
    std::cout << "Куда будете ходить? ";
    char c;
    std::cin >> c;
    x = c - 'a';
    std::cin >> c;
    y = c - '1';
  }
  while (x < 0 || x >= N || y < 0 || y >=
  ↪  N || field[y][x]);
  field[y][x] = 'X';
}
```

**Python**

```
def playerMove():
  x = -1
  y = -1
  while x < 0 or x >= N or y < 0 or y >=
  ↪  N or field[y][x] is not None:
    s = input('Куда будете ходить? ')
    x = ord(s[0]) - ord('a')
    y = int(s[1]) - 1
  field[y][x] = 'X'
```

### 8.2.5. Случайный ход компьютера

Пока искусственный интеллект для игры придумывать рано, ведь нужно протестировать остальные функции. А для этого лучше всего подойдёт выбор случайной клетки для хода компьютера.

Тогда программа использует это, чтобы можно было проверить почти всё.

**C++**

```
void computerMove()
{
  int x, y;
  do
  {
    y = rand() % N;
    x = rand() % N;
  }
  while (field[y][x]);
  field[y][x] = 'O';
}
```

**Python**

```
def computerMove():
  while True:
    x = random.randint(0, N - 1)
    y = random.randint(0, N - 1)
    if field[y][x] is None:
      break
  field[y][x] = 'O'
```



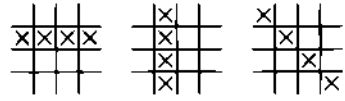
Напишите функцию-заглушку **checkLines**, возвращающую **false**. С ней программа уже компилируется, и можно «поиграть», чтобы посмотреть, как работают ввод и вывод.

### 8.2.6. Определение победителя

Победа компьютера (ноликов) уже определяется по числу ходов. Если все клетки заняты, а победа «крестиков» не наступила, то выиграли «нолики».

За проверку признака победы «крестиков» отвечает функция `checkLines`. Если она находит на поле линию из одинаковых символов, то возвращает `true`. Это означает, что «крестики» победили.

Проверять нужно горизонтали, вертикали и главную диагональ:



Чтобы проверить одну линию, нужно убедиться, что значения всех клеток в ней одинаковые. Но при этом они не должны быть пустыми, поэтому перед началом цикла со сравнениями всех клеток есть дополнительное условие.

C++

```
bool found = field[0][0] != 0;
for (int j = 1 ; found && j < N ; ++j)
{
  found = field[j][j] == field[0][0];
}
if (found)
{
  return true;
}
```

В условие продолжения цикла мы вставили проверку того, что переменная **found** продолжает оставаться истинной. Ведь зачем проверять линию дальше, если уже встретились неподходящие символы.

Python

```
found = field[0][0] is not None
for j in range(N):
  found = found and field[j][j] ==
  ↪ field[0][0]
if found:
  return True
```

Можно было прервать цикл **for** с помощью оператора **break**. Код был бы немного длиннее, но зато более эффективным. Вместо этого мы здесь «накапливаем» результат в переменной **found** с помощью логического оператора **and**.

Переменная `found` инициализируется значением `true`, если первая клетка не пустая. Потом все остальные клетки строки (или столбца, или диагонали) мы обходим в цикле. И если символ в одной из них не совпадает с символом в первой, `found` принимает значение `false`, а цикл завершается.

Поэтому если `found` остаётся «истиной», то линия найдена и можно возвращать `true`.

Фрагмент выше проверяет главную диагональ. Второстепенную диагональ в этом варианте игры проверять не нужно. Но ещё надо добавить такие же циклы для проверки вертикалей и горизонталей. Так как их много, эти проверки нужно делать ещё в одном цикле. Он перебирает сами строки и столбцы, а циклы внутри уже проверяют отдельные клетки.

C++

```
for (int i = 0 ; i < N ; ++i)
{
  found = field[i][0] != 0;
  for (int j = 1 ; found && j < N ; ++j)
  {
    found = field[i][j] == field[i][0];
  }
  if (found)
  {
    return true;
  }
  found = field[0][i] != 0;
  for (int j = 1 ; found && j < N ; ++j)
  {
    found = field[j][i] == field[0][i];
  }
  if (found)
  {
    return true;
  }
}
```

Python

```
for i in range(N):
  found = field[i][0] is not None
  for j in range(N):
    found = found and field[i][j] ==
    ↪ field[i][0]
  if found:
    return True
  found = field[0][i] is not None
  for j in range(N):
    found = found and field[j][i] ==
    ↪ field[0][i]
  if found:
    return True
```

Соберите все эти фрагменты вместе, чтобы получилась полнофункциональная версия `checkLines`.

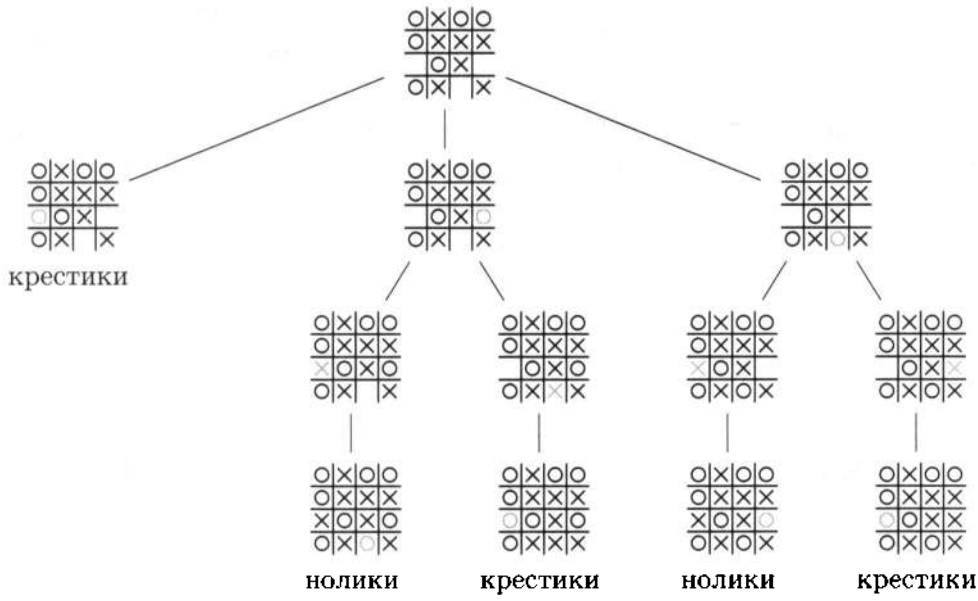
### 8.2.7. Выбор наилучшего хода

Выбор наилучшего хода — вопрос непростой. Если есть выбор, надо как-то оценить, какая из клеток выгоднее. В игре «Реверси», например, мы использовали эвристику, исходящую из положения фишек на доске. Варианты ответа противника там не рассматривались.

Но эвристики часто ошибаются именно по этой причине. Поэтому нужно анализировать не только позицию саму по себе, но и возможные последовательности ходов, которые могут приводить или к победе, или к поражению.

Для этого хорошо подходят рекурсивные алгоритмы. Функция анализа хода ставит крестик или нолик на поле, а потом вызывает саму себя. Этот новый вызов тоже ставит крестик или нолик, и так далее. Когда вызов завершится, функция анализа ставит крестик или нолик на другое поле. Так просматриваются все возможные варианты ходов, то есть делается их полный перебор.

Пока ничего непонятно, поэтому обратимся к рисунку. На нём приведено дерево всех возможных ходов, начиная с некоторой позиции, возникшей в процессе игры. Чтобы перебрать эти позиции, наша рекурсивная функция должна находить пустые клетки, делать в них ход и вызывать саму себя:




После возвращения из вызова функции нужно очищать выбранную клетку и искать следующую. Это может выглядеть примерно так:

**C++**

```
void find(bool player)
{
  for (int x = 0 ; x < N ; ++x)
  {
    for (int y = 0 ; y < N ; ++y)
    {
      if (!field[y][x])
      {
        field[y][x] = player ? 'X' : '0';
        find(!player);
        field[y][x] = 0;
      }
    }
  }
}
```

**Python**

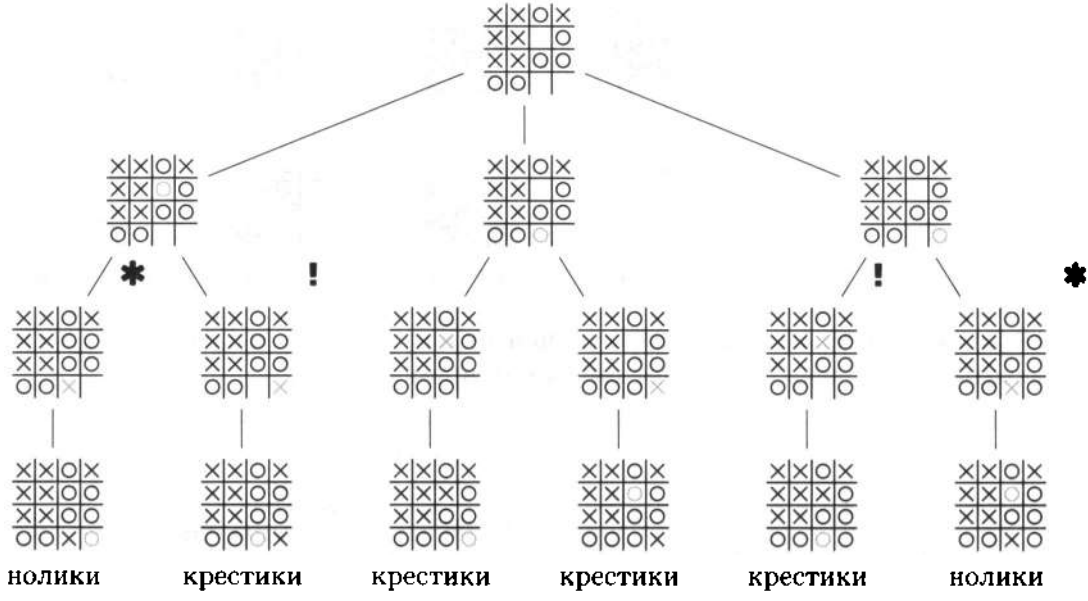
```
def find(player):
  for x in range(N):
    for y in range(N):
      if field[y][x] is None:
        field[y][x] = 'X' if player else
        ↪ '0'
        canWin(not player)
        field[y][x] = None
```

Единственный параметр функции — булева переменная `player`. Она принимает значение `true`, если анализируется ход игрока-человека (то есть «крестиков»). Когда функция вызывает саму себя, эта переменная меняет значение на противоположное, потому что ходы «крестиков» и «ноликов» чередуются.

Иногда ещё одним параметром бывает игровая позиция, но здесь это не нужно, потому что легко модифицировать и восстанавливать единственное хранилище для игрового поля — массив `field`.

Теперь нужно разобраться, как подобная функция сможет понять, кто победит для заданной позиции. Мы программируем игру за «ноликов», поэтому функция будет определять, могут ли выиграть «нолики».

Это похоже на анализ игры «23 спички», который мы делали раньше. Для каждой позиции можно определить, приводит она к победе или к проигрышу, если игроки будут играть оптимально. То есть не делать заведомо проигрышных ходов. Рассмотрим ещё одно дерево игры.




Вроде бы тут есть окончания игры, когда «нолики» побеждают. Но будут ли «крестики» делать ходы, отмеченные звёздочкой ✱? При анализе игр полным перебором всегда предполагается, что противник играет оптимально и не делает ходов, ведущих к поражению. А звёздочкой помечены именно такие ходы.

Поэтому «крестики» будут выбирать те ходы, что отмечены восклицательным знаком !. И получается, что любой ход «ноликов» в этом дереве игры ведёт к их поражению.

Переименуем нашу функцию в `canWin` (она же проверяет, могут ли «нолики» победить) и добавим проверку для реализации логики «крестиков».

Смысл проверки такой: если функция анализирует ход «крестиков», и при этом они могут сходить так, что «нолики» проиграют, то так и будет для всего анализируемого дерева ходов. «Нолики» проигрывают, а функция возвращает `false`.

C++

```
bool canWin(bool player)
{
  for (int x = 0 ; x < N ; ++x)
  {
    for (int y = 0 ; y < N ; ++y)
    {
      if (!field[y][x])
      {
        field[y][x] = player ? 'X' : '0';
        bool w = canWin(!player);
        field[y][x] = 0;
        if (player && !w)
        {
          return false;
        }
      }
    }
  }
}
```

Python

```
def canWin(player):
  for x in range(N):
    for y in range(N):
      if field[y][x] is None:
        field[y][x] = 'X' if player else
        ↪ '0'
        w = canWin(not player)
        field[y][x] = None
        if player and not w:
          return False
  return player
```

Также нужно добавить и обратное условие. Если анализируется позиция для «ноликов» и они каким-то своим ходом могут победить, то эта позиция будет для них выигрышной.

C++

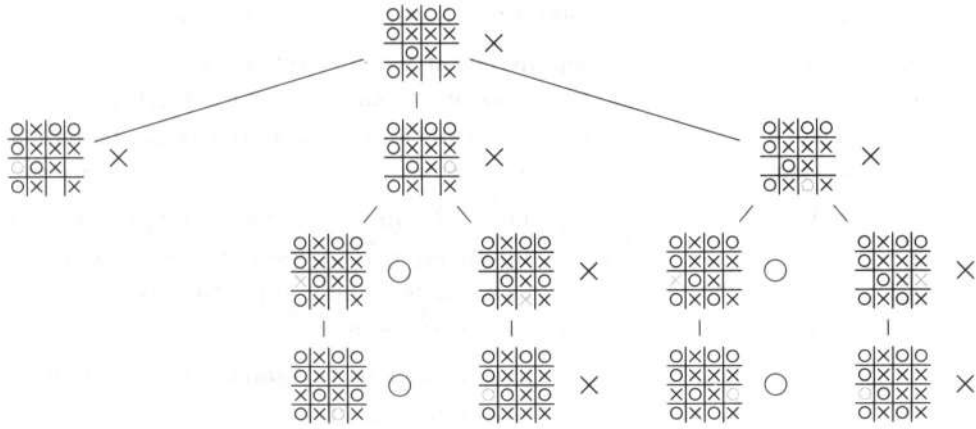
```
if (!player && w)
{
  return true;
}
```

Python

```
if not player and w:
  return True
```

То есть цель нашей функции `canWin` — определить для заданной позиции, является ли она выигрышной для «ноликов». Например, если вернуться к самой первой картинке, для каждой игровой позиции можно пометить, кто в ней будет побеждать при оптимальной игре обоих.

Рядом с позициями нарисуем крестики или нолики — признак победителя:



Этот признак и должна возвращать функция `canWin`. То есть «истину», если в такой позиции побеждают «нолики», и «ложь», если крестики.

### 8.2.8. Условие завершения рекурсии

Вы не могли не заметить, что функция `canWin` возвращает «истину» или «ложь», только если какой-то другой вызов этой функции что-то вернул. А где же тот самый первый вызов, что вернёт какое-то значение независимо от рекурсии?

Первое, что стоит учесть, это момент остановки игры при появлении линии на поле. Она может возникнуть после хода любого из игроков, но побеждают при этом «крестики», то есть возвращать нужно значение `false`.

**C++**

```
bool canWin(bool player)
{
  if (checkLines())
  {
    return false;
  }
  ...
}
```

**Python**

```
def canWin(player):
  if checkLines():
    return False
  ...
```

А что будет происходить, если пустых клеток на поле больше нет? Значит, побеждают «нолики», нужно вернуть `true`.

**C++**

```
bool canWin(bool player)
{
  ...
  return true;
}
```

**Python**

```
def canWin(player):
  ...
  return True
```

Всегда ли нужно возвращать здесь «истину»? Если пустых клеток нет, цикл с рекурсией ничего не сделает, функция не завершится, поэтому всё правильно.

Но ведь функция может не завершиться и в том случае, если пустые клетки есть. Например, если очередь «крестиков», а выигрышного хода у них не нашлось. Тогда логично тоже вернуть «истину» (раз «крестики» не выигрывают, то побеждают «нолики»).

А если анализируются ходы «ноликов»? Если у них нет выигрышного хода, то надо вернуть «ложь», они точно проиграют. Получается, что в конце нужно вернуть `true`, если поле заполнено, также `true`, если у «крестиков» нет хорошего хода, и `false`, если хорошего хода нет у «ноликов».

Эту логику можно немного упростить. Когда поле заполнено, последний ход сделали «нолики». То есть очередь ходить — у «крестиков». И хода для них не будет, потому что все клетки заняты. Значит, это тот же случай, когда у

«крестиков» нет хода. И отдельно его рассматривать не нужно. Следовательно, последний оператор `return` будет выглядеть так:

C++

```
bool canWin(bool player)
{
  ...
  return player;
}
```

Python

```
def canWin(player):
  ...
  return player
```

### 8.2.9. Встраиваем анализ ходов в игру

Функция `computerMove` сейчас делает только случайные ходы. А мы бы хотели, чтобы ходы были осмысленные. Но какой ход осмысленный?

Компьютеру лучше всего ставить нолик так, чтобы получалась выигрышная для него позиция. «Выигрышность» уже умеет определять функция `canWin`. Следовательно, нужно проверить, к чему приведёт каждый из возможных ходов. И если он приводит к победе, то там нолик и оставить.

C++

```
void computerMove()
{
  for (int x = 0 ; x < N ; ++x)
  {
    for (int y = 0 ; y < N ; ++y)
    {
      if (!field[y][x])
      {
        field[y][x] = '0';
        if (canWin(true))
        {
          return;
        }
        field[y][x] = 0;
      }
    }
  }
  ...
}
```

Python

```
def computerMove():
  for x in range(N):
    for y in range(N):
      if field[y][x] is None:
        field[y][x] = '0'
        if canWin(True):
          return
        field[y][x] = None
  ...
```

А если хороших ходов не осталось (вдруг всё-таки в этой игре «крестики» имеют выигрышную стратегию), то в конце функции остаётся выбор случайной клетки. То есть какой-то ход в любом случае будет сделан.

В этой игре крестикам не так-то просто победить, если вообще возможно. Теперь уже можно поиграть и проверить это на практике.

Первые ходы компьютер делает не очень быстро, потому что перебирается много позиций. А вот как это можно оптимизировать, мы ещё разберём в следующих главах.

### 8.2.10. Задания для самостоятельной работы

**1.** Код функции checkLines можно немного упростить. Ведь там похожие фрагменты проверки одной линии повторяются трижды. Придумайте, как вынести их в отдельную функцию.

**2.** Чтобы узнать, могут ли крестики выиграть, если нолики играют идеально, сделайте аналог функции canWin, но для крестиков. Тогда её вызов можно встроить в playerMove и наблюдать, как компьютер играет сам с собой.

**Листинг 8.5. tictactoe4.cpp**

```
#include <iostream>

const int N = 4;
char field[N][N];

void playerMove()
{
    int x, y;
    do
    {
        std::cout << "Куда будете ходить? ";
        char c;
        std::cin >> c;
        x = c - 'a';
        std::cin >> c;
        y = c - '1';
    }
    while (x < 0 || x >= N || y < 0 || y >= N || field[y][x]);
    field[y][x] = 'X';
}

bool checkLines()
{
    bool found;
    for (int i = 0 ; i < N ; ++i)
    {
        found = field[i][0] != 0;
        for (int j = 1 ; found && j < N ; ++j)
        {
```

```
            found = field[i][j] == field[i][0];
        }
        if (found)
        {
            return true;
        }
        found = field[0][i] != 0;
        for (int j = 1 ; found && j < N ; ++j)
        {
            found = field[j][i] == field[0][i];
        }
        if (found)
        {
            return true;
        }
    }
    found = field[0][0] != 0;
    for (int j = 1 ; found && j < N ; ++j)
    {
        found = field[j][j] == field[0][0];
    }
    return found;
}

bool canWin(bool player)
{
    if (checkLines())
    {
        return false;
    }
    for (int x = 0 ; x < N ; ++x)
    {
        for (int y = 0 ; y < N ; ++y)
        {
            if (!field[y][x])
            {
                field[y][x] = player ? 'X' : '0';
                bool w = canWin(!player);
                field[y][x] = 0;
                if (!player && w)
                {
```

```
                    return true;
                }
                if (player && !w)
                {
                    return false;
                }
            }
        }
    }
    return player;
}

void computerMove()
{
    for (int x = 0 ; x < N ; ++x)
    {
        for (int y = 0 ; y < N ; ++y)
        {
            if (!field[y][x])
            {
                field[y][x] = 'O';
                if (canWin(true))
                {
                    return;
                }
                field[y][x] = 0;
            }
        }
    }
    int x, y;
    do
    {
        y = rand() % N;
        x = rand() % N;
    }
    while (field[y][x]);
    field[y][x] = 'O';
}

void printField()
{
```

```
    std::cout << ' ';
    for (int j = 0 ; j < N ; ++j)
    {
        std::cout << ' ' << (char)('a' + j);
    }
    std::cout << '\n';
    for (int i = 0 ; i < N ; ++i)
    {
        std::cout << i + 1;
        for (int j = 0 ; j < N ; ++j)
        {
            if (field[i][j])
            {
                std::cout << ' ' << field[i][j];
            }
            else
            {
                std::cout << " .";
            }
        }
        std::cout << '\n';
    }
    std::cout << '\n';
}

int main()
{
    srand((int)time(NULL));

    int moves = 0;
    while (true)
    {
        printField();
        if (checkLines())
        {
            std::cout << "Вы выиграли! \n";
            break;
        }
        if (moves == N * N)
        {
            std::cout << "Я победил! \n";
```

```
            break;
        }
        playerMove();
        ++moves;
        printField();
        if (checkLines())
        {
            std::cout << "Вы выиграли! \n";
            break;
        }
        std::cout << "Мой ход: \n";
        computerMove();
        ++moves;
    }
}
```

**Листинг 8.6. tictactoe4.py**

```
#!/usr/bin/python3
import random

N = 4
field = [[None] * N for _ in range(N)]

def playerMove():
    x = -1
    y = -1
    while x < 0 or x >= N or y < 0 or y >= N or field[y][x] is not
    ↪ None:
        s = input('Куда будете ходить? ')
        x = ord(s[0]) - ord('a')
        y = int(s[1]) - 1
    field[y][x] = 'X'

def checkLines():
    for i in range(N):
        found = field[i][0] is not None
        for j in range(N):
            found = found and field[i][j] == field[i][0]
        if found:
            return True
```

```
            found = field[0][i] is not None
            for j in range(N):
                found = found and field[j][i] == field[0][i]
            if found:
                return True
        found = field[0][0] is not None
        for j in range(N):
            found = found and field[j][j] == field[0][0]
        return found

def canWin(player):
    if checkLines():
        return False
    for x in range(N):
        for y in range(N):
            if field[y][x] is None:
                field[y][x] = 'X' if player else '0'
                w = canWin(not player)
                field[y][x] = None
                if not player and w:
                    return True
                if player and not w:
                    return False
    return player

def computerMove():
    for x in range(N):
        for y in range(N):
            if field[y][x] is None:
                field[y][x] = '0'
                if canWin(True):
                    return
                field[y][x] = None
    while True:
        x = random.randint(0, N - 1)
        y = random.randint(0, N - 1)
        if field[y][x] is None:
            break
    field[y][x] = '0'

def printField():
```

```
    print(' ', end='')
    for j in range(N):
        print(f' {chr(ord("a") + j)}', end='')
    print()
    for i in range(N):
        print(f'{i + 1}', end='')
        for j in range(N):
            if field[i][j] is not None:
                print(f' {field[i][j]}', end='')
            else:
                print(' .', end='')
        print()
    print()

moves = 0
while True:
    printField()
    if checkLines():
        print('Вы выиграли!')
        break
    if moves == N * N:
        print('Я победил!')
        break
    playerMove()
    moves += 1
    printField()
    if checkLines():
        print('Вы выиграли!')
        break
    print('Мой ход:')
    computerMove()
    moves += 1
```

Глава 9

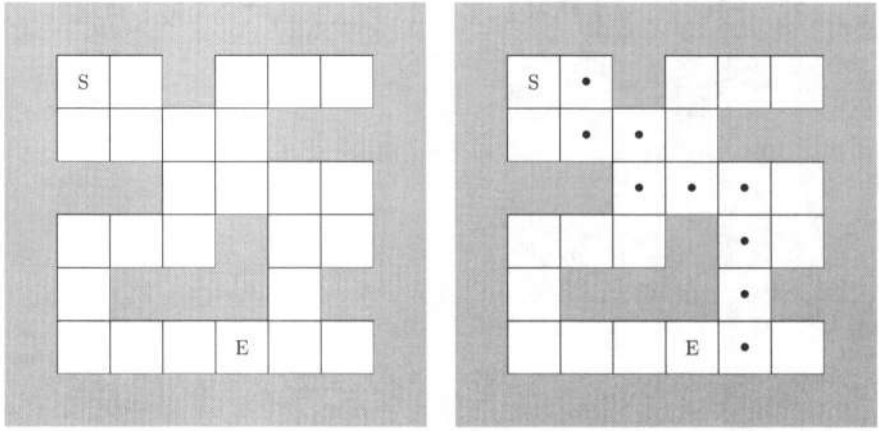
# Когда рекурсия не подходит

## 9.1. Головоломка «Лабиринт»

Лабиринты встречаются во многих играх. Иногда это отдельное препятствие для игрока, а иногда даже весь игровой мир можно представить как лабиринт.

Но не только игроки сталкиваются с лабиринтами. Например, когда компьютер передвигает войска, автомобили или даже шарики по команде игрока, он делает это по кратчайшему пути. А поиск кратчайшего пути почти не отличается от поиска пути в лабиринте.

Мы будем искать кратчайший маршрут в лабиринте такого вида:




В нём есть клетки нескольких типов: пустые клетки, через которые можно проходить, стены (закрашены серым), стартовая клетка (с буквой S) и финишная клетка (с буквой E). Картинка слева — это задание для поиска пути, а на картинке справа кратчайший маршрут в этом же лабиринте обозначен точками.

Задача программы — найти кратчайший путь, если он существует, и вывести его. То есть вывести правую картинку, если на вход поступает левая.

При работе с программой стены будут обозначаться символами `'#'`, пустые клетки — точками, а кратчайший путь — символами `'*'`.

### 9.1.1. Игровая позиция

Как такового, игрового процесса тут нет. Карты не раздаются, фишки не перемещаются. Но лабиринт всё равно нужно где-то хранить.

Лабиринт — это прямоугольник из символов. Для его представления в программе отлично подойдёт массив строк. Первая координата, $y$, отвечает за номер строки, то есть элемент самого массива. А вторая, $x$, отвечает за номер столбца, то есть символ в строке. Можно было бы и двумерный массив символов

использовать, но тогда усложнится считывание его с клавиатуры. А сама работа с содержимым в любом случае будет выглядеть одинаково.

C++

```
#include <vector>
#include <string>

std::vector<std::string> maze;
```

Python

```
maze = []
```

**Экран 1. Компьютер ищет кратчайший путь в лабиринте**

```
Введите лабиринт со стартовой
↪ клеткой S и финишной E.
########
#S.#...#
#....###
###....#
#...#..#
#.###.##
#...E..#
########

Кратчайший путь:
########
#S*#...#
#.***###
###.**.#
#...#*.#
#.###*##
#...E*.#
########
```

```
Введите лабиринт со стартовой
↪ клеткой S и финишной E.
########
#S.#...#
#....###
####...#
#...#..#
#.######
#...E..#
########

Пути между S и E не
↪ существует.
```

### 9.1.2. Структура программы

Так как это головоломка, которую решает компьютер, то игрового цикла здесь нет. Вместо этого программа должна ввести лабиринт, затем найти кратчайший путь, а в конце вывести этот путь на экран.

Эти задачи можно распределить по трём функциям:

C++

```
int main()
{
  inputMaze();
  findPath();
  outputPath();
}
```

Python

```
inputMaze()
findPath()
outputPath()
```

Всегда полезно начинать с «работающей» программы (даже если она ничего не делает), а затем постепенно наполнять её.
Создайте заготовки для функций `inputMaze`, `findPath` и `outputPath`. Дальше мы наполним их содержанием.

### 9.1.3. Ввод лабиринта

Пользователю нашей программы (или игроку?) будет удобнее не вводить отдельно размеры лабиринта, а сразу набирать все строки, из которых он состоит.

Поэтому использовать цикл `for` не получится, ведь неизвестно, сколько будет вводится строк. Вместо этого цикл будет работать до тех пор, пока не введена пустая строка. Это и будет окончанием пользовательского ввода.

C++

```
void inputMaze()
{
  while (true)
  {
    std::string s;
    std::getline(std::cin, s);
    if (s.empty())
    {
      break;
    }
    maze.push_back(s);
  }
}
```

Python

```
def inputMaze():
  while True:
    s = input()
    if len(s) == 0:
      break
    maze.append(list(s))
```

В массиве `maze` хранятся просто строки символов. Каждая строка — это одномерный массив, соответствующий одной горизонтали лабиринта.

Чтобы записывать найденный путь обратно в клетки лабиринта, его нужно модифицировать. А строки в Python модифицировать нельзя. Поэтому приходится преобразовывать строки в массивы из символов с помощью конструкции `list(s)`.

### 9.1.4. Вывод лабиринта

Пока кратчайший маршрут не находится, можно просто выводить сам лабиринт. Заодно можно будет и проверить, правильно ли работает ввод.

Тут просто перебираются и выводятся строки основного массива.

C++

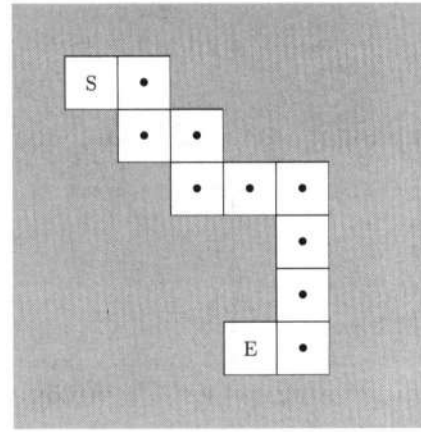
```
void outputPath()
{
  for (auto s : maze)
  {
    std::cout << s << '\n';
  }
}
```

Python

```
def outputPath():
  for s in maze:
    print("".join(s))
```

### 9.1.5. Поиск кратчайшего пути

Очень легко найти путь в лабиринте, если по пути нет никаких развилок. Например, так:




Это можно сделать, просто перебирая каждый раз соседние клетки. Начинается путь со стартовой клетки, значит, её нужно найти во введённом лабиринте. Для этого можно написать функцию `findChar`:

C++

```
Pos findChar(char c)
{
  for (int y = 0 ; y < maze.size() ; ++y)
  {
    for (int x = 0 ; x < maze[y].size() ;
    ↪  ++x)
    {
      if (maze[y][x] == c)
      {
        return {x, y};
      }
    }
  }
  return {-1, -1};
}
```

Python

```
def findChar(c):
  for y in range(len(maze)):
    for x in range(len(maze[y])):
      if maze[y][x] == c:
        return Pos(x, y)
  return Pos(-1, -1)
```

Возвращает она пару координат, поэтому придётся ещё определить новый тип данных Pos, где эти координаты будут храниться:

C++

```
struct Pos
{
  int x, y;
};
```

Python

```
@dataclass
class Pos:
    x: int
    y: int
```

А дальше напишем код для обхода пути: проверяем всех соседей клетки, потом переходим в следующую, а потом повторяем это всё.

C++

```
Pos dir[4] = {{-1, 0}, {1, 0}, {0, -1},
↪  {0, 1}};

Pos cur = findChar('S');
while (maze[cur.y][cur.x] != 'E')
{
  for (auto d : dir)
  {
    int x = cur.x + d.x;
    int y = cur.y + d.y;
    if (maze[y][x] != '#' && maze[y][x]
    ↪  != '*')
    {
      maze[y][x] = '*';
      cur = {x, y};
      break;
    }
  }
}
```

Python

```
dir = [Pos(-1, 0), Pos(1, 0), Pos(0, -1),
↪  Pos(0, 1)]

Pos cur = findChar('S')
while maze[cur.y][cur.x] != 'E':
  for d in dir
    x = cur.x + d.x
    y = cur.y + d.y
    if maze[y][x] != '#' and maze[y][x]
    ↪  != '*':
      maze[y][x] = '*'
      cur = Pos(x, y)
      break
```

В посещённые клетки записывается символ «*», чтобы при выводе поля можно было увидеть найденный путь.

При этом есть проверка на наличие звёздочек, чтобы повторно в эти клетки не заходить.

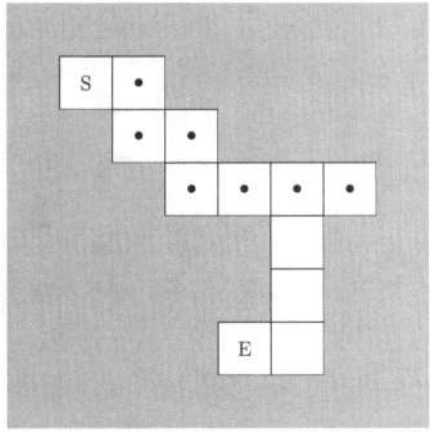
Подайте на вход программе лабиринт с единственным маршрутом. Сработает ли она? А что произойдёт, если возможных маршрутов несколько?

### 9.1.6. Путь с развилками

Наш первый вариант программы иногда мог находить путь нормально, как в предыдущем примере. Но если в лабиринте больше одного маршрута, то программа могла зайти в тупик.

При этом она даже не сможет завершиться, а просто зависнет, потому что проверку отсутствия пути мы не добавили.

Например, это произойдёт для такого лабиринта:




В предыдущих главах мы разбирали рекурсивные алгоритмы. Что если искать кратчайший маршрут с помощью подобного алгоритма? Чтобы функция проверяла возможные направления, вызывая себя рекурсивно. Так она будет продвигаться вперёд во все стороны поочерёдно, поэтому финальная клетка рано или поздно будет достигнута.

C++

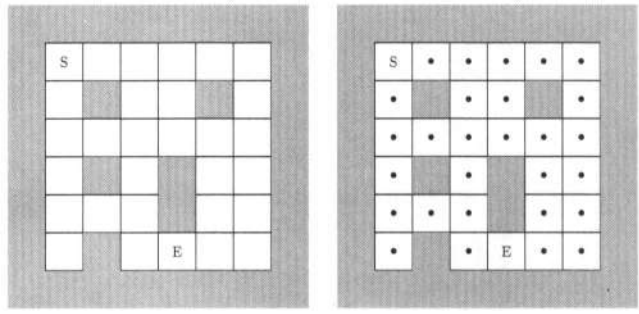
```
Pos dir[4] = {{-1, 0}, {1, 0}, {0, -1},
↪  {0, 1}};

void search(Pos p)
{
  for (auto d : dir)
  {
    int x = p.x + d.x;
    int y = p.y + d.y;
    if (maze[y][x] != '#' && maze[y][x]
    ↪  != '*')
    {
      maze[y][x] = '*';
      search({x, y});
    }
  }
}

void findPath()
{
  ...
  Pos start = findChar('S');
  search(start);
}
```

Python

```
dir = [Pos(-1, 0), Pos(1, 0), Pos(0, -1),
↪  Pos(0, 1)]

def search(p):
  for d in dir:
    x = p.x + d.x
    y = p.y + d.y
    if maze[y][x] != '#' and maze[y][x]
    ↪  != '*':
      maze[y][x] = '*'
      search(Pos(x, y))

def findPath():
  ...
  Pos start = findChar('S')
  search(start)
```

### 9.1.7. Собираем данные для восстановления пути

К сожалению, если просто заполнять звёздочками все направления, то все возможные пути будут помечены и не получится выделить из них только один.




А у нас-то цель найти кратчайший путь. Выходит, что нужно хранить дополнительные данные, чтобы после обхода всех клеток такой путь восстанавливать.

Один из подходов к восстановлению пути — это сохранение для каждой клетки координат, откуда мы в неё пришли. Тогда для конечной клетки можно будет определить предыдущую. Для предыдущей — предпредыдущую, и так далее, пока не вернёмся в стартовую точку.

Чтобы хранить эти данные, потребуется дополнительный массив координат. Для каждой клетки лабиринта будет храниться «предок», то есть координаты следующего шага в обратную сторону, от финиша к старту.

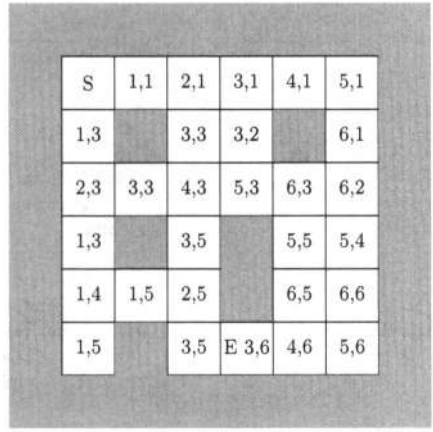
**C++**

```
std::vector<std::vector<Pos>> prev;

void search(Pos p)
{
  ...
    if (prev[y][x].x < 0 && maze[y][x] !=
    ↪  '#')
    {
      prev[y][x] = p;
      search({x, y});
    }
  ...
}

void findPath()
{
  prev.resize(maze.size());
  for (int y = 0 ; y < maze.size() ; ++y)
  {
    prev[y].resize(maze[y].size(), {-1,
    ↪  -1});
  }
  Pos start = findChar('S');
  prev[start.y][start.x] = start;
  ...
}
```

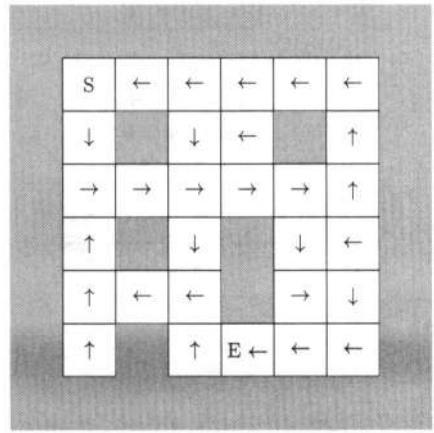
**Python**

```
prev = []

def search(p):
  ...
  if prev[y][x].x < 0 and maze[y][x] !=
  ↪  '#':
    prev[y][x] = p
    search(Pos(x, y))
  ...

def findPath():
  for y in range(len(maze)):
    prev.append([Pos(-1, -1)] *
    ↪  len(maze[y]))
  start = findChar('S')
  prev[start.y][start.x] = start
  ...
```

Содержимое массива `prev` для лабиринта из примера выше будет выглядеть примерно так:




Или, если координаты заменить на стрелочки, показывающие обратное направление (от финиша к старту):




А теперь посмотрим, как это использовать, чтобы восстановить маршрут от старта к финишу.

### 9.1.8. Восстановление пути

Добавим в функцию `outputPath` код для восстановления пути. То есть заполнения звёздочками тех клеток лабиринта, которые ведут от финиша к старту.

Путь восстанавливается именно в обратном направлении, потому что в массиве `prev` хранятся координаты предыдущих, а не следующих шагов. Зато когда маршрут восстановлен целиком, двигаться по нему можно будет и вперёд, и назад.

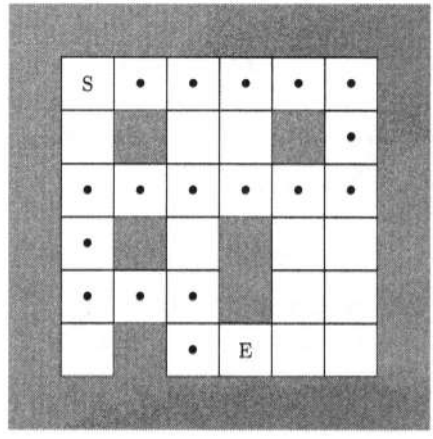
**C++**

```
void outputPath()
{
  Pos end = findChar('E');
  if (end.x < 0 || prev[end.y][end.x].x <
  ↪  0)
  {
    std::cout << "Пути между S и E не
    ↪  существует.\n";
    return;
  }
  while (prev[end.y][end.x].x != end.x
      || prev[end.y][end.x].y != end.y)
  {
    end = prev[end.y][end.x];
    maze[end.y][end.x] = '*';
  }
  maze[end.y][end.x] = 'S';
  ...
}
```

**Python**

```
def outputPath():
  end = findChar('E')
  if end.x < 0 or prev[end.y][end.x].x <
  ↪  0:
    print('Пути между S и E не
    ↪  существует.')
    return

  while prev[end.y][end.x].x != end.x \
    or prev[end.y][end.x].y != end.y:
    end = prev[end.y][end.x]
    maze[end.y][end.x] = '*'
  maze[end.y][end.x] = 'S'
  ...
```

Если теперь запустить программу, получится вот такой маршрут:

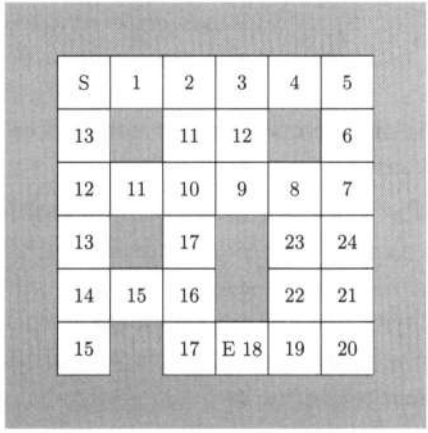



Не очень-то похоже на кратчайший путь. Получается, что в рекурсивном алгоритме поиска есть изъян, и для прохождения лабиринта нужен какой-то другой поиск.

### 9.1.9. Поиск в ширину

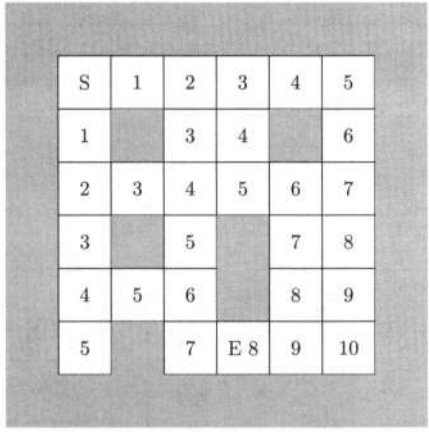
Тот рекурсивный алгоритм, что мы написали, называется поиском в глубину. Его суть в том, что перебираются все достижимые клетки лабиринта (или в общем виде узлы графа), но путь, которым эти клетки достигаются, может быть каким угодно. Хоть все выстроятся в одну длинную цепочку. И рекурсия может погрузиться на глубину всей этой цепочки.

Из-за этого момент посещения какой-либо клетки нельзя использовать как меру расстояния до неё. Вот номера шагов (порядок обхода) для того же самого примера:




Эти расстояния зависят от порядка просмотра всех направлений (то есть содержимого массивов dx и dy). Сейчас в первую очередь просматриваются левый и правый соседи клетки, а можно сделать, чтобы проверялись верхний и нижний. В этом случае расстояния изменятся. Это не то, что бы мы хотели видеть при поиске кратчайшего пути.

А получить нужно примерно следующее. В каждой клетке — кратчайшее расстояние до неё от стартовой:




Видно, что тут получается всё совсем не так, как при обходе в глубину. Будто бы мы делаем сразу много шагов одновременно. Сначала два шага в клетки с 1, потом ещё два в клетки с 2, а потом четыре — в клетки с 3. И так далее.

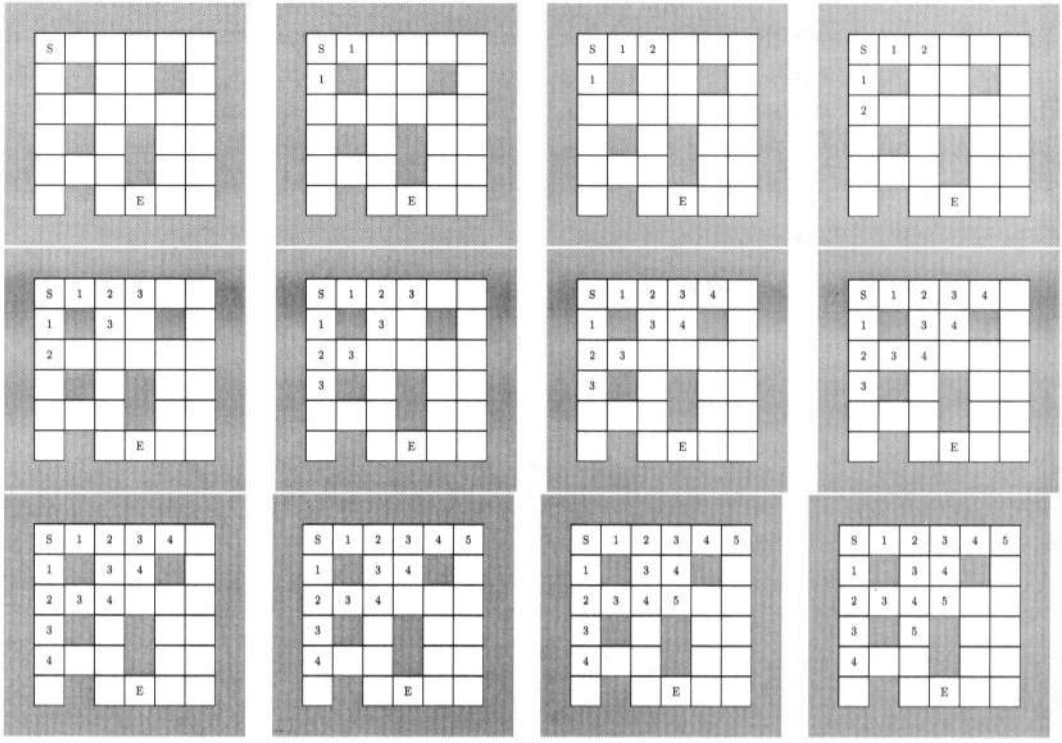
Заполнить лабиринт такими же расстояниями можно по следующему принципу. Начинаем с клетки $S$. Затем находим её соседей и ставим туда 1. Потом смотрим

на все клетки с 1 по очереди, находим их соседей и ставим туда 2. Дальше проделываем то же самое с 2 и получаем клетки с 3. Так продолжается до тех пор, пока весь лабиринт не будет заполнен.

Такой поиск расстояний (и путей) называется обходом в ширину. Тут мы не приступаем к обработке более дальних клеток, пока не будут рассмотрены все ближние. Это можно проделать с помощью структуры данных «очередь». Она работает так же, как обычная очередь в магазине. Кто первый туда попал, тот первым и будет обслуживаться.

Поэтому если добавлять в очередь клетки лабиринта по мере их обнаружения алгоритмом, то сначала туда попадёт стартовая клетка. Потом она будет «обслуживаться», и в очередь попадут клетки с расстоянием 1. Затем из очереди будет извлечена одна из этих клеток и добавлены её соседи с расстоянием 2 (тут он такой один). Потом то же самое произойдёт для второй клетки с расстоянием 1, после чего наступит черёд клеток 2.

Весь лабиринт будет заполняться примерно так:



### 9.1.10. Реализация поиска в ширину

Рекурсивная функция `search` нам больше не понадобится. Вместо того чтобы хранить данные в цепочке вызовов функции, мы будем держать найденные клетки в очереди. Первой туда добавляется стартовая клетка.

А дальше, пока очередь не опустеет, извлекаем новую клетку из очереди и добавляем туда её соседей, не забывая обновить соответствующую ячейку массива `prev`.

Код для поиска мало отличается от функции `search`. Разница в том, что когда находится новая непройденная клетка, она добавляется в очередь, а не передаётся в качестве параметра в рекурсивный вызов.

C++

```
#include <queue>

void findPath()
{
  ...
  std::queue<Pos> q;
  q.push(start);

  while (!q.empty())
  {
    Pos cur = q.front();
    q.pop();

    for (auto d : dir)
    {
      int x = cur.x + d.x;
      int y = cur.y + d.y;
      if (prev[y][x].x < 0 && maze[y][x]
      ↪  != '#')
      {
        prev[y][x] = cur;
        q.push({x, y});
      }
    }
  }
}
```

Python

```
import queue

def findPath():
  ...
  q = queue.Queue()
  q.put(start)

  while not q.empty():
    cur = q.get()
    for d in dir:
      x = cur.x + d.x
      y = cur.y + d.y
      if prev[y][x].x < 0 and maze[y][x]
      ↪  != '#':
        prev[y][x] = cur
        q.put(Pos(x, y))
```



Весь остальной код, который мы написали, используется и здесь. Только теперь программа должна находить кратчайший маршрут правильно. Протестируйте её на разных примерах.

### 9.1.11. Задания для самостоятельной работы

**1.** Сделайте программу более надёжной. Проверяйте корректность входных данных: что лабиринт прямоугольный, что есть стены по краям, что вход и выход только в одном экземпляре.

**Листинг 9.1. labyrinth.cpp**

```
#include <iostream>
#include <queue>
#include <vector>
#include <string>

struct Pos
{
    int x, y;
};
std::vector<std::string> maze;
std::vector<std::vector<Pos>> prev;
Pos dir[4] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

void inputMaze()
{
    std::cout << "Введите лабиринт со стартовой клеткой S и финишной
    ↪ E. \n";
    while (true)
    {
        std::string s;
        std::getline(std::cin, s);
        if (s.empty())
        {
            break;
        }
        maze.push_back(s);
    }
}

Pos findChar(char c)
{
    for (int y = 0 ; y < maze.size() ; ++y)
    {
        for (int x = 0 ; x < maze[y].size() ; ++x)
        {
            if (maze[y][x] == c)
            {
                return {x, y};
            }
        }
```

```
    }
    return {-1, -1};
}

void findPath()
{
    prev.resize(maze.size());
    for (int y = 0 ; y < maze.size() ; ++y)
    {
        prev[y].resize(maze[y].size(), {-1, -1});
    }
    Pos start = findChar('S');
    prev[start.y][start.x] = start;
    std::queue<Pos> q;
    q.push(start);
    while (!q.empty())
    {
        Pos cur = q.front();
        q.pop();
        for (auto d : dir)
        {
            int x = cur.x + d.x;
            int y = cur.y + d.y;
            if (prev[y][x].x < 0 && maze[y][x] != '#')
            {
                prev[y][x] = cur;
                q.push({x, y});
            }
        }
    }
}

void outputPath()
{
    Pos end = findChar('E');
    if (end.x < 0 || prev[end.y][end.x].x < 0)
    {
        std::cout << "Пути между S и E не существует.\n";
        return;
    }
    while (prev[end.y][end.x].x != end.x
```

```
        || prev[end.y][end.x].y != end.y)
    {
        end = prev[end.y][end.x];
        maze[end.y][end.x] = '*';
    }
    maze[end.y][end.x] = 'S';
    std::cout << "Кратчайший путь: \n";
    for (auto s : maze)
    {
        std::cout << s << '\n';
    }
}

int main()
{
    inputMaze();
    findPath();
    outputPath();
}
```

**Листинг 9.2. labyrinth.py**

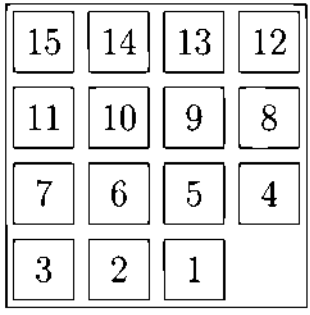
```
#!/usr/bin/python3
import queue
from dataclasses import dataclass

@dataclass
class Pos:
    x: int
    y: int
maze = []
prev = []
dir = [Pos(-1, 0), Pos(1, 0), Pos(0, -1), Pos(0, 1)]

def inputMaze():
    print('Введите лабиринт со стартовой клеткой S и финишной E.')
    while True:
        s = input()
        if len(s) == 0:
            break
        maze.append(list(s))
```

```
def findChar(c):
    for y in range(len(maze)):
        for x in range(len(maze[y])):
            if maze[y][x] == c:
                return Pos(x, y)
    return Pos(-1, -1)

def findPath():
    for y in range(len(maze)):
        prev.append([Pos(-1, -1)] * len(maze[y]))
    start = findChar('S')
    prev[start.y][start.x] = start
    q = queue.Queue()
    q.put(start)
    while not q.empty():
        cur = q.get()
        for d in dir:
            x = cur.x + d.x
            y = cur.y + d.y
            if prev[y][x].x < 0 and maze[y][x] != '#':
                prev[y][x] = cur
                q.put(Pos(x, y))

def outputPath():
    end = findChar('E')
    if end.x < 0 or prev[end.y][end.x].x < 0:
        print('Пути между S и E не существует.')
        return
    while prev[end.y][end.x].x != end.x or prev[end.y][end.x].y !=
    ↪ end.y:
        end = prev[end.y][end.x]
        maze[end.y][end.x] = '*'
    maze[end.y][end.x] = 'S'
    print('Кратчайший путь:')
    for s in maze:
        print("".join(s))

inputMaze()
findPath()
outputPath()
```

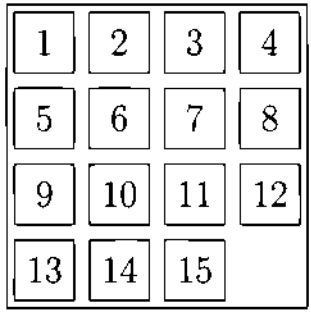
## 9.2. Игра «8»

Игра «8» — это упрощённый вариант игры «15». Об истории и математическом взгляде на эти головоломки можно почитать в книгах [2, 8, 9].

Суть игры «15» в следующем: имеются 15 фишек с числами от 1 до 15, расставленных на квадратном поле из 16 клеток. Одна клетка остаётся пустой, поэтому фишки могут перемещаться, сдвигаясь на пустое место. Изначально фишки расставлены на поле в некотором порядке:




Задача заключается в том чтобы перемещая фишки упорядочить их по возрастанию слева направо и сверху вниз:




Получить такой порядок можно не для любой исходной конфигурации. Поэтому мы будем создавать программу, которая за нас найдёт ответ или сообщит, если решения не существует.

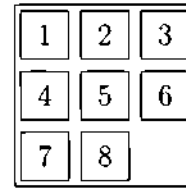
### 9.2.1. Оценка числа позиций

Игровая позиция — это положение фишек на поле. Вероятно, программе придётся перебирать разные варианты, передвигая фишки. Но сколько существует таких позиций?

Точно посчитать их количество достаточно сложно. Но можно получить упрощённую оценку, представив, что фишки могут перемещаться как угодно. То есть все возможные их перестановки достижимы (хотя на самом деле это и не так). Таких перестановок для игры «15» будет $16! = 20\,922\,789\,888\,000$, так как всего

клеток на поле 16, и в каждой из них находится уникальная фишка (или пустое место, которое тоже всего одно).

Это число очень велико. На хранение всех позиций точно не хватит памяти, а на обработку – времени. Но что если уменьшить число фишек до 8? Получится такое поле:



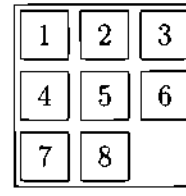

Число возможных перестановок для него составляет уже 9! = 362 880. Небольшая величина. Можно и в памяти все их сохранить, и перебором заниматься без особых проблем. Так что вместо игры «15» остановимся на игре «8».

### 9.2.2. Игровая позиция

Игровая позиция — это положение фишек на двумерной доске. Логично было бы использовать для их хранения двумерный массив. Каждая ячейка массива будет хранить число, соответствующее фишке, расположенной в этом месте. Но в C++ нет контейнера для двумерного массива, а ведь нам потребуется копировать позицию целиком, чтобы изменять её во время перебора. Так как с двумерными массивами это придётся делать с помощью двух циклов, лучше будем использовать одномерный массив.

Где-то одномерный массив усложнит манипуляции с координатами фишек (например, проверку выхода за границу строки влево или вправо), а где-то, наоборот, упростит (хранить надо всего одну координату, а не две).

Таким образом, следующая позиция описывается приведённым ниже фрагментом кода:




C++

```
#include <array>
const int N = 3;
const int SIZE = N * N;
typedef std::array<int, SIZE> Board;
Board board = {1, 2, 3, 4, 5, 6, 7, 8,
↪ 0};
```

Python

```
N = 3
SIZE = N * N
board = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

Контейнер `array` позволяет определять переменные-массивы, которые можно копировать как обычные значения. Циклы при этом не понадобятся. Константа `SIZE` нужна, потому что размер поля придётся использовать неоднократно — в других циклах, условиях и функциях. По той же причине нам нужен тип `Board`. Игровую позицию нужно будет копировать, передавать в функции и т.п. Каждый раз писать `std::array<int, SIZE>` было бы неудобно.

В языке Python встроенных констант нет, поэтому объявим просто переменную `SIZE`. Несмотря на то что она не используется при объявлении переменной-игровой позиции, она будет нужна в других местах. А размер массива здесь задаётся только неявно, поэтому никто не проверит, что в нём именно `SIZE` элементов.

### 9.2.3. Ввод позиции с клавиатуры

Чтобы начать игру, положение фишек надо внести в программу. Конечно же, лучше всего реализовать ввод позиции с клавиатуры, а не менять программу каждый раз.

C++

```
Board start;
bool b[SIZE] = {};
for (int i = 0 ; i < SIZE ; ++i)
{
  std::cin >> start[i];
  if (start[i] >= 0 && start[i] < SIZE)
  {
    b[start[i]] = true;
  }
}
// проверить ввод
for (int i = 0 ; i < SIZE ; ++i)
{
  if (!b[i])
  {
    std::cout << "Некорректная
    ↪  позиция\n";
    return 0;
  }
}
```

Python

```
start = [int(x) for x in input().split()]
b = [0] * SIZE
for s in start:
  if s >= 0 and s < SIZE:
    b[s] = 1
# проверить ввод
if sum(b) != SIZE:
  print('Некорректная позиция')
  exit(0)
```

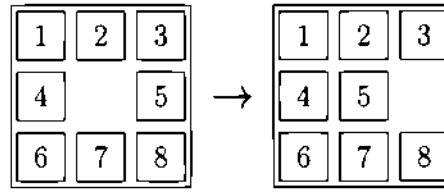
В массиве `b`, который состоит из 9 ячеек, мы записываем, какие числа были введены пользователем. Но только делается это не по порядку. В ячейку массива с индексом, соответствующим введённому числу, записывается значение. Тогда если весь этот массив после завершения ввода заполнен, то никакие значения не были введены дважды (и, соответственно, никакие числа не были пропущены).

### 9.2.4. Как перемещаются фишки

Каждая фишка «занимает» один элемент массива. И ещё есть один «пустой» элемент, где записан ноль, который кодирует пустое место.

Что будет происходить с позицией, если фишка перемещается? Во-первых, это эквивалентно тому, что по полю перемещается пустая клетка. А во-вторых, значения для фишки и для пустой клетки на поле будут обмениваться местами.

Вот движение фишки влево:




Или как это будет происходить в массиве:

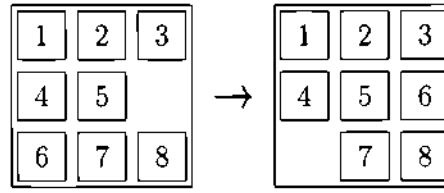
`1,2,3,4,0,5,6,7,8` → `1,2,3,4,5,0,6,7,8`

То есть фишка 5 передвинулась влево, а пустая клетка (число 0 в массиве) — вправо. Таким образом, номер ячейки, который «занимает» пустая клетка, увеличился на единицу.

Может ли ноль продолжать двигаться вправо?

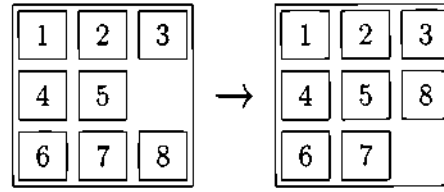
`1,2,3,4,5,0,6,7,8` → `1,2,3,4,5,6,0,7,8`

Игровая доска при этом изменяется так:




Уже не то, пустая клетка перескочила с одной строки на другую, но не ровно по вертикали, а в совсем другое место. Так делать нельзя. Получается, пустую клетку можно передвигать вправо, если она не находится в конце строки (на позициях 2 и 5).

При перемещении фишек вверх и вниз их вертикальная координата меняется на единицу.



Но такую координату мы не храним, поэтому придётся перевести её в номер элемента в массиве. Так как в одной строке доски всего 3 клетки, то индекс в массиве перемещаемой по вертикали фишки изменится на 3. Аналогичным образом изменится и позиция пустой клетки, только в обратную сторону:

$$1,2,3,4,5,0,6,7,8 \rightarrow 1,2,3,4,5,8,6,7,0$$

В программе перемещение одной фишки будет выглядеть как обмен двух ячеек массива:

C++
```
std::swap(board[empty], board[next]);
```

Python
```
board[empty], board[next] = board[next],
↪ board[empty]
```

Таким образом, чтобы сделать ход, нужно знать индекс пустой клетки и индекс перемещаемой фишки. В игровой позиции мы храним только состояния клеток, а положение пустой клетки отдельно не записано. Конечно, можно добавить переменные с этими координатами (или индексом). А можно просто определять, где находится пустая клетка, когда это необходимо. Первый способ должен быть немного эффективнее по времени работы, так как не нужен никакой поиск. Но из-за этого код станет более запутанным, поэтому пока что пойдём по простому пути и напишем функцию, которая находит позицию пустой клетки в массиве.

C++
```
int findEmpty(const Board &board)
{
  for (int i = 0 ; i < SIZE ; ++i)
  {
    if (board[i] == 0)
    {
      return i;
    }
  }
  return -1;
}
```

Python
```
def findEmpty(board):
for i in range(SIZE):
  if board[i] == 0:
    return i
return -1
```

Если клетка не нашлась, функция `findEmpty` возвращает `-1`. Такого быть не может, но если эту строчку не написать, то по меньшей мере компилятор C++ будет выдавать ошибку. Он же не знает, что нулевое значение всегда в массиве есть.

### 9.2.5. Определяем возможные направления движения

Когда пустая клетка найдена, соседствующие с ней фишки могут перемещаться, чтобы перейти к другой игровой позиции. Очевидно, что это не всегда 4 возможных направления, так как пустая клетка могла оказаться на краю поля. Как же нам найти корректные ходы?

Чтобы правильно проверять возможность движения влево или вправо, сначала надо получить горизонтальную координату пустой клетки. Это можно сделать

с помощью остатка от деления индекса на длину строки: `i % N`. `i` — это индекс в массиве, а клетки в нём идут подряд, сначала первая строка, потом вторая и т.д. И так как длина строки равна `N`, остаток от деления на `N` даёт нам позицию внутри строки.

Ход по горизонтали допустим, если пустая клетка не пытается «уехать» за пределы строки. То есть если её координата больше `0` при движении влево и меньше `N-1` при движении вправо.

Аналогичные проверки делаются для хода по вертикали, только они получаются немного проще. Так как первая (нулевая) строка доски занимает первые `N` элементов массива, то если индекс пустой клетки не меньше, чем `N`, она не в первой строке. Значит, может двигаться вверх. Похожее условие можно придумать и для последней строки.

Теперь можно описать эту логику в виде ветвлений:

C++

```
// пустая клетка движется влево
if (empty % N != 0)
{
  next = empty - 1;
  ...
}
// пустая клетка движется вправо
if (empty % N != N - 1)
{
  next = empty + 1;
  ...
}
// пустая клетка движется вверх
if (empty >= N)
{
  next = empty - N;
  ...
}
// пустая клетка движется вниз
if (empty < N * N - N)
{
  next = empty + N;
  ...
}
```

Python

```
# пустая клетка движется влево
if empty % N != 0:
  next = empty - 1
  ...
# пустая клетка движется вправо
if empty % N != N - 1:
  next = empty + 1
  ...
# пустая клетка движется вверх
if empty >= N:
  next = empty - N
  ...
# пустая клетка движется вниз
if empty < N * N - N:
  next = empty + N
  ...
```

Это удобно, если внутри оператора `if` выполняется какое-то несложное действие. Если же там хотя бы несколько строк кода, то копировать их нежелательно, потому что легко сделать ошибку, если эти строчки отличаются друг от друга либо будут меняться со временем. Такие ошибки одни из наиболее часто встречающихся в реальных программах.

Лучше всего повторяющиеся действия вынести в отдельную функцию либо переписать код, чтобы все направления обрабатывались поочерёдно с помощью цикла. Оба способа приемлемы, а первый уже даже «готов», вместо `...` уже

можно подставлять вызов функции. Поэтому сейчас ещё разберём второй способ — переделаем несколько похожих ветвлений в цикл.

Всего направлений движения 4, поэтому цикл будет из 4 итераций:

C++
```
empty = findEmpty(board);
for (int i = 0 ; i < 4 ; ++i)
  ...
```

Python
```
empty = findEmpty(board)
for i in range(4):
  ...
```

Теперь по номеру итерации нужно определить, куда перемещается пустая клетка. Вот здесь уже понадобится оператор `if`. Либо можно использовать сложную формулу или массив с возможными смещениями. При использовании массива есть только один недостаток — всё равно придётся написать непонятное условие в операторе `if`, чтобы контролировать корректность хода. Так что пока напишем несколько условий.

C++
```
empty = findEmpty(board);
for (int i = 1 ; i <= 4 ; ++i)
{
  int next = -1;
  if (i == 1 && empty % N != 0)
  {
    next = empty - 1;
  }
  else if (i == 2 && empty % N != N - 1)
  {
    next = empty + 1;
  }
  else if (i == 3 && empty >= N)
  {
    next = empty - N;
  }
  else if (i == 4 && empty < N * N - N)
  {
    next = empty + N;
  }
  if (next != -1)
  {
    Board b = board;
    std::swap(b[empty], b[next]);
    ...
  }
}
```

Python
```
empty = findEmpty(board)
for i in range(1, 5):
  next = -1
  if i == 1 and empty % N != 0:
    next = empty - 1
  elif i == 2 and empty % N != N - 1:
    next = empty + 1
  elif i == 3 and empty >= N:
    next = empty - N
  elif i == 4 and empty < N * N - N:
    next = empty + N
  if next != -1:
    b = board.copy()
    b[empty], b[next] = b[next], b[empty]
    ...
```

Немного громоздко, но в целом понятно, что происходит. Дальше с этими знаниями уже можно искать решение головоломки.

### 9.2.6. Пробуем рекурсивный перебор

До сих пор мы сталкивались с рекурсивным перебором в такой форме:

1. Если перебор закончен, вывести решение (или сделать что-то ещё).

2. Проверять все возможные позиции для новой фигуры:
   (a) разместить новую фигуру в выбранной позиции;
   (b) вызвать функцию рекурсивно;
   (c) убрать размещённую фигуру.

Здесь же никаких новых фигур нет, есть только перемещающиеся по полю фишки. Поэтому вместо действия «разместить новую фигуру» нужно «переместить какую-то фишку». Так мы надеемся приблизиться к решению головоломки, по аналогии с тем как мы приближаемся к решению задачи про ферзей, добавляя на доску очередную фигуру.

Получается почти такой же рекурсивный перебор, как и раньше. Берём игровую позицию, перемещаем какую-то из фишек, переходим к следующей позиции и так далее. Если будет достигнута финишная позиция, когда все фишки на своих местах, решение найдено.

C++

```
void find(Board &board)
{
  if (board_is_final(board))
  {
    // вывести решение
    return;
  }
  int empty = findEmpty(board);
  for (int i = 0 ; i < 4 ; ++i)
  {
    int next = -1;
    // цепочка ветвлений, проверяющая ход
    ...
    if (next != -1)
    {
      std::swap(board[empty],
      ↪ board[next]);
      find(board);
      std::swap(board[empty],
      ↪ board[next]);
    }
  }
}
```

Python

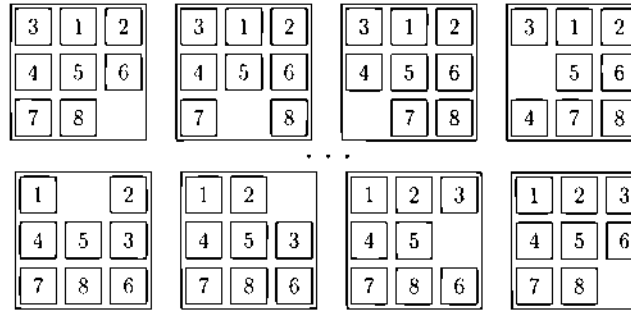
```
def find(board):
  if board_is_final(board):
    # вывести решение
    return;
  empty = findEmpty(board)
  for i in range(4):
    next = -1
    # цепочка ветвлений, проверяющая ход
    ...
    if next != -1:
      board[empty], board[next] =
      ↪ board[next], board[empty]
      find(board)
      board[next], board[empty] =
      ↪ board[empty], board[next]
```

Попробуйте собрать все эти функции в работающую программу. Несмотря на то что рекурсию мы ещё будем переделывать, почти весь код нам пригодится.

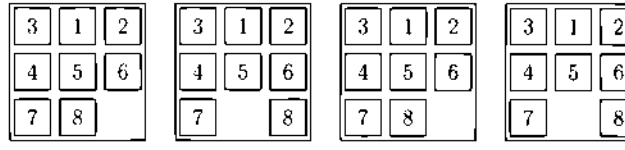
### 9.2.7. Почему рекурсия не подходит

Если вы попробовали запустить рекурсивный алгоритм на компьютере, скорее всего, у вас ничего не вышло. Представим, как могут перемещаться фишки.

Например, вот фрагмент последовательности перестановок, которая могла бы получиться:



Но тут есть один нюанс. Непонятно, кратчайший ли это путь к успеху, или нет: позиции могут повторяться. Например, вот так:



Нигде же не проверяется, что позиция уже встречалась раньше. Во-первых, кратчайшего решения не получится, а, во-вторых, программа может вообще не завершиться, если будет перемещать одни и те же фишки по кругу. Поэтому с отсечением повторов надо как-то разобраться.

### 9.2.8. Определение повторяющихся позиций

Как же понять, что позиция уже встречалась ранее? Для этого нужно в каком-то виде все их сохранять, а при получении новой комбинации фишек искать её в этом хранилище.

Описание позиции состоит из 10 цифр. Если просто выписать их подряд, получится десятизначное число (или девятизначное, если ноль оказался на первом месте). И если использовать это число как индекс в массиве, можно в каждой ячейке массива хранить признак того, что позиция уже встречалась.

Не слишком ли большой получится массив? Для десятизначного индекса понадобится $10^{10}$ элементов, по одному на индекс. Программа с таким массивом хоть и запустится, но будет тратить слишком уж много памяти.

Особенно это неприятно из-за того, что большинство элементов не будут использованы. Ведь если посчитать все возможные перестановки фишек, получится $9! = 362880$ вариантов. А на самом деле позиций ещё меньше, потому что не все перестановки можно получить в игре.

В идеале на каждую перестановку мы тратим только одну ячейку памяти. Этого можно достичь двумя способами: каким-то образом присвоить последователь-

ные номера всем перестановкам либо использовать не массив, а другое хранилище данных.

Оба способа хороши. Первый немного посложнее, а второй понадобится нам для других игр. Поэтому разберём второй вариант, когда вместо массива используется что-то ещё. Массив — это непрерывная последовательность однотипных элементов. Из-за его непрерывности освободить неиспользуемое место в нём не получится. Значит, нужна похожая структура данных, в которой по номеру элемента можно находить сам элемент (например, признак того, что игровая позиция уже рассматривалась).

Такая структура данных называется словарём. Словарь содержит набор элементов вида «ключ-значение». Ключ — это аналог индекса массива. Он используется для поиска нужного элемента, и, как правило, выбор нужного значения даже записывается так же, как для массива: `a[i]`.

Для игры «Восемь» ключом должна выступать игровая позиция. Ведь для достигнутой позиции мы ищем признак того, встречалась она уже или нет. Из словаря нужно будет извлечь этот признак по ключу (то есть позиции). Чтобы понять, нашёлся он или нет, ключи придётся сравнивать. Так чаще всего и устроены словари — что ключи там тоже хранятся, чтобы не ошибиться с выбором элемента[1]. Выходит, что нужен способ сравнить две заданные позиции.

В нашем случае можно сравнивать позиции, преобразованные в числовой вид, как описано выше. Например, позиции `[3, 5, 1, 2, 0, 6, 7, 8, 4]` будет соответствовать число 351206784. А числа уже легко проверять на равенство или определять наибольшее из них.

Начнём с функции для преобразования описания позиции в целое число. Чтобы это сделать, нужно домножить каждый из элементов массива на разную степень некоторого числа:

C++

```
int getKey(const Board &board)
{
    int r = 0;
    for (auto x : board)
    {
        r = r * 9 + x;
    }
    return r;
}
```

Python

```
def getKey(board):
    r = 0
    for x in board:
        r = r * 9 + x
    return r
```

[1] Из-за необходимости хранить ключ каждая позиция будет требовать больше одной ячейки памяти. Но суммарно их всё равно будет меньше, чем $10^{10}$ для обычного массива с позициями-индексами.

Число 9 вместо 10 выбрано для того, чтобы итоговый результат всегда умещался в тип `int`.

Для единообразия тут тоже используется 9, а не 10, как предлагалось выше, несмотря на то что в маленький тип данных укладываться не нужно.

Вместо «выписывания всех цифр подряд» с помощью умножения на 10, мы умножаем промежуточный результат на 9. Ячейки в массиве `board` принимают значения от 0 до 8, поэтому так можно делать, чтобы получить уникальный индекс для каждой позиции. Вот если бы мы умножали на 3, то последние две цифры $1,0$ и $0,3$ давали бы одинаковый вклад в вычисляемый индекс. И из-за этого позиции нельзя было бы различить в нашем словаре.

Работу функции можно описать такой формулой: $b_0 \cdot 9^8 + b_1 \cdot 9^7 + \cdots + b_7 \cdot 9 + b_8$. Нужные степени получаются за счёт многократного умножения частичной суммы на 9: $(((((((b_0 \cdot 9 + b_1) \cdot 9 + \cdots + b_7) \cdot 9 + b_8$.

### 9.2.9. Хранение позиций в словаре

Теперь, когда мы можем сопоставить уникальное число каждой позиции, легко сделать хранение их в словаре. Точнее, нам нужно хранить не сами позиции, а лишь признаки того, что они уже встречались раньше.

Пока обозначим такой признак числом 1. Тип `bool` использовать не будем, потому что дальше нам понадобится больше двух значений для этого признака.

Теперь можно объявить переменную, в которой будет храниться словарь. Она будет называться `positions`.

**C++**

```
#include <map>
...
std::map<int, int> positions;
```

`std::map` — это стандартный контейнер, который можно использовать в качестве словаря. У этого контейнера нужно указывать два параметра-типа. Здесь это `int` и `int`. Первый тип — это тип ключа, а второй — тип значения, которое нужно по этому ключу находить.

**Python**

```
positions = {}
```

В Python никакие типы при создании словаря указывать не нужно. Конечно, удобно, что писать надо намного меньше, чем в C++. Но и забыть, что именно мы задумали при объявлении этой переменной, довольно просто.

Чтобы добавить новую позицию в контейнер, нужно сначала рассчитать ключ, а затем использовать его для добавления числа в словарь. Этот код будет почти одинаковым в C++ и Python:

C++
```
int key = getKey(board);
positions[key] = 1;
```

Python
```
key = getKey(board)
positions[key] = 1
```

А вот фрагмент кода для проверки того, что позиция в словаре уже есть:

C++
```
int key = getKey(board);
if (positions.contains(key))
    ...
```

Python
```
key = getKey(board)
if key in positions:
    ...
```

Дальше остаётся добавить эти строчки к нашему рекурсивному перебору: в самом начале функции проверяем, что текущая позиция ещё не обрабатывалась, а потом либо завершаем функцию, либо добавляем позицию в контейнер. Функция никакие позиции не обрабатывает повторно, чтобы не попасть в бесконечный цикл.

C++
```
void find(Board &board)
{
  int key = getKey(board);
  if (positions.contains(key))
    return;
  positions[key] = 1;
  ...
}
```

Python
```
def find(board):
  key = getKey(board)
  if key in positions:
    return
  positions[key] = 1
  ...
```

### 9.2.10. Проблемы с рекурсией продолжаются

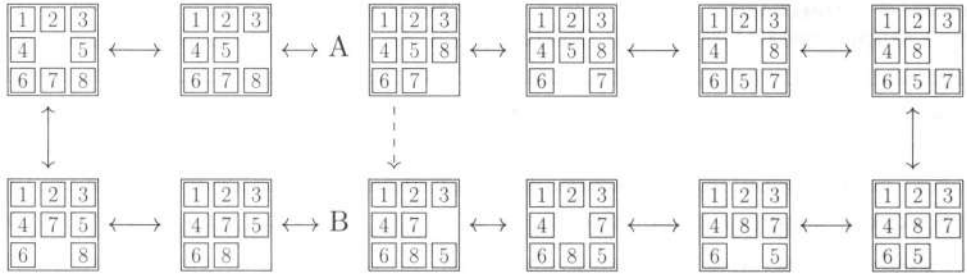
Наверняка даже после всех улучшений ваша рекурсивная программа не заработала нормально. А всё дело в том, что только исключать повторяющиеся позиции недостаточно. Обход уникальных позиций помогает не зайти в бесконечный цикл, но даже и без него цепочка перемещений фишек может быть слишком велика.

Что если значительная часть доступных позиций выстроится в один длинный путь к решению? Каждый рекурсивный вызов тратит память на локальные переменные и адрес возврата. Поэтому слишком длинная цепочка незавершённых вызовов `find` съедает всю доступную память, и программа аварийно завершается.

Из этой же проблемы следует, что перебор не сможет находить наилучшее решение. Мы не реализовали никаких механизмов, позволяющих хотя бы выбирать из двух разных решений то, которое содержит меньшее число ходов. Такие проверки добавить можно, но всё равно сам подход останется не слишком эффективным.

Сама задача поиска кратчайшей последовательности ходов неудобно (и зачастую неэффективно) решается рекурсивными алгоритмами. Поэтому давайте посмотрим на эту задачу под другим углом.

Поиск кратчайшей последовательности перемещений фишек очень похож на поиск кратчайшего пути. Посмотрите на картинку:



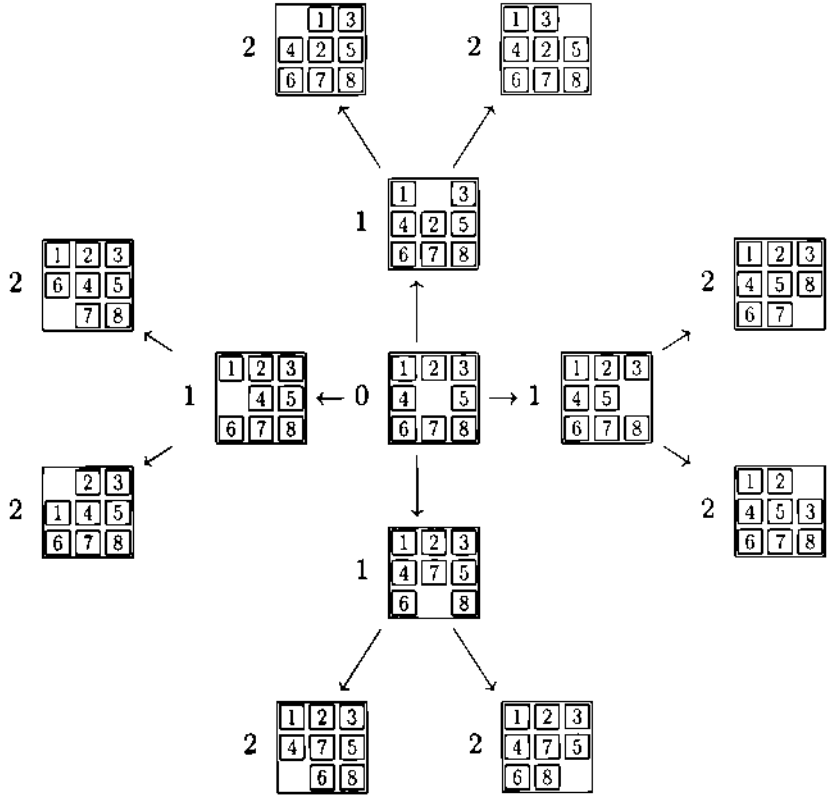

Предположим, что нам нужно из позиции A достичь позиции B, как показывает пунктирная стрелочка. Как рекурсивный алгоритм поиска поймёт, двигать фишки по часовой стрелке или против? Получается что, если повезёт, он найдёт более короткий путь (если двигаться по стрелкам влево), а не более длинный (если двигаться вправо).

К счастью, в этой ситуации применимы другие алгоритмы, позволяющие не полагаться на везение. Это алгоритмы поиска кратчайшего пути. На что похожа картинка выше? На города, связанные дорогами. Или на перекрёстки. Или на что угодно, для чего хочется найти маршрут.

### 9.2.11. Обход в ширину

Один из алгоритмов поиска кратчайшего маршрута — это обход в ширину (или поиск в ширину). Он применим для тех ситуаций, где длина отдельных переходов между точками маршрута одинакова. Это как раз наш случай. Один игровой ход (передвижение фишки) изменяет позицию на новую. То есть мы будто бы переходим из одной точки маршрута в другую. И главное в том, не какие будут ходы, а сколько их будет. То есть длина маршрута измеряется в числе переходов. Эта особенность и даёт нам возможность применить обход в ширину.

Как работает обход в ширину? Как следует из названия, он будет помогать нам обходить все игровые позиции. Сначала из стартовой позиции он находит все достижимые за один ход. Затем берёт весь этот набор и для него снова находит достижимые за один ход (исключая те, где мы уже были). Так получается набор из позиций, достижимых за два хода. Дальше этот процесс повторяется, пока все позиции не закончатся.



На рисунке рядом с каждой позицией указаны цифры, означающие число сделанных ходов (от 0 до 2). Сначала алгоритм обхода работает со стартовой позицией (0 ходов), затем обрабатывает полученные из неё позиции с 1. Из этих позиций он получает следующие, для них число ходов от старта уже будет 2. И так продолжается до тех пор, пока не будут пройдены все позиции (либо если не будет достигнута нужная расстановка).

### 9.2.12. Реализация обхода в ширину

Итак, чтобы запрограммировать обход в ширину и решить, наконец, головоломку, потребуются следующие компоненты:

1. Поиск соседних позиций для текущей.
2. Сохранение набора позиций для продолжения обхода.
3. Проверка того, что позиция уже встречалась ранее.

С первым и последним пунктами списка мы уже разбирались раньше. Осталось придумать, что делать со вторым. Обход в ширину обрабатывает найденный на предыдущем шаге набор позиций, значит, его нужно где-то хранить. При этом

новые позиции должны добавляться в новый набор, который будет использован на следующем шаге.

Если напрямую так и делать, понадобится два контейнера, в которых будут храниться позиции. Из одного достаются обрабатываемые, а в другой добавляются вновь найденные. Что-то вроде:

**C++**

```
std::list<Board> current, next;
// обрабатываем current
...
// переходим к следующему шагу
current = next;
next.clear();
```

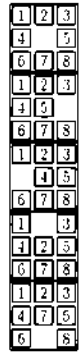
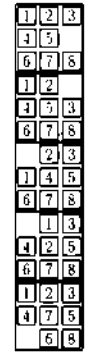
**Python**

```
current = []
next = []
# обрабатываем current
...
# переходим к следующему шагу
current = next
next = []
```

Это вполне неплохой способ, но его можно немного упростить. На самом деле не имеет смысла хранить текущий набор позиций в отдельном контейнере, его вполне нормально смешать с новыми позициями, которые будут обрабатываться дальше. Главное, чтобы этот алгоритм работал сначала с более ранними позициями, а потом с более поздними. То есть если в контейнер будет добавлена новая позиция, она не обработается раньше, чем те, что уже хранятся там. Снова получается очередь, как и в программе с лабиринтом.

Так как сначала мы находим позиции с «расстоянием» до них в 1 ход, они первые попадут в очередь. Затем, когда обход приступит к их обработке, в очередь будут добавляться позиции с «расстоянием» 2. Но алгоритм не будет с ними работать, пока не кончатся добавленные ранее. Этот порядок гарантируется нам устройством очереди.

Таким образом, для рисунка из предыдущего раздела может получиться такой порядок обхода позиций:

| **Расстояние** | **Обрабатываемая позиция** | **Добавляемые позиции** |
|---|---|---|
| 0 |  |     |
| 1 | | |
| 1 | | |
| 1 | | |
| 1 | | |
| 2 | и так далее | |

Создадим очередь для хранения позиций с помощью стандартной библиотеки:

C++
```
#include <queue>
...
std::queue<Board> q;
q.push(start);
positions[getKey(start)] = 0;
```

Python
```
import queue
...
q = queue.Queue()
q.put(start)
positions[getKey(start)] = 0
```

`start` — это начальная позиция головоломки. Та, которую пользователь ввёл с клавиатуры. Кроме добавления в очередь, мы также помечаем эту позицию как просмотренную, добавляя запись в контейнер `positions`.

Дальше в очередь попадают позиции, которые находит алгоритм. Он извлекает их по одной и «двигает фишки» в разных направлениях:

C++
```
while (!q.empty())
{
  Board board = q.front();
  q.pop();
  int empty = findEmpty(board);
  for (int i = 1 ; i <= 4 ; ++i)
  {
    int next = -1;
    // цепочка ветвлений, проверяющая ход
    ...
    if (next != -1)
    {
      Board b(board);
      std::swap(b[empty], b[next]);
      int key = getKey(b);
      if (!positions.contains(key))
      {
          positions[key] = 1;
          q.push(b);
      }
    }
  }
}
```

Python
```
while not q.empty():
  board = q.get()
  empty = findEmpty(board)
  for i in range(1,5):
    next = -1
    # цепочка ветвлений, проверяющая ход
    ...
    if next != -1:
      b = board.copy()
      b[empty], b[next] = b[next],
      ↪ b[empty]
      key = getKey(b)
      if key not in positions:
        positions[key] = 1
        q.put(b)
```

Многоточием тут заменён фрагмент кода с вычислением номера следующей клетки `next`, который мы уже разрабатывали ранее. В конце нужно проверить, достиг ли обход целевого положения фишек, когда все они идут по порядку:

C++
```
Board end = {1, 2, 3, 4, 5, 6, 7, 8, 0};
if (!positions.contains(getKey(end)))
{
  std::cout << "Решения нет!\n";
}
else
{
  // ???
}
```

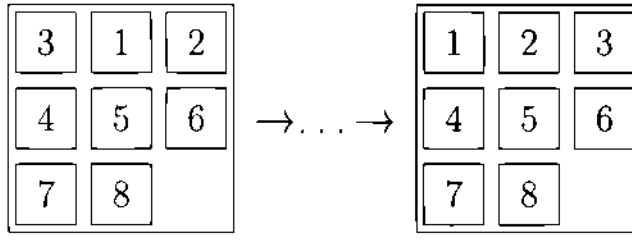
Python
```
end = [1, 2, 3, 4, 5, 6, 7, 8, 0]
if getKey(end) not in positions:
  print('Решения нет!')
else:
  # ???
```

Понять, что решения не существует, легко. Если финальная позиция не оказалась в контейнере `positions`, то её достигнуть нельзя. Но если оно существует, как найти последовательность перемещений фишек? Скоро разберёмся и с этим.

А пока попробуйте собрать все кусочки в одну программу, чтобы проверить, корректно ли программа сообщает о (не)существовании решения.

### 9.2.13. Восстановление последовательности перемещений

Программа, которую мы написали, умеет определять, что решение существует. То есть можно сделать так:




На месте многоточия находятся какие-то позиции. И информация о них даже хранится в `positions`. Но как они связаны между собой? Какие фишки двигать, чтобы всё получилось?

Оказывается, что уже сохранённой информации недостаточно. Когда мы помечаем позицию как просмотренную, то не запоминаем, откуда она взялась. А если это сделать, то потом можно будет вернуться назад по записанным перемещениям.

Небольшой задел под это уже есть — в `positions` для каждого найденного состояния игрового поля хранится число. Но ведь это число может кодировать перемещение фишки, благодаря которой мы попали в эту позицию.

Например, в процессе обработки  мы сдвинули 8 вправо и получилась позиция  . Пустая клетка сдвинулась влево, а такое перемещение кодируется индексом 1. То есть переменная цикла `i` при обработке этого перемещения равнялась 1.

Значит, если для новой позиции мы запишем в `positions[key]` значение `i` вместо значения `1`, то эту информацию можно будет использовать в дальнейшем, для того чтобы вернуться из  обратно в  .

Это происходит так: извлекаем для текущей позиции сохранённое в `positions` значение `i`. Это номер перемещения, которое привело из предыдущей позиции в текущую. Поэтому чтобы эту предыдущую позицию получить, нужно сделать обратное перемещение фишек. Только теперь корректность хода проверять не нужно, потому что это уже было сделано во время поиска решения.

Если повторять обратные перемещения достаточное число раз, мы вернёмся в исходную позицию. Так и получится решение, то есть последовательность положений фишек на доске. Только вот последовательность получится «перевёрнутая», от конечной позиции к начальной. А нам хотелось бы видеть её в том порядке, в котором головоломка решается, а не запутывается.

Последовательность ходов проще всего «перевернуть» с помощью рекурсии. Не выводить текущую позицию и потом искать дальше, а, наоборот, сначала выводить решение от предыдущей позиции до стартовой, а потом печатать текущую.

C++

```
void printSolution(Board &board)
{
  int dir = positions[getKey(board)];
  int empty = findEmpty(board);
  int next = -1;
  if (dir == 1)
  {
    next = empty + 1;
  }
  else if (dir == 2)
  {
    next = empty - 1;
  }
  else if (dir == 3)
  {
    next = empty + N;
  }
  else if (dir == 4)
  {
    next = empty - N;
  }
  if (next != -1)
  {
    std::swap(board[next], board[empty]);
    printSolution(board);
    std::swap(board[next], board[empty]);
  }
  for (auto x : board)
  {
    std::cout << x;
  }
  std::cout << "\n";
}
```

Python

```
def printSolution(board):
  dir = positions[getKey(board)]
  empty = findEmpty(board)
  if dir == 1:
    next = empty + 1
  elif dir == 2:
    next = empty - 1
  elif dir == 3:
    next = empty + N
  elif dir == 4:
    next = empty - N
  if next != -1:
    board[next], board[empty] =
    ↪ board[empty], board[next]
    printSolution(board)
    board[next], board[empty] =
    ↪ board[empty], board[next]

  for x in board:
    print(x, end='')
  print()
```

Работа функции остановится, когда не получится вычислить значение `next`. А это происходит, если `dir` не попадает в диапазон от 1 до 4. Когда это должно случиться? Если мы доходим до стартовой позиции. То есть перед поиском решения в `positions` надо было записывать, к примеру, 0, а не 1 для начального расположения фишек.

Сможете собрать все фрагменты кода, чтобы программа наконец-то заработала?

В книге [9] утверждается, что нет математической теории для поиска кратчайшего решения головоломок «8» и «15». Там же сказано, что для позиции 876543210 лучшее известное на тот момент (а книга достаточно старая) решение требовало 36 ходов. Может ли наша программа превзойти этот результат?

### 9.2.14. Задания для самостоятельной работы

Получившийся код ещё можно долго улучшать. Вот возможные идеи, для того чтобы сделать программу красивее.

**1.** Печатайте игровое поле в виде квадрата, а не в виде строки, похожей на пользовательский ввод.

**2.** Введите именованные константы для чисел, кодирующих перемещение фишек (`LEFT`, `RIGHT`, `UP`, `DOWN`).

**3.** Вынесите последовательность ветвлений с проверками направления движения (так как они дважды повторяются почти в идентичном виде) в отдельную функцию.

**Листинг 9.3. 8.cpp**

```
#include <iostream>
#include <array>
#include <map>
#include <queue>

const int N = 3;
const int SIZE = N * N;

typedef std::array<int, SIZE> Board;

int getKey(const Board &board)
{
    int r = 0;
    for (auto x : board)
    {
        r = r * 9 + x;
    }
    return r;
}
```

```
typedef std::map<long long, int> Positions;
Positions positions;

int findEmpty(const Board &board)
{
    for (int i = 0 ; i < SIZE ; ++i)
    {
        if (board[i] == 0)
        {
            return i;
        }
    }
    return -1;
}

void printSolution(Board &board)
{
    int dir = positions[getKey(board)];
    int empty = findEmpty(board);
    int next = -1;
    if (dir == 1)
    {
        next = empty + 1;
    }
    else if (dir == 2)
    {
        next = empty - 1;
    }
    else if (dir == 3)
    {
        next = empty + N;
    }
    else if (dir == 4)
    {
        next = empty - N;
    }
    if (next != -1)
    {
        std::swap(board[next], board[empty]);
        printSolution(board);
        std::swap(board[next], board[empty]);
```

```
    }
    for (auto x : board)
    {
        std::cout << x;
    }
    std::cout << "\n";
}

int main()
{
    Board start;
    bool b[SIZE] = {};
    for (int i = 0 ; i < SIZE ; ++i)
    {
        std::cin >> start[i];
        if (start[i] >= 0 && start[i] < SIZE)
        {
            b[start[i]] = true;
        }
    }
    // проверить ввод
    for (int i = 0 ; i < SIZE ; ++i)
    {
        if (!b[i])
        {
            std::cout << "Некорректная позиция\n";
            return 0;
        }
    }

    std::queue<Board> q;
    q.push(start);
    positions[getKey(start)] = -1;
    while (!q.empty())
    {
        Board board = q.front();
        q.pop();
        int empty = findEmpty(board);
        // перебираем все направления движения
        for (int i = 1 ; i <= 4 ; ++i)
        {
```

```
            int next = -1;
            if (i == 1 && empty % N != 0)
            {
                next = empty - 1;
            }
            else if (i == 2 && empty % N != N - 1)
            {
                next = empty + 1;
            }
            else if (i == 3 && empty >= N)
            {
                next = empty - N;
            }
            else if (i == 4 && empty < N * N - N)
            {
                next = empty + N;
            }
            if (next != -1)
            {
                Board b = board;
                std::swap(b[empty], b[next]);
                int key = getKey(b);
                if (!positions.contains(key))
                {
                    positions[key] = i;
                    q.push(b);
                }
            }
        }
    }

    Board final = {1, 2, 3, 4, 5, 6, 7, 8, 0};
    if (!positions.contains(getKey(final)))
    {
        std::cout << "Решения не существует! \n";
    }
    else
    {
        printSolution(final);
    }
}
```

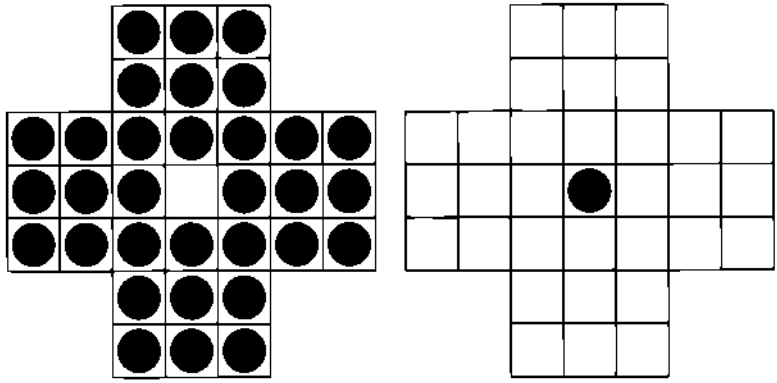
**Листинг 9.4. 8.py**

```
#!/usr/bin/python3
import queue

N = 3
SIZE = N * N
positions = {}

def getKey(board):
    r = 0
    for x in board:
        r = r * 9 + x
    return r

def findEmpty(board):
    for i in range(SIZE):
        if board[i] == 0:
            return i
    return -1

def printSolution(board):
    dir = positions[getKey(board)]
    empty = findEmpty(board)
    next = -1
    if dir == 1:
        next = empty + 1
    elif dir == 2:
        next = empty - 1
    elif dir == 3:
        next = empty + N
    elif dir == 4:
        next = empty - N
    if next != -1:
        board[next], board[empty] = board[empty], board[next]
        printSolution(board)
        board[next], board[empty] = board[empty], board[next]
    for x in board:
        print(x, end='')
    print()
```

```
start = [int(x) for x in input().split()]
b = [0] * SIZE
for s in start:
    if s >= 0 and s < SIZE:
        b[s] = 1
# проверить ввод
if sum(b) != SIZE:
    print('Некорректная позиция')
    exit(0)

q = queue.Queue()
q.put(start)
positions[getKey(start)] = 0
while not q.empty():
    board = q.get()
    empty = findEmpty(board)
    # перебираем все направления движения
    for i in range(1, 5):
        next = -1
        if i == 1 and empty % N != 0:
            next = empty - 1
        elif i == 2 and empty % N != N - 1:
            next = empty + 1
        elif i == 3 and empty >= N:
            next = empty - N
        elif i == 4 and empty < N * N - N:
            next = empty + N
        if next != -1:
            b = board.copy()
            b[empty], b[next] = b[next], b[empty]
            key = getKey(b)
            if key not in positions:
                positions[key] = i
                q.put(b)

final = [1, 2, 3, 4, 5, 6, 7, 8, 0]
if getKey(final) not in positions:
    print('Решения не существует!')
else:
    printSolution(final)
```

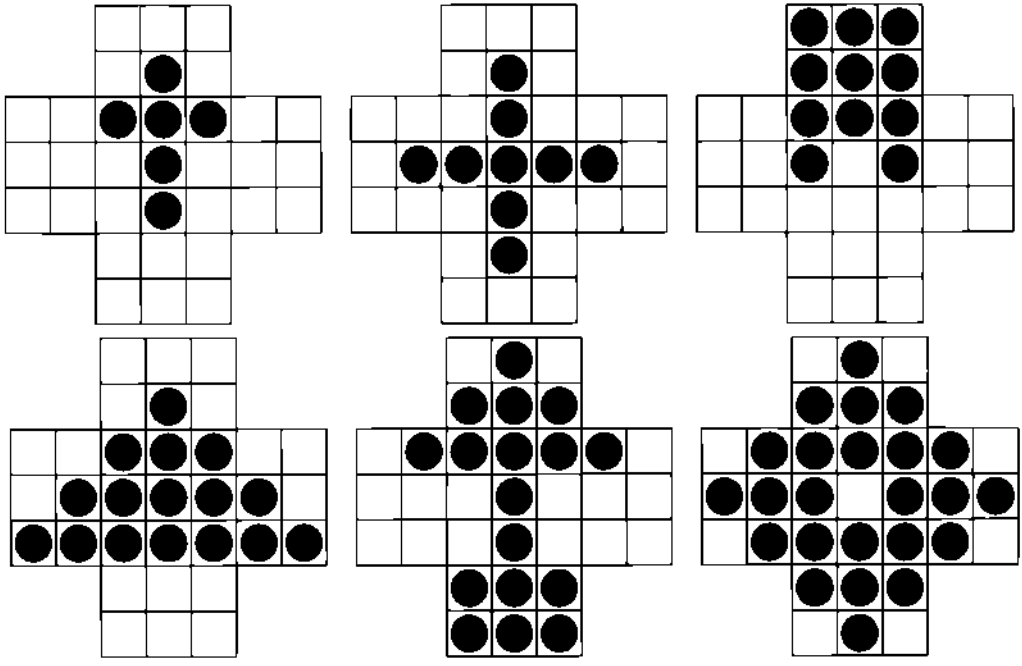
## 9.3. Игра «Солитер»

Классический вариант игры — из позиции слева получить позицию справа:



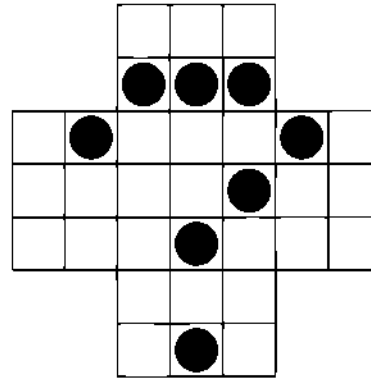
Для этого нужно делать фишками ходы по следующим правилам: любая выбранная фишка может перепрыгнуть другую, стоящую рядом (по горизонтали или вертикали), и приземлиться на следующую за ней пустую клетку. А та фишка, которую перепрыгнули, снимается с доски.

Хитрость в том, чтобы выбрать правильную последовательность ходов и получить в конце нужную расстановку. Например, в книге [2] предлагаются следующие варианты начальной расстановки фишек:



### 9.3.1. Игровая позиция

Что мы видим на игровом поле?



На нём есть пустые клетки, есть фишки, а также запретные места, куда фишки попасть не могут[1]. Что из этого относится к игровой позиции?

Игровая позиция — это те части игры, которые меняются. В нашем случае это расположение фишек на доске. Таким образом, запретные или непроходимые поля к позиции не относятся.

С другой стороны, в программу можно добавить поддержку разных вариантов поля. Тогда эту конфигурацию (где обычная клетка, а где непроходимый угол) тоже нужно где-то хранить. Хотя она и не будет изменяться в течение одной игры, в отличие от положения фишек.

Итак, для поля $7 \times 7$ клеток нужно закодировать положение максимум 48 фишек (это в том случае, когда угловые клетки являются проходимыми).

Есть несколько вариантов. Первый — это хранить координаты всех фишек в виде массива пар координат:

C++
```
int position[][2] = {
    {1, 1}, {2, 1}, ...
};
```

Python
```
position = [
    (1, 1), (2, 1), ...
]
```

В чём тут удобство? Легко перебирать все оставшиеся на доске фишки, когда мы ищем варианты для следующего хода. Но что будет с перемещаемыми фишками дальше? Сначала надо проверить, есть ли рядом фишка, чтобы через неё перепрыгнуть. Как это сделать? Только проверив снова весь массив, так как фишки в нём никак не упорядочены. Потом нужно убедиться, что клетка, куда

[1] Запретных мест на рисунке не видно, потому что поле не прямоугольное и они никак не очерчены. А вот в программе проще всего будет сделать поле прямоугольным, поэтому придётся лишние клетки как-то обозначить.

фишка перепрыгивает, не занята. И снова придётся перебирать всё содержимое массива фишек. Не очень-то удобный подход.

Так как нужно быстро и эффективно проверять, что находится в соседней клетке, а также через одну, логичнее завести двумерный массив $7 \times 7$. Например, 0 в таком массиве будет означать свободную клетку, а 1 — занятую:

C++
```
int position[7][7] = {
    {0, 0, 1, 1, 1, 0, 0}, ...
};
```

Python
```
position = [
    [0, 0, 1, 1, 1, 0, 0], ...
]
```

Такая структура данных подходит очень хорошо для поиска ходов. Что ещё понадобится делать с игровым полем в программе? Помните, что было в игре «8»? Во время обхода в ширину (для поиска решения) мы сохраняли позиции в очереди, а также сравнивали их, чтобы повторно не попадать на одни и те же.

Получается, что если хранить игровую позицию в массиве, придётся многократно копировать массивы, а также сравнивать их. Это возможно, но не слишком удобно. Да и памяти много требует: каждая клетка описывается минимум одним байтом (для типов `char` и `bool` в C++). А для типа `int` или в языке Python и того больше.

Но в реальности для описания каждой клетки используется один бит - либо есть фишка в определённой клетке, либо её нет. Выходит, что на всё поле потребуется 49 бит, что легко помещается в большие целые типы вроде `long long int` или `uint64_t`. Поэтому чтобы описать всю игровую позицию, потребуется всего одна целая переменная. Это удобно и для хранения в памяти (занимает мало места), и для поиска новых позиций (два целых числа для двух позиций легко сравнить между собой).

Тогда задать стартовую позицию для программы можно, например, вот так:

C++
```
uint64_t position = 0b00111000011100...;
```

Python
```
position = 0b00111000011100...
```

Биты, описывающие расположение фишек, соответствуют выстроенным последовательно строкам доски. Сначала первая строка, потом вторая, потом третья...

### 9.3.2. Вывод позиции на экран

Если так удобно хранить позицию в виде битов целого числа, то в чём же подвох? А в том, что при этом несколько усложняется работа с отдельными фишками.

Так как в данном случае многократная экономия памяти стоит того, будем разбираться с отдельными битами. Чтобы потренироваться, напишем функцию вывода поля на экран.

C++

```
static const int N = 7;
typedef uint64_t Field;

void printField(Field f)
{
  for (int y = 0 ; y < N ; ++y)
  {
    for (int x = 0 ; x < N ; ++x)
    {
      if (hasPeg(f, x, y))
      {
        std::cout << "o";
      }
      else
      {
        std::cout << ".";
      }
    }
    std::cout << "\n";
  }
  std::cout << "\n";
}
```

Директива `typedef` создаёт тип с названием `Field`, чтобы в программе удобнее было объявлять соответствующие переменные, не задумываясь о том, какая должна быть у них разрядность.

Python

```
N = 7

def printField(f):
  for y in range(N):
    for x in range(N):
      if hasPeg(f, x, y):
        print('o', end='')
      else:
        print('.', end='')
    print()
  print()
```

В Python типы описывать не нужно. Но и подсказок о том, что это за переменная в каком-то месте кода, из типов получить не выйдет.

Сам алгоритм вывода поля несложный. Проходим по всем строкам и столбцам с помощью вложенных циклов и проверяем, есть ли фишка по нужным координатам. Но вопрос и состоял в том, как проверить, есть ли в нужном месте фишка, а тут просто вызывается функция **hasPeg**!

Функция принимает на вход поле (большое число) и две координаты (по вертикали и горизонтали). Координатам соответствует одна клетка или же один бит в закодированном поле. Как проверить бит в числе? Нужно выполнить операцию «побитовое И» исходного числа и другого, в котором только этот один бит установлен. Тогда если получится 0, единичного бита в нужном месте нет. А значит, и фишки нет.

А как получить единичный бит в нужном месте? И какое место нужное? Так как строки поля кодируются выстроенными подряд битами, то надо сдвинуть единичку (установленный бит на позиции 0) влево на величину `y * N + x`. Так единичка попадёт на нужную позицию, и получится число для выполнения проверки:

**C++**

```
bool hasPeg(Field f, int x, int y)
{
  return f & (1ULL << (y * N + x));
}
```

1ULL тут написано из-за того, что поле хранится в 64-битном числе (тип uint64_t). А если написать просто 1, то константа будет 32-битная (тип int) и сдвиг сработает неправильно.

**Python**

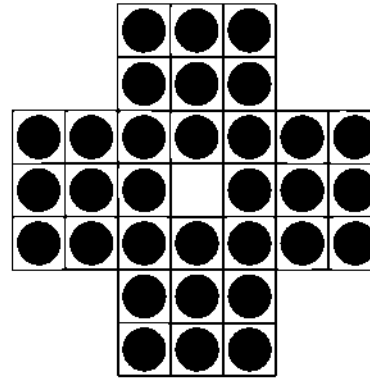
```
def hasPeg(f, x, y):
  return f & (1 << (y * N + x)) != 0
```

Сравнение с нулём необходимо, чтобы результат был не целым числом, а величиной булевого типа.

Потренируйтесь в создании «магических констант», вызывая функцию `printField` так, чтобы получались расположения фишек, изображенные в самом начале раздела.

### 9.3.3. Ввод стартовой позиции

Чтобы программа решала не просто одну заданную головоломку, напишем функцию для ввода игрового поля. Стандартное стартовое поле



будет задаваться в виде набора строк символов:

**Экран 1. Начальное положение фишек в игре**

```
XXoooXX
XXoooXX
ooooooo
ooo.ooo
ooooooo
XXoooXX
XXoooXX
```

Для каждой буквы «o» нужно записать единичку в нужный бит числа-позиции, а для точек — ноль (можно сразу инициализировать позицию нулём и ничего специально не делать).

Остаются клетки, где записан символ «X». Туда фишки попадать не должны, поэтому логично в описании позиции оставить там нули, а непосредственно с ходами разобраться чуть позднее.

Тогда код функции будет состоять из считывания $N$ строк и проверки каждого символа в них:

C++

```
Field readField()
{
  Field res = 0;
  for (int i = 0 ; i < N ; ++i)
  {
    std::string s;
    do
    {
      std::getline(std::cin, s);
    }
    while (s.length() != N);

    for (int j = 0 ; j < N ; ++j)
    {
      if (s[j] == 'o')
      {
        res |= 1ULL << (i * N + j);
      }
    }
  }
  return res;
}
```

Python

```
def readField():
  field = []
  for i in range(N):
    while True:
      s = input()
      if len(s) == N:
        break
      field.append(s)

  res = 0
  for i in range(N):
    for j in range(N):
      index = i * N + j
      if field[i][j] == 'o':
        res |= 1 << index
  return res
```

### 9.3.4. Поиск в ширину

Мы уже знаем, что решение подобных головоломок можно находить с помощью поиска в ширину. Для этого нужно хранить игровые позиции в очереди и извлекать их одну за другой, чтобы находить следующие.

При этом, чтобы искать не просто само решение, но и путь к нему, нужно для каждой найденной позиции сохранять предыдущий шаг. Можно это делать с помощью ассоциативного массива. Ключом будет найденная позиция, а значением — предыдущая. Очень удобно, что позиция описывается одним целым числом. Благодаря этому её можно сразу использовать в качестве ключа без написания дополнительного кода.

Тогда получится такая заготовка для поиска в ширину:

C++

```
#include <queue>
#include <unordered_map>

std::unordered_map<Field, Field>
↪ positions;
...
int main()
{
  ...
  std::queue<Field> q;
  q.push(start);
  positions[start] = 0;
  Field last;
  while (!q.empty())
  {
    last = q.front();
    count(last);
    q.pop();
    Field next;
    // что-то делаем с позицией last,
    // чтобы найти следующую next
    if (!positions.contains(next))
    {
        positions[next] = last;
        q.push(next);
    }
  }
  // выводим ответ для last
}
```

Python

```
import queue
positions = {}
...
q = queue.Queue()
q.put(start)
positions[start] = 0
while not q.empty():
  last = q.get()
  # что-то делаем с позицией last,
  # чтобы найти следующую next
  next = ...
  if next not in positions:
    positions[next] = last
    q.put(next)

# выводим ответ для last
```

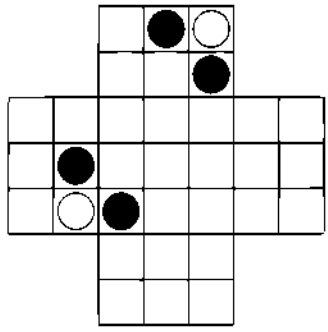
Так как цель игры — убрать с поля как можно больше фишек, то самая последняя обработанная позиция подойдёт и в качестве ответа. Ведь во время каждого хода с поля исчезает одна фишка, а в очередь позиции попадают в порядке убывания числа фишек в них. Тогда последней будет обработана одна из тех позиций, в которых меньше всего фишек, значит, её и надо выводить как ответ.

### 9.3.5. Определение возможных перемещений

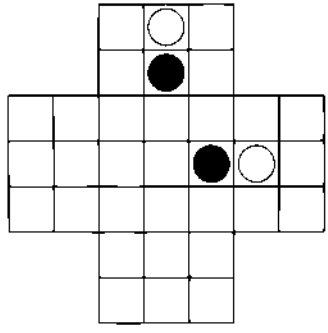
В заготовке поиска решений не хватает двух вещей — поиска следующих позиций, в которые можно перейти из заданной, а также вывода ответа. Начнём с поиска.

Возможные переходы между позициями не получится перечислить заранее. Во-первых, их очень много, а во-вторых, мы же их и ищем. Получается, нужно просматривать все фишки на поле, а потом выбирать для них возможные ходы.

Фишка может прыгнуть в одном из четырёх направлений. Но всегда ли? Нет, не всегда. Например, на следующем поле у белых фишек только по два возможных хода.



А вот здесь у белых фишек только по одному ходу. И как бы остальные фишки ни стояли, больше одного варианта хода у них не будет.



Когда ход невозможен? Если в направлении прыжка оказывается граница поля либо клетка, в которой фишка не может находиться.

Эти проверки можно делать как во время поиска, так и заранее. Конечно, во время поиска это будет несколько проще (и кода чуть поменьше), но зато если мы все проверим заранее, не придётся тратить лишнее время при переборе. Это довольно выгодно, потому что перебирается много тысяч позиций, и экономия на операциях ускорит получение результата.

Когда фишка прыгает в каком-то направлении, нужно ещё учитывать и то, есть ли рядом другая фишка (через которую прыгаем), а также пустая ли целевая клетка. Поэтому для каждой клетки нужно определить только возможные направления. А сами проверки, что в этом направлении находится, будем делать во время перебора.

Границу поля найти очень легко. Например, если горизонтальная координата фишки равна 0 или 1, то фишка не может прыгнуть влево, потому что так координата станет равной $-1$.

А что делать с непроходимыми клетками? В описании позиции они никак не закодированы. Да это и невозможно сделать, потому что каждая клетка описывается одним битом — либо там есть фишка, либо нет. Для непроходимых клеток никакого кода не выбрать.

Придётся вспомнить о той части кода, где вводилась начальная позиция. Там информация о видах клеток ещё была доступна – пользователь вводил символ «.», «o» или «X».

Нас интересуют либо пустые клетки, либо те, где уже есть фишки. Для них и будем искать варианты ходов (ведь в пустые клетки тоже может переместиться какая-то фишка).

Во-первых, нам понадобится массив, где для каждой клетки будут сохраняться ходы. А элементами массива будут контейнеры со списком ходов. Что такое ход? Можно кодировать его числом от 0 до 3, так как направлений движения всего 4.

А чтобы узнать, возможен ли выбранный ход из выбранной клетки, достаточно проверить клетку на расстоянии 2 в нужном направлении. Если она находится в границах поля, а также не содержит символа X, то ход возможен. При этом соседнюю клетку проверять нет смысла, потому что если там окажется одинокий символ X, фишка всё равно не сможет прыгнуть, потому что прыгать надо через другую фишку, а там её точно не окажется.

C++

```
#include <vector>
std::vector<int> moves[N * N];
static const int dx[4] = {-1, 1, 0, 0};
static const int dy[4] = {0, 0, -1, 1};

...

if (field[i][j] != 'X')
{
  for (int d = 0 ; d < 4 ; ++d)
  {
    int x = j + dx[d] * 2;
    int y = i + dy[d] * 2;
    if (x >= 0 && x < N
      && y >= 0 && y < N
      && field[y][x] != 'X')
    {
      moves[index].push_back(d);
    }
  }
}
```

Python

```
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]
moves = []
...
m = []
if field[i][j] != 'X':
  for d in range(4):
    x = j + dx[d] * 2
    y = i + dy[d] * 2
    if x >= 0 and x < N
      and y >= 0 and y < N
      and field[y][x] != 'X':
      m.append(d)
moves.append(m)
```

### 9.3.6. Перемещение фишек

Теперь, когда список ходов уже есть, осталось научиться «перемещать» фишки. Как вы помните, клетке поля с номером $i = x + N \cdot y$ соответствует $i$-й бит числа, кодирующего игровую позицию.

Фишка перемещается из $i$-й клетки, перепрыгивает $j$-ю, а останавливается в $k$-й. До того как фишка прыгнула, первые два из этих битов были установлены (фишки в этих клетках есть), а третий сброшен (клетка пустая). После хода

всё наоборот — первые два бита сброшены (из первой клетки фишка ушла, а вторую убрали), а третий установлен (фишка переместилась сюда).

Если применить к этим трём битам операцию «хог» (исключающее или) с единицей (конечно же, сдвинутой на нужное место), то произойдёт нужное изменение, каждый из них изменит своё значение на противоположное.

Теперь доработаем фрагмент с обходом в ширину. Вместо одного расчёта следующей позиции `next` их может быть много, от 1 до 4 для каждой из фишек на поле.

**C++**

```
while (!q.empty())
{
  last = q.front();
  count(last);
  q.pop();
  for (int i = 0 ; i < N ; ++i)
  {
    for (int j = 0 ; j < N ; ++j)
    {
      if (!hasPeg(last, j, i))
      {
        continue;
      }
      for (int d : moves[i * N + j])
      {
        int x = j + dx[d] * 2;
        int y = i + dy[d] * 2;
        int mx = j + dx[d];
        int my = i + dy[d];
        if (hasPeg(last, x, y)
          || !hasPeg(last, mx, my))
        {
          continue;
        }
        Field next = last;
        next ^= 1ULL << (i * N + j);
        next ^= 1ULL << (y * N + x);
        next ^= 1ULL << (my * N + mx);
        if (!positions.contains(next))
        {
          positions[next] = last;
          q.push(next);
        }
      }
    }
  }
}
```

**Python**

```
while not q.empty():
  last = q.get()
  for i in range(N):
    for j in range(N):
      if not hasPeg(last, j, i):
        continue
      for d in moves[i * N + j]:
        x = j + dx[d] * 2
        y = i + dy[d] * 2
        mx = j + dx[d]
        my = i + dy[d]
        if hasPeg(last, x, y)
          or not hasPeg(last, mx, my):
          continue
        next = last
        next ^= 1 << (i * N + j)
        next ^= 1 << (y * N + x)
        next ^= 1 << (my * N + mx)
        if next not in positions:
          positions[next] = last
          q.put(next)
```

Уже почти всё готово. Вставьте в конец программы вывод переменной `last` с помощью функции `printField`. Проверьте, что программа не зависает, а получившаяся позиция содержит мало фишек (сама позиция, конечно же, зависит от стартовой расстановки).

Главное, не начинайте тестировать с позиции, в которой только одна пустая клетка.

А когда убедитесь, что программа не зависает, запаситесь терпением и запускайте её для полной доски. Позиций в ходе перебора генерируется много, поэтому в памяти программы постоянно создаются новые объекты. Работать программа будет несколько минут, а при использовании Python даже дольше.

### 9.3.7. Вывод ответа

Вывод решения можно сделать так же, как это было с игрой «8» – рекурсивная функция, которая находит предыдущую позицию, печатает решение для неё, а затем выводит текущую. Так нужно делать, потому что для каждой найденной позиции в контейнере `positions` хранится предыдущая, из которой мы в неё перешли.

Для стартовой позиции там хранится 0, потому что мы его туда добавили перед началом перебора, а 0 — это позиция без фишек вообще, достичь которую невозможно (кто-то же должен перепрыгнуть последнюю фишку).

**C++**

```
void printSolution(Field f)
{
  Field next = positions[f];
  if (next)
  {
    printSolution(next);
  }
  printField(f);
}

printSolution(last);
```

**Python**

```
def printSolution(f):
  next = positions[f]
  if next:
    printSolution(next)
  printField(f)

printSolution(last)
```

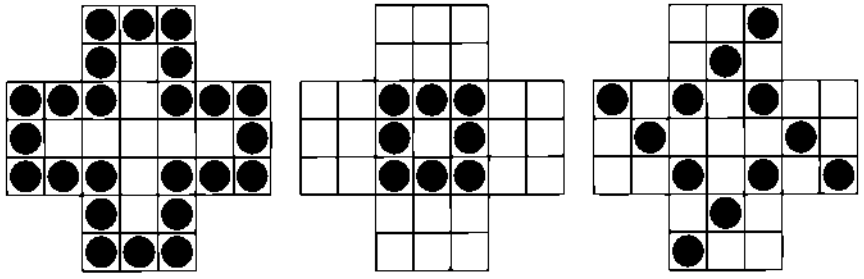
### 9.3.8. Задания для самостоятельной работы

**1.** Посчитайте число позиций, которые обработала программа, если ограничивать поле конфигурациями $5 \times 5$, $6 \times 6$, $7 \times 7$ фишек. Справится ли программа с полями большего размера? По крайней мере, битов в 64-битном числе хватает для полей $8 \times 8$.

**2.** Сделайте вывод поля более красивым, когда непроходимые клетки при выводе на экран изображаются в виде отдельных символов (хотя бы тех же «X»), а не просто точек.

**3.** Доработайте программу, чтобы в ней появился выбор финальной позиции пользователем. Например, её можно тоже вводить с клавиатуры. В таком варианте программа сможет строить заданные узоры.

Вот варианты финальных позиций из книги [2]:



**4.** Сделайте так, чтобы размер поля определялся автоматически (на основе введённой пользователем стартовой позиции), а не был задан с помощью константы в коде. Поле даже не обязано быть квадратным.

**5.** Придумайте, как доработать вывод поля, чтобы подсвечивалась последняя фишка, которая делала ход.

### Листинг 9.5. solitaire.cpp

```
#include <iostream>
#include <inttypes.h>
#include <queue>
#include <string>
#include <vector>
#include <unordered_map>

static const int N = 7;
typedef uint64_t Field;
std::unordered_map<Field, Field> positions;
static const int dx[4] = {-1, 1, 0, 0};
static const int dy[4] = {0, 0, -1, 1};
std::vector<int> moves[N * N];

bool hasPeg(Field f, int x, int y)
{
    return f & (1ULL << (y * N + x));
}
```

```
void printField(Field f)
{
    for (int y = 0 ; y < N ; ++y)
    {
        for (int x = 0 ; x < N ; ++x)
        {
            if (hasPeg(f, x, y))
            {
                std::cout << "o";
            }
            else
            {
                std::cout << ".";
            }
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}

void printSolution(Field f)
{
    Field next = positions[f];
    if (next)
    {
        printSolution(next);
    }
    printField(f);
}

Field readField()
{
    std::string field[N];
    for (int i = 0 ; i < N ; ++i)
    {
        do
        {
            std::getline(std::cin, field[i]);
        }
        while (field[i].length() != N);
    }
```

```
    Field res = 0;
    for (int i = 0 ; i < N ; ++i)
    {
        for (int j = 0 ; j < N ; ++j)
        {
            int index = i * N + j;
            if (field[i][j] == 'o')
            {
                res |= 1ULL << index;
            }
            if (field[i][j] != 'X')
            {
                for (int d = 0 ; d < 4 ; ++d)
                {
                    int x = j + dx[d] * 2;
                    int y = i + dy[d] * 2;
                    if (x >= 0 && x < N && y >= 0 && y < N
                        && field[y][x] != 'X')
                    {
                        moves[index].push_back(d);
                    }
                }
            }
        }
    }
    return res;
}

int main()
{
    Field start = readField();

    std::queue<Field> q;
    q.push(start);
    positions[start] = 0;
    Field last;
    while (!q.empty())
    {
        last = q.front();
        q.pop();
        for (int i = 0 ; i < N ; ++i)
```

```
        {
            for (int j = 0 ; j < N ; ++j)
            {
                if (!hasPeg(last, j, i))
                {
                    continue;
                }
                for (int d : moves[i * N + j])
                {
                    int x = j + dx[d] * 2;
                    int y = i + dy[d] * 2;
                    int mx = j + dx[d];
                    int my = i + dy[d];
                    if (hasPeg(last, x, y) || !hasPeg(last, mx, my))
                    {
                        continue;
                    }
                    Field next = last;
                    next ^= 1ULL << (i * N + j);
                    next ^= 1ULL << (y * N + x);
                    next ^= 1ULL << (my * N + mx);
                    if (!positions.contains(next))
                    {
                        positions[next] = last;
                        q.push(next);
                    }
                }
            }
        }
    }

    std::cout << "\n";
    printSolution(last);
}
```

**Листинг 9.6. solitaire.py**

```
#!/usr/bin/python3
import queue

N = 7
positions = {}
dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]
moves = []

def hasPeg(f, x, y):
    return f & (1 << (y * N + x)) != 0

def printField(f):
    for y in range(N):
        for x in range(N):
            if hasPeg(f, x, y):
                print('o', end='')
            else:
                print('.', end='')
        print()
    print()

def printSolution(f):
    next = positions[f]
    if next:
        printSolution(next)
    printField(f)

def readField():
    field = []
    for i in range(N):
        while True:
            s = input()
            if len(s) == N:
                break
        field.append(s)
    res = 0
    for i in range(N):
        for j in range(N):
            index = i * N + j
```

```
            if field[i][j] == 'o':
                res |= 1 << index
            m = []
            if field[i][j] != 'X':
                for d in range(4):
                    x = j + dx[d] * 2
                    y = i + dy[d] * 2
                    if x >= 0 and x < N and y >= 0 and y < N and
                    ↪ field[y][x] != 'X':
                        m.append(d)
            moves.append(m)
    return res

start = readField()

q = queue.Queue()
q.put(start)
positions[start] = 0
while not q.empty():
    last = q.get()
    for i in range(N):
        for j in range(N):
            if not hasPeg(last, j, i):
                continue
            for d in moves[i * N + j]:
                x = j + dx[d] * 2
                y = i + dy[d] * 2
                mx = j + dx[d]
                my = i + dy[d]
                if hasPeg(last, x, y) or not hasPeg(last, mx, my):
                    continue
                next = last
                next ^= 1 << (i * N + j)
                next ^= 1 << (y * N + x)
                next ^= 1 << (my * N + mx)
                if next not in positions:
                    positions[next] = last
                    q.put(next)
print()
printSolution(last)
```

Глава 10

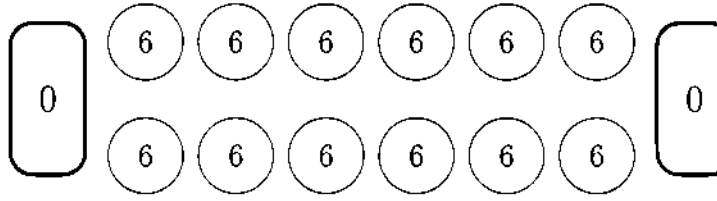
# Альфа-бета отсечение: ускоряем рекурсивный перебор

## 10.1. Игра «Калах»

Калах — это настольная игра из целого семейства игр манкала. Поэтому вариантов правил может быть несколько, у них даже могут быть собственные названия. Мы будем писать программу, играющую по правилам, описанным в статье «Калах» в Википедии.

Во всех вариантах поле для игры — это доска с углублениями (лунками), расположенными в два ряда по шесть лунок в каждом. Также с противоположных краёв доски расположены две большие лунки, которые называются «калах».

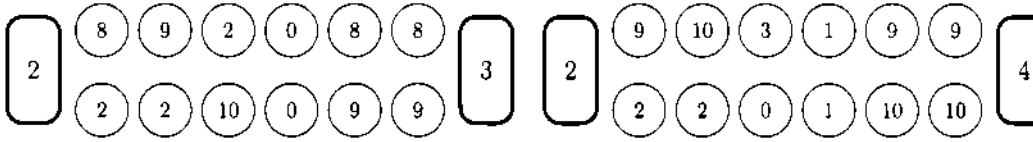
В лунки помещаются камни (шарики, семена и т.п.). Начинается игра с помещения одинакового числа камней в каждую лунку (но не в калахи). Обычно это по 4 или по 6 камней. Если число камней меньше 4, то игру слишком легко просчитывать. Позже мы это проверим на практике.




Каждому игроку принадлежит один ряд лунок перед ним и один калах, который расположен справа от этих лунок. Оба калаха расположены справа, потому что второй игрок смотрит на доску с противоположной стороны. Для него «право» — это «лево» для первого игрока.

Ход в калахе называется «посевом» (потому что в оригинале для игры иногда используются семена). Во время хода игрок выбирает одну из своих шести лунок, вынимает оттуда все камни (семена) и раскладывает их против часовой стрелки по одному во все последующие лунки, пропуская калах противника. Если делается ход из лунки, содержащей 13 и более камней, то в неё тоже будет попадать минимум один камень.

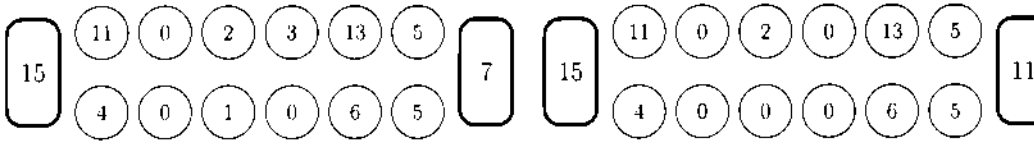
Вот первый («нижний») игрок берёт из лунки 10 камней и раскладывает их. Один из камней попадает в калах:




Камни можно захватывать (перемещать в калах) двумя способами. Первый — это попадание камня в калах в процессе посева. Из калаха камни уже достать никак нельзя. Второй способ захвата — это попадание последнего камня в пустую лунку в своём ряду. Тогда если противоположная лунка (в ряду соперника)

не пуста, то захватываются как все камни из этой лунки, так и свой последний камень.

Вот ещё пример. Опять ход первого игрока. Он ходит камнем из третьей лунки и захватывает 3 камня противника.




Если в конце хода камень попадает в калах, то игрок делает ещё один ход. Такие ходы могут повторяться неоднократно.

Игра заканчивается, если лунки одного из игроков опустеют. В этом случае оставшиеся камни захватывает игрок, которому они принадлежат, а победителем становится игрок, захвативший большее число камней. То есть чем больше камней попадёт в калах, тем лучше для владеющего им игрока.

### 10.1.1. Игровая позиция

Всего в игре есть 14 лунок (включая калахи) с камнями. То есть каждую лунку можно описать одним целым числом – количеством камней, которые там лежат.

Для хранения всех этих чисел можно просто создать массив из 14 ячеек. Но калах каждого игрока — это особенная лунка. Из него нельзя брать камни, он используется при определении победителя, в калах противника камни не попадают.

Поэтому если делать не единый массив, а два массива для наборов из обычных лунок и две переменные для калахов, обращаться с ними в некоторых ситуациях будет проще. С другой стороны, это усложнит посев. Ведь там мы просто движемся по кругу, раскладывая камни. То есть идём по ячейкам массива последовательно (конечно, если мы сопоставим ячейкам лунки против часовой стрелки).

Поэтому просто создадим массив, а чтобы проще было обращаться к калахам, зададим для их индексов в массиве именованные константы K1 и K2.

C++

```
#include <array>

const int N = 6;
const int K1 = N;
const int K2 = 2 * N + 1;
const int SZ = N * 2 + 2;

typedef std::array<int, SZ> Board;
Board board;
```

Python

```
N = 6
K1 = N
K2 = 2 * N + 1
SZ = N * 2 + 2

board = [0] * SZ
```

**Экран 1. Окончание партии в Калах**

```
Выберите лунку с камнями: 2
   6  5  4  3  2  1
  16  1  1  3  0  2
21                    13
   1  0  6  5  1  2
   1  2  3  4  5  6

Мой ход: 1
   6  5  4  3  2  1
  16  1  1  4  1  0
21                    13
   1  0  6  5  1  2
   1  2  3  4  5  6

Выберите лунку с камнями: 1
   6  5  4  3  2  1
  16  0  1  4  1  0
21                    15
   0  0  6  5  1  2
   1  2  3  4  5  6

Мой ход: 3
   6  5  4  3  2  1
  17  1  2  0  1  0
22                    15
   0  0  6  5  1  2
   1  2  3  4  5  6

Мой ход: 2
   6  5  4  3  2  1
  17  1  2  0  0  0
28                    15
   0  0  6  0  1  2
   1  2  3  4  5  6

Выберите лунку с камнями: 3
   6  5  4  3  2  1
  17  1  2  0  1  1
28                    16
   0  0  0  1  2  3
   1  2  3  4  5  6

Мой ход: 2
   6  5  4  3  2  1
  17  1  2  0  0  1
30                    16
   0  0  0  0  2  3
   1  2  3  4  5  6

Выберите лунку с камнями: 5
   6  5  4  3  2  1
  17  1  2  0  0  1
30                    17
   0  0  0  0  0  4
   1  2  3  4  5  6

Выберите лунку с камнями: 6
   6  5  4  3  2  1
   0  0  0  0  0  0
54                    18
   0  0  0  0  0  0
   1  2  3  4  5  6

Я выиграл!
```

Перед началом игры массив нужно инициализировать. «Разложим» во все обычные лунки по 6 камней. Снова понадобится несколько именованных констант, чтобы находить диапазоны индексов обычных лунок. Они пригодятся и в других частях программы, например, при проверке корректности хода игрока.

C++

```
const int First1 = 0;
const int Last1 = N - 1;
const int First2 = K1 + 1;
const int Last2 = K2 - 1;
const int S = 6;

int main()
{
  for (int i = 0 ; i < N ; ++i)
  {
    board[First1 + i] = S;
    board[First2 + i] = S;
  }
  ...
}
```

Python

```
First1 = 0
Last1 = N - 1
First2 = K1 + 1
Last2 = K2 - 1
S = 6

for i in range(N):
  board[First1 + i] = S
  board[First2 + i] = S
```

### 10.1.2. Главный игровой цикл

Игровой цикл очень простой. Сначала один игрок делает свой ход, потом другой. Перед каждым ходом выводится игровое поле, так будет удобнее отслеживать ход игры.

C++

```
int main()
{
  ...
  while (true)
  {
    // ход игрока
    printBoard();
    ...
    // ход компьютера
    printBoard();
    ...
  }
  ...
}
```

Python

```
...
while True:
  # ход игрока
  printBoard()
  ...
  # ход компьютера
  printBoard()
  ...
..
```

После завершения главного цикла (не смотрите на то, что он бесконечный, мы позже это исправим) нужно вывести сообщение о том, кто победил. Для этого сравниваем содержимое калахов. И снова нам пригодились именованные константы с индексами калахов в массиве-игровом поле.

C++
```
printBoard();

// определение победителя
if (board[K1] > board[K2])
{
  std::cout << "Вы победили! \n";
}
else if (board[K2] > board[K1])
{
  std::cout << "Я выиграл! \n";
}
else
{
  std::cout << "Ничья. \n";
}
```

Python
```
printBoard()

# определение победителя
if board[K1] > board[K2]:
  print('Вы победили!')
elif board[K2] > board[K1]:
  print('Я выиграл!')
else:
  print('Ничья.')
```

### 10.1.3. Вывод игрового поля

Игровое поле стоит выводить так, чтобы это было похоже на настоящую доску с лунками. По крайней мере, расположение лунок должно быть понятным:

```
   6  5  4  3  2  1
  16  1  1  4  1  0
21                    13
   1  0  6  5  1  2
   1  2  3  4  5  6
```

Для каждой лунки выводится число лежащих там камней. Кроме того, для обычных лунок печатается номер от 1 до 6, чтобы игрок не пересчитывал лунки, когда делает ход.

Параметров у функции не будет, так как выводится всегда только основное поле:

C++
```
#include <iostream>
#include <iomanip>

void printBoard()
{
  ...
  std::cout << "\n\n";
}
```

Python
```
def printBoard():
  ...
  print()
  print()
```

Сначала функция выводит номера лунок компьютера, потом содержимое его лунок, потом содержимое калахов, а в конце уже выводятся лунки игрока и их номера.

C++

```
std::cout << " ";
for (int i = N ; i >= 1 ; --i)
{
  std::cout << std::setw(3) << i;
}
std::cout << "\n ";
for (int i = Last2 ; i >= First2 ; --i)
{
  std::cout << std::setw(3) << board[i];
}
std::cout << "\n" << std::setw(2) <<
↪  board[K2]
  << "                    " << board[K1] <<
  ↪  "\n ";
for (int i = First1 ; i <= Last1 ; ++i)
{
  std::cout << std::setw(3) << board[i];
}
std::cout << "\n ";
for (int i = 1 ; i <= N ; ++i)
{
  std::cout << std::setw(3) << i;
}
```

Python

```
print(' ', end='')
for i in range(N, 0, -1):
  print(f'{i:>3}', end='')
print()
print(' ', end='')
for i in range(Last2, First2 - 1, -1):
  print(f'{board[i]:>3}', end='')
print()
print(f'{board[K2]:>2}
↪  {board[K1]}')
print(' ', end='')
for i in range(First1, Last1 + 1):
  print(f'{board[i]:>3}', end='')
print()
print(' ', end='')
for i in range(1, N + 1):
  print(f'{i:>3}', end='')
```

## 10.1.4. Чередование ходов

Калах отличается от других игр (вроде реверси и крестиков-ноликов) тем, что игроки могут делать повторные ходы. Поэтому если ходом (всей цепочкой ходов одного игрока) будет заниматься один вызов функции (`playerMove` или `computerMove`), то внутри придётся сделать цикл, а в нём ещё и выводить игровое поле.

Другой вариант, если функция делает выбор лунки и один посев, а потом возвращает значение, позволяющее определить, нужен ли повторный ход. Тогда в главном игровом цикле будут ещё два цикла (для игрока и для компьютера).

Кажется, что второй вариант выглядит поэлегантнее. Тогда в основном цикле появятся вот такие фрагменты:

C++

```
do
{
  printBoard();
  res = playerMove();
}
while (res == ...);
```

Python

```
res = ...
while res == ...:
  printBoard()
  res = playerMove()
```

Какие возможны возвращаемые значения? Хватит ли для них булевого типа? Во-первых, после посева может или не может быть повторный ход. А во-вторых, после хода любого из игроков партия может закончиться. Получается, что ва-

риантов всего три и булевого типа не хватит. Сделаем для всех вариантов окончания хода именованные константы:

C++
```
enum MoveResult
{
  MoveNext,
  MoveAgain,
  MoveEnd,
};
```

Python
```
MoveNext = 0
MoveAgain = 1
MoveEnd = 2
```

Ну и теперь можно доделать наши мини-циклы для хода игрока и для хода компьютера:

C++
```
while (true)
{
  MoveResult res;
  do
  {
    printBoard();
    res = playerMove();
  }
  while (res == MoveAgain);
  if (res == MoveEnd)
  {
    break;
  }
  ...
}
```

Python
```
while True:
  res = MoveAgain
  while res == MoveAgain:
    printBoard()
    res = playerMove()
  if res == MoveEnd:
    break
  ...
```

### 10.1.5. Ход игрока

Как всегда, начнём с функции, отвечающей за ход игрока. Её задача — запросить номер лунки, а потом разложить из неё камни. Конечно же, корректность ввода тоже нужно проверить.

C++
```
MoveResult playerMove()
{
  int m;
  do
  {
    std::cout << "Выберите лунку с
    ↪  камнями: ";
    std::cin >> m;
    --m;
  }
  while (m < 0 || m >= N || !board[First1
  ↪  + m]);
  return makeMove(First1 + m);
}
```

Python
```
def playerMove():
  m = -1
  while m < 0 or m >= N or board[First1 +
  ↪  m] == 0:
    m = int(input('Выберите лунку с
    ↪  камнями: '))
    m -= 1
  return makeMove(board, First1 + m)
```

Конечно же, константа `First1` не очень-то нам нужна (ведь она равна нулю). Она здесь присутствует лишь для того, чтобы показать, что введённое `m` — это номер лунки на доске, который ещё надо преобразовать в индекс в массиве. Тем более что такой же код будет и для хода компьютера уже с ненулевой константой `First2` — получится красиво и единообразно.

### 10.1.6. Функция посева

С посевом всё сложнее, чем кажется:

- Сначала надо разложить камни.
- Потом проверить возможность захвата.
- После этого может оказаться, что игра закончилась, потому что у одного из игроков нет камней.
- Ну и наконец, если игра не окончилась, нужно проверить, получил ли игрок право на повторный ход.

Кроме всего прочего, функция должна работать как для ходов игрока, так и для ходов компьютера.

**C++**

```
MoveResult makeMove(int m)
{
  // непосредственно посев
  ...
  // захват камней
  ...
  // проверка окончания игры
  ...
  // проверка повторного хода
  ...
  return MoveNext;
}
```

**Python**

```
def makeMove(m):
  # непосредственно посев
  ...
  # захват камней
  ...
  # проверка окончания игры
  ...
  # проверка повторного хода
  ...
  return MoveNext
```

### 10.1.7. Посев камней

Чтобы разложить камни по лункам, нужно сначала «забрать» их из исходной лунки, а потом двигаться по кругу, раскладывая по одному камню (то есть увеличивая каждую ячейку массива на 1).

Только нужно учесть, что калах оппонента пропускается. Для этого сначала вычислим номер игрока (0 или 1, это число ещё пригодится позднее), а потом запишем в переменную `kalahOpp` номер ячейки с калахом противника.

C++
```
int s = board[m];
board[m] = 0;
int cur = m;
int player = m <= Last1 ? 0 : 1;
int kalahOpp = player == 0 ? K2 : K1;
while (s)
{
  cur = (cur + 1) % SZ;
  if (cur != kalahOpp)
  {
    --s;
    ++board[cur];
  }
}
```

Python
```
s = board[m]
board[m] = 0
cur = m
player = 0 if m <= Last1 else 1
kalahOpp = K2 if player == 0 else K1
while s > 0:
  cur = (cur + 1) % SZ
  if cur != kalahOpp:
    s -= 1
    board[cur] += 1
```

Номер очередной ячейки вычисляется с использованием остатка от деления на размер массива. Так, после помещения камня в ячейку 13, мы переходим к ячейке 0.

### 10.1.8. Захват камней

Сначала нужно определить, попал ли последний камень в собственную лунку игрока (и при этом это не калах). Потом найти противоположную лунку (переменная opp), а после всего переместить захваченные камни в калах игрока.

Тут-то и пригодится переменная `player`, потому что захват происходит, только если номер игрока, владеющего последней лункой (переменная `lastPlayer`), совпадает с номером текущего игрока.

C++
```
int lastPlayer = cur <= Last1 ? 0 : 1;
int opp = SZ - 2 - cur;
int kalah = player == 0 ? K1 : K2;
if (cur != kalah && player == lastPlayer
↪  && board[cur] == 1 && board[opp])
{
  board[kalah] += board[cur] +
  ↪  board[opp];
  board[cur] = 0;
  board[opp] = 0;
}
```

Python
```
lastPlayer = 0 if cur <= Last1 else 1
opp = SZ - 2 - cur
kalah = K1 if player == 0 else K2
if cur != kalah and player == lastPlayer
↪  and board[cur] == 1 and board[opp]:
  board[kalah] += board[cur] + board[opp]
  board[cur] = 0
  board[opp] = 0
```

### 10.1.9. Проверка окончания игры

Проверить, что игра закончилась, несложно. Считаем, сколько у каждого из игроков осталось камней в лунках, а потом сравниваем эти числа с нулём.

Если хотя бы одна из сумм равняется нулю, перемещаем оставшиеся камни в калах. Проще всего перемещать камни обоих игроков сразу. Просто в одном

случае это будут остатки камней из лунок, а во втором — просто нули (которые на содержимое калаха повлиять не могут).

И, конечно же, в этом случае нужно не забыть вернуть из функции признак того, что игра закончилась.

C++

```
int sum1 = 0;
int sum2 = 0;
for (int i = 0 ; i < N ; ++i)
{
  sum1 += board[First1 + i];
  sum2 += board[First2 + i];
}
if (!sum1 || !sum2)
{
  for (int i = 0 ; i < N ; ++i)
  {
    board[K1] += board[First1 + i];
    board[K2] += board[First2 + i];
    board[First1 + i] = 0;
    board[First2 + i] = 0;
  }
  return MoveEnd;
}
```

Python

```
sum1 = 0
sum2 = 0
for i in range(N):
  sum1 += board[First1 + i]
  sum2 += board[First2 + i]
if sum1 == 0 or sum2 == 0:
  for i in range(N):
    board[K1] += board[First1 + i]
    board[K2] += board[First2 + i]
    board[First1 + i] = 0
    board[First2 + i] = 0
  return MoveEnd
```

### 10.1.10. Проверка повторного хода

После всех сложных манипуляций остаётся самая простая часть — проверить, положен ли игроку повторный ход. Это происходит, если индекс последней заполненной лунки совпадает с индексом калаха, принадлежащего игроку.

C++

```
if (cur == kalah)
{
  return MoveAgain;
}
```

Python

```
if cur == kalah:
  return MoveAgain
```



С появлением функции `makeMove` уже можно «играть». Брать с доски камни и раскладывать их по другим лункам. Только вот компьютер ответить не сможет. Но ничего, скоро мы это исправим.

### 10.1.11. Случайный ход компьютера

Функция для случайного хода компьютера мало отличается от функции `playerMove` Только вместо ввода номера лунки с клавиатуры мы генерируем случайный индекс в массиве. Ну и, конечно, тут используется константа `First2`, обозначающая начало серии лунок компьютера в массиве.

C++

```
MoveResult computerMove()
{
  int m;
  do
  {
    m = First2 + rand() % N;
  }
  while (!board[m])

  std::cout << "Мой ход: " << m - First2
  ↪  + 1 << "\n";
  return makeMove(m);
}
```

Python

```
import random

def computerMove():
  while True:
    m = First1 + random.randint(0, N - 1)
    if board[m] > 0:
      break

  print(f'Мой ход {m - First2 + 1}')
  return makeMove(m)
```

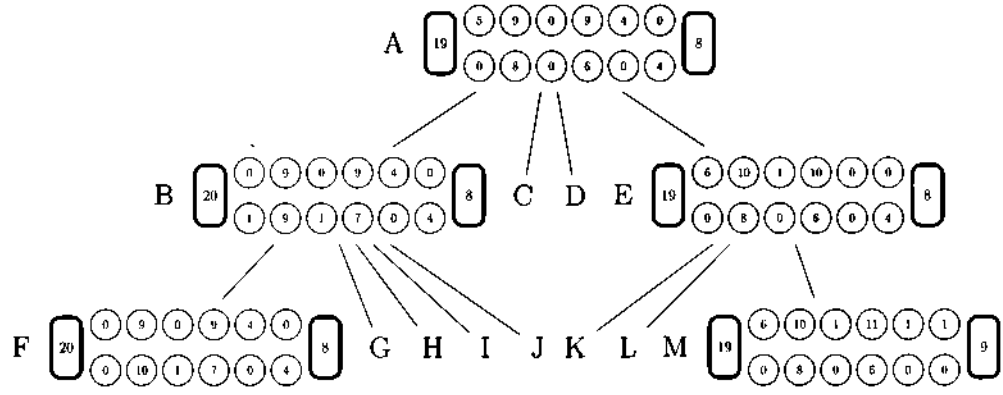
Наконец-то можно поиграть! Скорее протестируйте программу. Легко ли обыграть компьютер, делающий случайные ходы?

### 10.1.12. Рекурсивный перебор

Так как глава, в которой мы играем в Калах, посвящена усовершенствованию рекурсивного перебора, то не будем придумывать никаких эвристик, а сразу перебором и займёмся.

Рекурсивный перебор с возвратом проверяет все возможные ходы, делая для каждого такой же перебор с точки зрения противника, там тоже перебирая все возможные ходы, и так далее.

Всего в каждой позиции возможно не более 6 вариантов хода, по числу лунок у игрока:




Как и в крестиках-ноликах $4 \times 4$, перебор начинается с той позиции, которая досталась компьютеру. Он должен попробовать все доступные ходы и получить для каждого оценку, насколько этот ход хорош. А потом выбрать ход с наилучшей оценкой. Сама оценка будет вычисляться с помощью рекурсивной функции `checkMove`, а пока напишем цикл для её вызова:

C++

```
MoveResult computerMove()
{
  int m = -1;
  int max = ...;
  for (int i = First2 ; i <= Last2 ; ++i)
  {
    if (board[i])
    {
      int r = checkMove(board, 1, i, 10);
      if (r > max)
      {
        m = i;
        max = r;
      }
    }
  }

  std::cout << "Мой ход: " << m - First2
  ↪  + 1 << "\n";
  return makeMove(board, m);
}
```

Python

```
def computerMove():
  m = -1
  max = ...
  for i in range(First2, Last2 + 1):
    if board[i] > 0:
      r = checkMove(board, 1, i, 10)
      if r > max:
        m = i
        max = r

  print(f'Мой ход {m - First2 + 1}')
  return makeMove(board, m)
```

Что записать в переменную max, пока не ясно. Она должна принимать значение, которое меньше любой возможной оценки. Так как способ оценки позиции ещё не придуман, то минимум мы выбрать не можем.

У функции checkMove будет 4 параметра. Первый — это игровая позиция. Так как она будет меняться, нужно её копировать в виде аргумента функции. В крестиках-ноликах мы могли этого не делать, а просто затирать уже проверенный ход. Здесь же ход меняет много ячеек, и поэтому проще всего создать дубликат игровой позиции, который будет меняться, а потом его уничтожить (точнее, он уничтожится сам, когда станет не нужен).

Второй параметр функции — это номер игрока, число 0 или 1. Без этого аргумента не обойтись, так как по мере обхода дерева игры ход делают оба игрока, у каждого свой набор лунок (а также будет отличаться способ оценки позиции).

Третий параметр -- это лунка, из которой берутся камни. Функцию makeMove удобнее вызывать уже внутри, потому что после неё игра может закончиться, может потребоваться повторный ход и так далее. Все эти варианты мы уже разберём в checkMove, чтобы не дублировать их здесь.

И последний параметр — это глубина перебора. Игра может длиться довольно долго (уж никак не меньше 10 ходов). А при полном переборе 10 возможных ходов может получиться до $6^{10} = 60\,466\,176$. Это не так уж мало, результатов такой проверки придётся ждать десятки секунд (для программы на C++). Поэтому глубину перебора придётся ограничивать. Попробуем начать с числа 10, а если не понравится, поменяем его на большее или меньшее.

### 10.1.13. Оценка позиции

В книге [11] предлагается использовать такую оценку позиции: число камней в калахе компьютера минус число камней в калахе игрока. Из калаха камни не забираются, поэтому такая оценка неплохо показывает приближение к победе (или проигрышу).

Во время перебора один из игроков должен максимизировать оценку в своей стратегии, а второй – минимизировать. Из-за этого в литературе игроков называют Макс и Мин, в зависимости от того максимизирует или минимизирует ли он оценку позиции.

Для определённости у нас максимизировать оценку позиции будет компьютер. Всё же мы программируем его ход, а выигрыш ассоциируется с большим числом. А игрок в представлении компьютера будет оценку минимизировать. Конечно, в реальности компьютер не может быть в этом уверен, ведь человек может ходить как угодно. Но в этом случае он проиграет ещё быстрее.

Получается, что в качестве недостижимого минимального значения оценки позиции можно выбрать ситуацию, когда все камни попали в калах игрока. Такая позиция невозможна (любой дебютный ход помещает один камень в калах). Поэтому такой минимум будет соответствовать значению `-TOTAL`, если из 0 (камни в калахе компьютера) вычесть всю сумму камней в игре (которая потенциально может попасть в калах игрока).

### 10.1.14. Перебор позиций

Суть перебора в том, чтобы выбрать ход, который приведёт к позиции с наилучшей оценкой. Но сначала нужно доделать ход, «посеять» камни, которые уже выбраны в вызывающей функции. Если игра при этом заканчивается или достигнута максимальная глубина перебора, нужно вернуть из функции оценку достигнутой позиции. Это будет разница между числом камней в калахах компьютера и человека.

C++

```
int checkMove(Board board, int player,
↪  int m, int depth)
{
    MoveResult mr = makeMove(board, m);
    if (mr == MoveEnd || depth == 0)
    {
        // оценка позиции
        return board[K2] - board[K1];
    }
    ...
}
```

Python

```
def checkMove(brd, player, m, depth):
  board = brd.copy()
  mr = makeMove(board, m)
  if mr == MoveEnd or depth == 0:
    # оценка позиции
    return board[K2] - board[K1]
  ...
```

Переменная `board` передаётся по значению. То есть у функции будет собственная копия, которая может изменяться. Это нужно для того, чтобы перемещать камни, а потом возвращаться в исходное состояние, уничтожая эту переменную.

Переменная `board` — это копия параметра `brd`. В Python нельзя передать по значению сложный объект, а функции нужна собственная копия доски, которая будет изменяться. Это нужно для того, чтобы перемещать камни, а потом возвращаться в исходное состояние, уничтожая эту переменную.

Дальше нужно определить, кто будет ходить следующим, чтобы перебирать его ходы. Обычно это не тот игрок, для которого делался текущий ход, то есть `1 - player` (0 вместо 1 и наоборот). Но если последний камень попал в калах, текущий игрок не меняется. Тогда функция `makeMove` вернёт значение `MoveAgain`.

C++

```
int nextPlayer = 1 - player;
if (mr == MoveAgain)
{
  nextPlayer = player;
}
```

Python

```
nextPlayer = 1 - player
if mr == MoveAgain:
  nextPlayer = player
```

Потом перебираются ходы. Всего у игрока $N$ лунок, для каждой, где есть камни, делается рекурсивный вызов `checkMove`, но уже с меньшей оставшейся глубиной перебора.

C++

```
for (int i = 0 ; i < N ; ++i)
{
  int m = i + (nextPlayer ? First2 :
  ↪  First1);
  if (board[m])
  {
    checkMove(board, nextPlayer, m, depth
    ↪  - 1);
  }
}
```

Python

```
for i in range(N):
  m = i + (First2 if nextPlayer > 0 else
  ↪  First1)
  if board[m] > 0:
    checkMove(board, nextPlayer, m, depth
    ↪  - 1)
```

Но ведь рекурсивный вызов функции возвращает значение, которое нужно использовать, чтобы получить оценку для текущего хода. Ход будет хорошим для компьютера, если оценка большая, и для человека, если оценка маленькая.

Смысл всего этого такой: функция `checkMove` для всех ходов игрока вернёт оценки позиций. Так как игрок хотел бы получить больше камней в финальной позиции, он «выберет» (то есть мы думаем, что выберет) такой ход, где это значение минимально.

Компьютер, видя все эти варианты, постарается поставить человека в такую позицию, где его выбор из минимумов получится наихудшим для игрока. То есть максимальным. Значит, у компьютера будет больше шансов выиграть.

Окончательный выбор максимума мы делаем в функции `computerMove`. Вот аналогичным образом работает и выбор хода в `checkMove`, только в итоге выбирается не сам ход, а оценка соответствующей позиции.

C++

```
int res = nextPlayer == 0 ? TOTAL :
↪  -TOTAL;
for (int i = 0 ; i < N ; ++i)
{
  int m = i + (nextPlayer ? First2 :
  ↪  First1);
  if (board[m])
  {
    int r = checkMove(board, nextPlayer,
    ↪  m, depth - 1);
    // игрок минимизирует оценку
    if (nextPlayer == 0 && r < res)
    {
      res = r;
    }
    // компьютер максимизирует оценку
    if (nextPlayer == 1 && r > res)
    {
      res = r;
    }
  }
}
return res;
```

Python

```
res = TOTAL if nextPlayer == 0 else
↪  -TOTAL
for i in range(N):
  m = i + (First2 if nextPlayer > 0 else
  ↪  First1)
  if board[m] > 0:
    r = checkMove(board, nextPlayer, m,
    ↪  depth - 1)
    # игрок минимизирует оценку
    if nextPlayer == 0 and r < res:
      res = r
    # компьютер максимизирует оценку
    if nextPlayer == 1 and r > res:
      res = r
return res
```



У нас получился настоящий искусственный интеллект, который уже не так-то просто обыграть. Какая получается максимальная глубина хода, при которой компьютер «думает» не слишком долго?

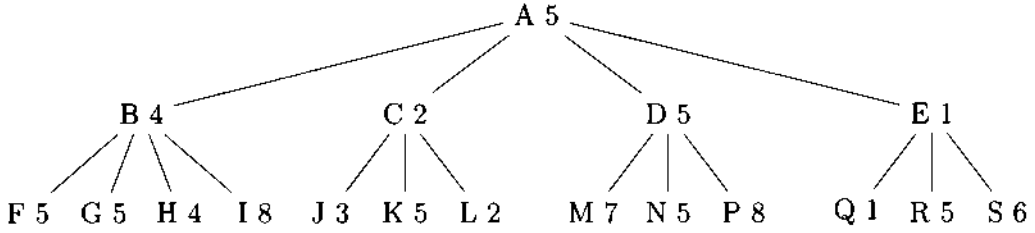
### 10.1.15. Альфа-бета отсечение

Полный перебор это хорошо. Все ходы и позиции проверены, поэтому компьютер всегда делает наилучший выбор, но глубина просмотра ограничена. С глубиной поиска 10 ходов играть вполне комфортно, долго ждать ответа не надо, но партия длится ходов 30–40. Так что для идеального выбора глубину перебора компьютером хочется сделать побольше, и он станет совсем непобедимым.

В пару раз ускорить перебор можно, если немного улучшить манипуляции с камнями, но гораздо лучше придумать, как отбрасывать лишние позиции. Например, если вместо 6 ходов будет выбор из 5, то для глубины перебора 10 придётся просмотреть уже $5^{10} = 9\,765\,625$ позиций, а это будет работать сразу в 6 раз быстрее, чем раньше.

Как же находить такие бесперспективные позиции, чтобы их отбрасывать? Давайте снова посмотрим на дерево перебора. В позиции $A$ ход делает компьютер, поэтому выбирает максимум из $B$, $C$, $D$, $E$. А в них уже ход делает человек и выбирает минимум.

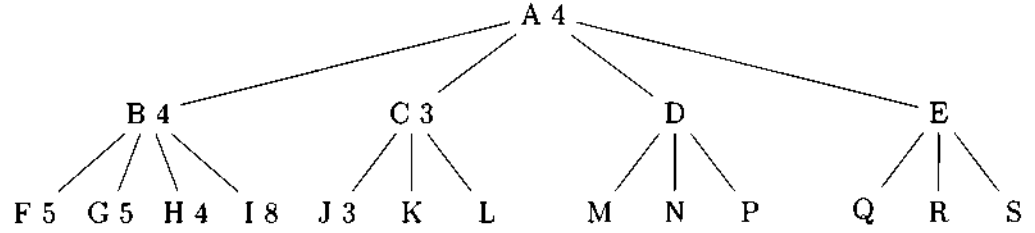
Вот дерево ходов с уже просчитанными оценками позиций:




Нужно сделать так, чтобы пропускать некоторые вершины, но при этом получить ту же оценку для позиции $A$.

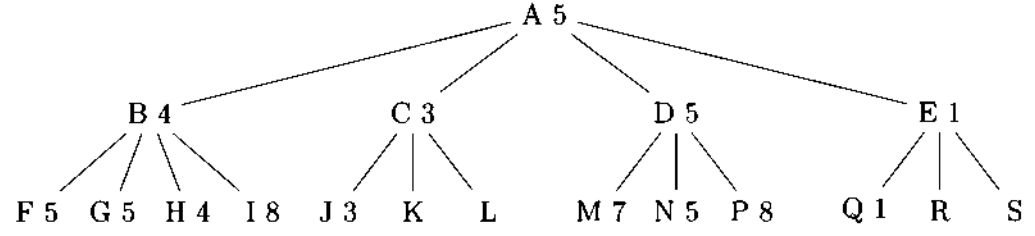
При анализе позиции $A$ её стартовая оценка равняется $-TOTAL$. Предположим, что переборная функция `checkMove` уже изучила все ходы, начиная с $B$, и текущая оценка позиции $A$ становится 4.

Теперь функция приступает к анализу позиции $C$ и ходов из неё. Первый ход — в $J$, после чего оценка позиции $C$ становится равной 3:




В позиции $C$ очередь хода за человеком. Значит, `checkMove` будет за него выбирать ходы, ведущие к позициям с минимальной оценкой. Сейчас оценка позиции равна 3, значит больше 3 она уже не станет. И в функцию, анализирующую позицию $A$, вернётся число, не большее 3. А так как текущая оценка $A$ уже равняется 4, это означает, что окончательная оценка позиции $C$ уже точно не повлияет на результат, ведь она может только уменьшаться.

Поэтому нет никакого смысла продолжать перебор и уточнять оценку $C$. Можно сэкономить время и сразу перейти к оценке $D$. Рассуждая аналогичным образом, получаем такую картину:



Получается, что из 18 позиций на картинке мы рассмотрели лишь 14. А ведь тут изучались только два хода в глубину; если бы их было десять, число отброшенных позиций (узлов дерева) было бы ещё больше, ведь из каждой пропущенной вершины «росло» бы большое поддерево.

Чтобы реализовать такое отсечение, рекурсивной функции понадобится дополнительный параметр. Назовём его «альфа». Он будет обновляться каждый раз, когда увеличивается оценка текущей позиции. А когда ищется минимальная оценка, то если получается значение, меньшее, чем альфа, работа функции завершается.

C++

```
int checkMove(Board board, int player,
↪  int m, int depth, int alpha)
{
  ...
  int r = checkMove(board, nextPlayer, m,
  ↪  depth - 1, alpha);
  // игрок минимизирует оценку
  if (nextPlayer == 0)
  {
    res = std::min(r, res);
    if (res < alpha)
    {
      break;
    }
  }
  // компьютер максимизирует оценку
  if (nextPlayer == 1)
  {
    res = std::max(r, res);
    alpha = std::max(alpha, res);
  }
  ...
}
```

Python

```
def checkMove(brd, player, m, depth,
↪  alpha):
  ...
  r = checkMove(board, nextPlayer, m,
  ↪  depth - 1, alpha)
  # игрок минимизирует оценку
  if nextPlayer == 0:
    res = min(r, res)
    if res < alpha:
      break
    # компьютер максимизирует оценку
    if nextPlayer == 1:
      res = max(r, res)
      alpha = max(alpha, res)
  ...
```

Теперь фрагменты `checkMove`, обрабатывающие ходы игрока и ходы компьютера, отличаются. А ведь ту же логику отсечения можно применить и к ходу человека. Если анализируемые в глубину ходы получаются слишком хорошими для компьютера, а не для игрока, то их можно не анализировать, отбрасывая целые ветви дерева перебора. Параметр с такой оценкой назовём «бета»:

C++

```
int checkMove(Board board, int player,
↪  int m, int depth, int alpha, int
↪  beta)
{
  ...
}
```

Python

```
def checkMove(brd, player, m, depth,
↪  alpha, beta):
  ...
```

Он может только уменьшаться, когда обнаруживается лучший ход для игрока:

C++

```
...
int r = checkMove(board, nextPlayer, m,
↪  depth - 1, alpha, beta);
// игрок минимизирует оценку
if (nextPlayer == 0)
{
  res = std::min(r, res);
  if (res < alpha)
  {
    break;
  }
  beta = std::min(beta, res);
}
// компьютер максимизирует оценку
if (nextPlayer == 1)
{
  res = std::max(r, res);
  if (res > beta)
  {
    break;
  }
  alpha = std::max(alpha, res);
}
...
```

Python

```
...
r = checkMove(board, nextPlayer, m, depth
↪  - 1, alpha, beta)
# игрок минимизирует оценку
if nextPlayer == 0:
  res = min(r, res)
  if res < alpha:
    break
  beta = min(beta, res)
# компьютер максимизирует оценку
if nextPlayer == 1:
  res = max(r, res)
  if res > beta:
    break
  alpha = max(alpha, res)
...
```

Сам перебор готов, но нужно передать в функцию начальные значения параметров `alpha` и `beta`. Так как первый только растёт, сначала он будет равен минимальному значению. А второй, соответственно, максимальному.

C++

```
MoveResult computerMove()
{
  ...
  int r = checkMove(board, 1, i, 12,
  ↪  -TOTAL, TOTAL);
  ...
}
```

Python

```
def computerMove():
  ...
  r = checkMove(board, 1, i, 10, -TOTAL,
  ↪  TOTAL)
  ...
```

Теперь программа играет ещё лучше. Правда, чтобы в этом убедиться, нужно очень хорошо играть самому.

### 10.1.16. Задания для самостоятельной работы

**1.** Протестируйте корректность работы альфа-бета отсечения. Делайте перебор дважды — с помощью обычной рекурсивной функции и с помощью функции с отсечением. Выбранные ходы должны совпадать. Конечно же, если глубина перебора будет одинаковой.

**2.** Замените функцию `playerMove` на функцию обычного перебора, без отсечения. Теперь разные алгоритмы будут играть сами с собой, и можно определить, насколько альфа-бета отсечение лучше.

**Листинг 10.1. kalah.cpp**

```
#include <iostream>
#include <iomanip>
#include <array>

const int N = 6;
const int K1 = N;
const int K2 = 2 * N + 1;
const int First1 = 0;
const int Last1 = N - 1;
const int First2 = K1 + 1;
const int Last2 = K2 - 1;
const int SZ = N * 2 + 2;
const int S = 6;
const int TOTAL = S * N * 2;

typedef std::array<int, SZ> Board;

Board board;

enum MoveResult
{
    MoveNext,
    MoveAgain,
    MoveEnd,
};

void printBoard()
{
    std::cout << " ";
    for (int i = N ; i >= 1 ; --i)
    {
        std::cout << std::setw(3) << i;
    }
    std::cout << "\n ";
    for (int i = Last2 ; i >= First2 ; --i)
    {
        std::cout << std::setw(3) << board[i];
    }
    std::cout << "\n" << std::setw(2) << board[K2]
        << "                  " << board[K1] << "\n ";
```

```
    for (int i = First1 ; i <= Last1 ; ++i)
    {
        std::cout << std::setw(3) << board[i];
    }
    std::cout << "\n ";
    for (int i = 1 ; i <= N ; ++i)
    {
        std::cout << std::setw(3) << i;
    }
    std::cout << "\n\n";
}

MoveResult makeMove(Board &board, int m)
{
    // посев камней
    int s = board[m];
    board[m] = 0;
    int cur = m;
    int player = m <= Last1 ? 0 : 1;
    int kalahOpp = player == 0 ? K2 : K1;
    while (s)
    {
        cur = (cur + 1) % SZ;
        if (cur != kalahOpp)
        {
            --s;
            ++board[cur];
        }
    }
    // захват камней противника
    int lastPlayer = cur <= Last1 ? 0 : 1;
    int opp = SZ - 2 - cur;
    int kalah = player == 0 ? K1 : K2;
    if (cur != kalah && player == lastPlayer && board[cur] == 1 &&
    ↪ board[opp])
    {
        // обновить калах
        board[kalah] += board[cur] + board[opp];
        // очистить лунки
        board[cur] = 0;
        board[opp] = 0;
```

```
    }
    // проверка, стал ли ход последним
    int sum1 = 0;
    int sum2 = 0;
    for (int i = 0 ; i < N ; ++i)
    {
        sum1 += board[First1 + i];
        sum2 += board[First2 + i];
    }
    if (!sum1 || !sum2)
    {
        for (int i = 0 ; i < N ; ++i)
        {
            board[K1] += board[First1 + i];
            board[K2] += board[First2 + i];
            board[First1 + i] = 0;
            board[First2 + i] = 0;
        }
        return MoveEnd;
    }
    // повторный ход
    if (cur == kalah)
    {
        return MoveAgain;
    }

    return MoveNext;
}

MoveResult playerMove()
{
    int m;
    do
    {
        std::cout << "Выберите лунку с камнями: ";
        std::cin >> m;
        --m;
    }
    while (m < 0 || m >= N || !board[First1 + m]);
    return makeMove(board, First1 + m);
}
```

```
int checkMove(Board board, int player, int m, int depth, int alpha,
↪  int beta)
{
    MoveResult mr = makeMove(board, m);
    if (mr == MoveEnd || depth == 0)
    {
        // оценка позиции
        return board[K2] - board[K1];
    }
    int nextPlayer = 1 - player;
    if (mr == MoveAgain)
    {
        nextPlayer = player;
    }
    int res = nextPlayer == 0 ? TOTAL : -TOTAL;
    for (int i = 0 ; i < N ; ++i)
    {
        int m = i + (nextPlayer ? First2 : First1);
        if (board[m])
        {
            int r = checkMove(board, nextPlayer, m, depth - 1, alpha,
            ↪  beta);
            // игрок минимизирует оценку
            if (nextPlayer == 0)
            {
                res = std::min(r, res);
                if (res < alpha)
                {
                    break;
                }
                beta = std::min(beta, res);
            }
            // компьютер максимизирует оценку
            if (nextPlayer == 1)
            {
                res = std::max(r, res);
                if (res > beta)
                {
                    break;
                }
```

```
                alpha = std::max(alpha, res);
            }
        }
    }
    return res;
}

MoveResult computerMove()
{
    int m = -1;
    int max = -TOTAL;
    for (int i = First2 ; i <= Last2 ; ++i)
    {
        if (board[i])
        {
            int r = checkMove(board, 1, i, 12, -TOTAL, TOTAL);
            if (r > max)
            {
                m = i;
                max = r;
            }
        }
    }

    std::cout << "Мой ход: " << m - First2 + 1 << "\n";
    return makeMove(board, m);
}

int main()
{
    for (int i = 0 ; i < N ; ++i)
    {
        board[First1 + i] = S;
        board[First2 + i] = S;
    }
    while (true)
    {
        MoveResult res;
        do
        {
            printBoard();
```

```
            res = playerMove();
        }
        while (res == MoveAgain);
        if (res == MoveEnd)
        {
            break;
        }
        do
        {
            printBoard();
            res = computerMove();
        }
        while (res == MoveAgain);
        if (res == MoveEnd)
        {
            break;
        }
    }
    printBoard();
    if (board[K1] > board[K2])
    {
        std::cout << "Вы победили! \n";
    }
    else if (board[K2] > board[K1])
    {
        std::cout << "Я выиграл! \n";
    }
    else
    {
        std::cout << "Ничья. \n";
    }
}
```

**Листинг 10.2. kalah.py**

```
#!/usr/bin/python3

N = 6
K1 = N
K2 = 2 * N + 1
First1 = 0
```

```
Last1 = N - 1
First2 = K1 + 1
Last2 = K2 - 1
SZ = N * 2 + 2
S = 6
TOTAL = S * N * 2

board = [0] * SZ

MoveNext = 0
MoveAgain = 1
MoveEnd = 2

def printBoard():
    print('  ', end='')
    for i in range(N, 0, -1):
        print(f'{i:>3}', end='')
    print()
    print('  ', end='')
    for i in range(Last2, First2 - 1, -1):
        print(f'{board[i]:>3}', end='')
    print()
    print(f'{board[K2]:>2}                    {board[K1]}')
    print('  ', end='')
    for i in range(First1, Last1 + 1):
        print(f'{board[i]:>3}', end='')
    print()
    print('  ', end='')
    for i in range(1, N + 1):
        print(f'{i:>3}', end='')
    print()
    print()

def makeMove(board, m):
    # посев камней
    s = board[m]
    board[m] = 0
    cur = m
    player = 0 if m <= Last1 else 1
    kalahOpp = K2 if player == 0 else K1
    while s > 0:
```

```
            cur = (cur + 1) % SZ
            if cur != kalahOpp:
                s -= 1
                board[cur] += 1
        # захват камней противника
        lastPlayer = 0 if cur <= Last1 else 1
        opp = SZ - 2 - cur
        kalah = K1 if player == 0 else K2
        if cur != kalah and player == lastPlayer and board[cur] == 1 and
        ↪ board[opp]:
            # обновить калах
            board[kalah] += board[cur] + board[opp]
            # очистить лунки
            board[cur] = 0
            board[opp] = 0
        # проверка, стал ли ход последним
        sum1 = 0
        sum2 = 0
        for i in range(N):
            sum1 += board[First1 + i]
            sum2 += board[First2 + i]
        if sum1 == 0 or sum2 == 0:
            for i in range(N):
                board[K1] += board[First1 + i]
                board[K2] += board[First2 + i]
                board[First1 + i] = 0
                board[First2 + i] = 0
            return MoveEnd
        # повторный ход
        if cur == kalah:
            return MoveAgain

        return MoveNext

def playerMove():
    m = -1
    while m < 0 or m >= N or board[First1 + m] == 0:
        m = int(input('Выберите лунку с камнями: '))
        m -= 1
    return makeMove(board, First1 + m)
```

```
def checkMove(brd, player, m, depth, alpha, beta):
    board = brd.copy()
    mr = makeMove(board, m)
    if mr == MoveEnd or depth == 0:
        # оценка позиции
        return board[K2] - board[K1]
    nextPlayer = 1 - player
    if mr == MoveAgain:
        nextPlayer = player
    res = TOTAL if nextPlayer == 0 else -TOTAL
    for i in range(N):
        m = i + (First2 if nextPlayer > 0 else First1)
        if board[m] > 0:
            r = checkMove(board, nextPlayer, m, depth - 1, alpha,
            ↪ beta)
            # игрок минимизирует оценку
            if nextPlayer == 0:
                res = min(r, res)
                if res < alpha:
                    break
                beta = min(beta, res)
            # компьютер максимизирует оценку
            if nextPlayer == 1:
                res = max(r, res)
                if res > beta:
                    break
                alpha = max(alpha, res)
    return res

def computerMove():
    m = -1
    max = -TOTAL
    for i in range(First2, Last2 + 1):
        if board[i] > 0:
            r = checkMove(board, 1, i, 10, -TOTAL, TOTAL)
            if r > max:
                m = i
                max = r

    print(f'Мой ход {m - First2 + 1}')
    return makeMove(board, m)
```

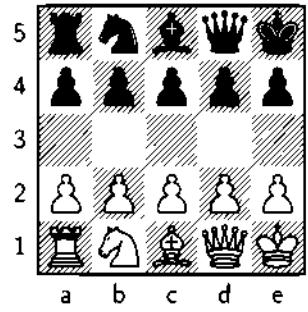
```
for i in range(N):
    board[First1 + i] = S
    board[First2 + i] = S

while True:
    res = MoveAgain
    while res == MoveAgain:
        printBoard()
        res = playerMove()
    if res == MoveEnd:
        break
    res = MoveAgain
    while res == MoveAgain:
        printBoard()
        res = computerMove()
    if res == MoveEnd:
        break
printBoard();
if board[K1] > board[K2]:
    print('Вы победили!')
elif board[K2] > board[K1]:
    print('Я выиграл!')
else:
    print('Ничья.')
```

## 10.2. Игра «Мини-шахматы»

Программы для игры в шахматы придумывают уже очень давно. Но для полноценного варианта игры нужно много памяти и процессорных ресурсов. А пользоваться всем этим должны хитроумные алгоритмы.

Так как мы изучаем алгоритмы попроще, то и шахматы лучше упростить. На отладку времени уйдёт меньше (потому что дерево ходов уменьшится), а с нужными идеями разберёмся быстрее.

В книге [2]упоминается, что хоть сколько-то играбельный вариант шахмат может быть на доске не меньше, чем $5 \times 5$:




Вот его и попробуем сделать. Тут есть все виды фигур, как и в обычных шахматах, так что в дальнейшем можно будет продолжать развивать программу, чтобы она смогла играть на доске $8 \times 8$.

Игровой процесс будет немного упрощённым: вместо мата королю противника нужно будет его взять другой фигурой. Ведь по существу мат и является такой позицией, при которой король не может избавиться от угрозы. Просто придётся делать дополнительный ход.

### 10.2.1. Игровая позиция

Игровая позиция в шахматах — это расположение фигур на доске. И, как всегда, есть два варианта хранить эту информацию. Первый – это записывать координаты всех фигур в массиве, списке или в чём-то ещё. Так можно легко обращаться к фигурам конкретного игрока.

Второй вариант — это хранить состояния всех клеток на доске. С этим подходом проще определять, куда фигура может переместиться и какие другие фигуры атаковать.

Так как мы планируем только перебирать варианты ходов, не сохраняя всё дерево, то объём памяти, используемой под игровую позицию, большого значения

**Экран 1. Фрагмент партии в мини-шахматы**

```
  abcde
5 rnbqk
4 ppppp
3 .....
2 PPPPP
1 RNBQK

Введите ваш ход: a2a3
  abcde
5 rnbqk
4 ppppp
3 P....
2 .PPPP
1 RNBQK

Мой ход: b4-a3

  abcde
5 rnbqk
4 p.ppp
3 p....
2 .PPPP
1 RNBQK

Введите ваш ход: b2a3
  abcde
5 rnbqk
4 p.ppp
3 P....
2 ..PPP
1 RNBQK

Мой ход: c4-c3
```

```
  abcde
5 rnbqk
4 p..pp
3 P.p..
2 ..PPP
1 RNBQK

Введите ваш ход: d2c3
  abcde
5 rnbqk
4 p..pp
3 P.P..
2 ..P.P
1 RNBQK

Мой ход: b5-c3

  abcde
5 r.bqk
4 p..pp
3 P.n..
2 ..P.P
1 RNBQK

Введите ваш ход: b1c3
  abcde
5 r.bqk
4 p..pp
3 P.N..
2 ..P.P
1 R.BQK

Мой ход: d4-c3
```

не имеет. А вот обработка хода проще во втором варианте, когда есть массив для хранения клеток поля.

Состояние каждой клетки будет представлено одним символом. Если она пустая, то это точка, а если нет, то буква, соответствующая фигуре, которая там расположена. Большие буквы будут для белых (за которых играет человек), а маленькие — для чёрных (за них играет компьютер).

Обозначать фигуры будем первой буквой их названия. Правда, конь и король тогда будут называться одинаково, поэтому придётся для коня использовать вторую букву:

- P (Pawn) — пешка;
- R (Rook) — ладья;
- N (kNight) — конь;
- B (Bishop) — слон;
- Q (Queen) — ферзь;
- K (King) — король.

**C++**

```
#include <array>
#include <string>

const int N = 5;

typedef std::array<std::string, N> Board;

Board board = {{
  "RNBQK",
  "PPPPP",
  ".....",
  "ppppp",
  "rnbqk"
}};
```

Горизонтали игровой доски хранятся в переменных строкового типа. Это почти обычный массив, но с возможностью вывести его на экран одним оператором. Строки занимают немного больше памяти, чем массив фиксированного размера, но в нашем случае это неважно.

**Python**

```
N = 5

board = [
  list('RNBQK'),
  list('PPPPP'),
  list('.....'),
  list('ppppp'),
  list('rnbqk')
]
```

Если строки игрового поля хранить в виде обычных строк символов, то фигуры нельзя будет передвигать, потому что строки в Python неизменяемые. Поэтому приходится преобразовывать все строки в списки (они же массивы, то есть `list`).

### 10.2.2. Главный игровой цикл

Игровой цикл здесь типовой. Стороны всегда ходят поочерёдно, игровое поле выводится после каждого хода. Можно было бы выводить поле и перед ходами,

но тогда пришлось бы добавить ещё по вызову **printBoard** перед завершением программы, чтобы игрок мог увидеть финальную позицию в игре. А так добавляется всего лишь один дополнительный вывод перед началом цикла.

**C++**

```
#include <iostream>

int main()
{
  printBoard();
  while (true)
  {
    playerMove();
    printBoard();
    if (...)
    {
      std::cout << "Вы выиграли!\n";
      break;
    }
    computerMove();
    printBoard();
    if (...)
    {
      std::cout << "Я победил!\n";
      break;
    }
  }
}
```

**Python**

```
printBoard()
while True:
  playerMove()
  printBoard()
  if ...:
    print('Вы выиграли!')
    break
  computerMove()
  printBoard()
  if ...:
    print('Я победил!')
    break
```

Как определять окончание игры? Если один из королей пропал, то есть был «съеден» после очередного хода, значит, игра окончена. Как сделать такую проверку, разберёмся чуть позже.

### 10.2.3. Вывод игрового поля

Нет ничего проще, чем вывести готовый набор строк. Очень удобно, что в них сразу хранятся нужные символы. Единственное, что нужно добавить, это координатные оси, помогающие при вводе хода.

**C++**

```
void printBoard()
{
  std::cout << "  abcde\n";
  for (int i = N - 1 ; i >= 0 ; --i)
  {
    std::cout << i + 1 << " " << board[i]
    ↪  << "\n";
  }
  std::cout << "\n";
}
```

**Python**

```
def printBoard():
  print('  abcde')
  for i in range(N - 1, -1, -1):
    print(f'{i + 1}
    ↪  {"".join(board[i])}')
  print()
```

### 10.2.4. Ход игрока

Как обычно, пользователь должен ввести координаты. Сначала координаты фигуры, потом координаты клетки, куда эта фигура пойдёт.

Чтобы из введённых пользователем символов (букв и цифр) получить координаты от 0 до $N-1$, нужно вычесть из кодов введённых символов коды символов, с которых начинается нумерация.

C++

```
void playerMove()
{
  char c1, r1, c2, r2;
  do
  {
    std::cout << "Введите ваш ход: ";
    std::cin >> c1 >> r1 >> c2 >> r2;
    c1 -= 'a';
    r1 -= '1';
    c2 -= 'a';
    r2 -= '1';
  }
  while (!isValidMove(board, 0, c1, r1,
  ↪  c2, r2));
  makeMove(board, c1, r1, c2, r2);
}
```

Python

```
def playerMove():
  c1 = 0
  r1 = 0
  c2 = 0
  r2 = 0
  while not isValidMove(board, 0, c1, r1,
  ↪  c2, r2):
    line = input('Введите ваш ход: ')
    c1, r1, c2, r2 = list(line)
    c1 = ord(c1) - ord('a')
    r1 = ord(r1) - ord('1')
    c2 = ord(c2) - ord('a')
    r2 = ord(r2) - ord('1')
  return makeMove(board, c1, r1, c2, r2)
```

### 10.2.5. Перемещение фигуры

Попробуем сделать программу рабочей. Для этого добавим функцию проверки корректности хода, которая будет одобрять все ходы, а также простейшую функцию, перемещающую фигуру на новое место.

Что такое перемещение фигуры? Нужно скопировать символ из клетки, где стоит фигура, на новое место, а старое заменить на точку. Что делать, если клетка-назначение занята? А ничего, это и будет взятие фигуры. Пока любой, а позже мы будем проверять, чтобы игрок не смог атаковать собственные фигуры.

C++

```
bool isValidMove(const Board &board, int
↪  c1, int r1, int c2, int r2)
{
  return true;
}

void makeMove(Board &board, int c1, int
↪  r1, int c2, int r2)
{
  char f = board[r1][c1];
  board[r1][c1] = '.';
  board[r2][c2] = f;
}
```

Python

```
def isValidMove(board, player, c1, r1,
↪  c2, r2):
  return True

def makeMove(board, c1, r1, c2, r2):
  f = board[r1][c1]
  board[r1][c1] = '.'
  board[r2][c2] = f
```

Уже можно «играть». Командовать любыми фигурами. И даже точками, ведь проверок-то пока нет. Главное, не вводите некорректных координат, а то можно нарушить работу программы, ведь запись за границы массива — это неопределённое поведение в C++ и исключительная ситуация в Python.

### 10.2.6. Взятие фигуры соперника

Если достаточно долго играть с уже получившейся программой, можно потерять короля, но программа при этом не остановится. Получается, что в тот момент, когда король пропадает с доски, функция `playerMove` (да и `computerMove` тоже) должна возвращать какое-то значение. А откуда она его возьмёт? Конечно, проще всего получить его из функции `makeMove`, а не сканировать всю доску.

И раз уж мы расширяем возможности `makeMove`, есть смысл что-то возвращать и при взятии других фигур. Тогда эти значения можно будет использовать при оценке ходов компьютера.

**C++**

```
#include <cctype>

const int KING = 1000;

int makeMove(Board &board, int c1, int
↪ r1, int c2, int r2)
{
  char f = board[r1][c1];
  char t = std::toupper(board[r2][c2]);
  board[r1][c1] = '.';
  board[r2][c2] = f;
  switch (t)
  {
  case 'P':
    return 1;
  case 'R':
    return 5;
  case 'N':
    return 3;
  case 'B':
    return 3;
  case 'Q':
    return 9;
  case 'K':
    return KING;
  }
  return 0;
}
```

**Python**

```
KING = 1000

def makeMove(board, c1, r1, c2, r2):
  f = board[r1][c1]
  t = board[r2][c2].upper()
  board[r1][c1] = '.'
  board[r2][c2] = f
  if t == 'P':
    return 1
  if t == 'R':
    return 5
  if t == 'N':
    return 3
  if t == 'B':
    return 3
  if t == 'Q':
    return 9
  if t == 'K':
    return KING
  return 0
```

Теперь функция возвращает «стоимость» фигуры, если она была взята в результате хода. Стоимости это более-менее классические значения, принятые для обычных шахмат. Но вы позже можете подобрать и другие, чтобы потом «настроить» игру компьютера.

Король, в отличие от других фигур, имеет именованную константу, потому что нужно также использовать её в главном игровом цикле, чтобы узнать, что игра завершилась:

C++

```
int m = playerMove();
printBoard();
if (m == KING)
{
  std::cout << "Вы выиграли!\n";
  break;
}
```

Python

```
m = playerMove()
printBoard()
if m == KING:
  print('Вы выиграли!')
  break
```

Играть становится всё интереснее. Теперь при взятии короля программа должна завершиться.

### 10.2.7. Проверка корректности хода

Пользователь вводит координаты с клавиатуры, а значит, может ошибиться. Поэтому в первую очередь нужно проверить, корректны ли координаты предполагаемой фигуры.

Кроме того, описание хода должно ссылаться на разные клетки старта и финиша движения фигуры. Ну и, конечно, стартовая клетка не должна быть пустой. Это всё нужно проверять в самом начале функции, перед тем как будем смотреть на корректность перемещения фигур:

C++

```
bool isValidMove(const Board &board, int
↪ player, int c1, int r1, int c2, int
↪ r2)
{
  if (c1 < 0 || c1 >= N || r1 < 0 || r1
  ↪ >= N
    || c2 < 0 || c2 >= N || r2 < 0 || r2
    ↪ >= N)
  {
    return false;
  }
  if (c1 == c2 && r1 == r2)
  {
    return false;
  }
  char f = board[r1][c1];
  if (f == '.')
  {
    return false;
  }
  ...
  return false;
}
```

Python

```
def isValidMove(board, player, c1, r1,
↪ c2, r2):
  if c1 < 0 or c1 >= N or r1 < 0 or r1 >=
  ↪ N or c2 < 0 or c2 >= N or r2 < 0 or
  ↪ r2 >= N:
    return False
  if c1 == c2 and r1 == r2:
    return False
  f = board[r1][c1]
  if f == '.':
    return False
  ...
  return False
```

### 10.2.8. Перемещение фигур

Дальше стоит проверить, что в исходной клетке стоит фигура текущего игрока (не зря же у функции параметр `player`), а в клетке-назначении другой его фигуры нет (но зато там может быть фигура оппонента или не быть ничего).

Оба условия можно реализовать с помощью функции проверки принадлежности фигуры игроку, `isOwnFigure`.

C++

```
char t = board[r2][c2];
if (!isOwnFigure(player, f)
  || isOwnFigure(player, t))
{
  return false;
}
```

Python

```
t = board[r2][c2]
if not isOwnFigure(player, f)
    or isOwnFigure(player, t):
  return False
```

Функция `isOwnFigure` должна убедиться, что для первого игрока (`player==0`) передана большая буква, а для второго — маленькая. Конкретные буквы проверять не нужно, потому что проще проверить сразу диапазон.

C++

```
bool isOwnFigure(int player, char f)
{
  if (player == 0 && std::isupper(f))
  {
    return true;
  }
  if (player == 1 && std::islower(f))
  {
    return true;
  }
  return false;
}
```

Python

```
def isOwnFigure(player, f):
  if player == 0 and f.isupper():
    return True
  if player == 1 and f.islower():
    return True
  return False
```

### 10.2.9. Ход пешки

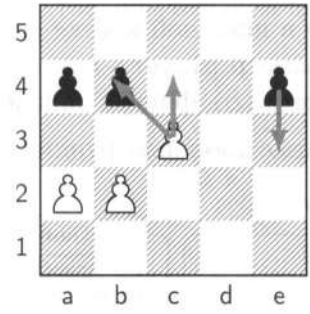
Пока что `isValidMove` всегда возвращает «ложь», поэтому сделать какой-нибудь ход не выйдет. Давайте реализуем хоть что-нибудь, чтобы начать тестирование.

Самая простая фигура с точки зрения правил — это король. Но в начале игры он заперт, поэтому займёмся пешками.

Пешка ходит только вперёд, но это понятие зависит от точки зрения. Пешки первого игрока увеличивают координату по вертикали, а второго — уменьшают.

Поэтому сначала мы проверяем, в правильном ли направлении идёт пешка в зависимости от её буквы (заглавной или строчной).

Вторая проверка срабатывает, если пешка идёт прямо. Тогда она может занять только пустую клетку.




И третья проверка нужна для хода по диагонали. Ход по диагонали возможен только в случае взятия фигуры противника. Это значит, что можно снова использовать функцию `isOwnFigure`, передав туда номер противоположного игрока.

C++

```
if (f == 'P' || f == 'p')
{
  int dy = f == 'P' ? 1 : -1;
  if (r1 + dy != r2)
  {
    return false;
  }
  if (c1 == c2)
  {
    return t == '.';
  }
  if (c1 + 1 == c2 || c1 - 1 == c2)
  {
    return isOwnFigure(1 - player, t);
  }
  return false;
}
```

Python

```
if f == 'P' or f == 'p':
  dy = 1 if f == 'P' else -1
  if r1 + dy != r2:
    return False
  if c1 == c2:
    return t == '.'
  if c1 + 1 == c2 or c1 - 1 == c2:
    return isOwnFigure(1 - player, t)
  return False
```

Можно начинать играть пешками. Сколько фигур противника вам удалось «съесть»?

### 10.2.10. Ход короля

Ходы всех остальных фигур не зависят от их цвета. И чёрные, и белые ходят в любых направлениях. Поэтому можно привести букву-обозначение текущей фигуры к верхнему регистру, чтобы упростить её сравнение.

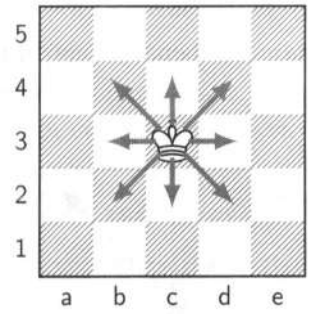
C++

```
f = std::toupper(f);
int dx = std::abs(c2 - c1);
int dy = std::abs(r2 - r1);
```

Python

```
f = f.upper()
dx = abs(c2 - c1)
dy = abs(r2 - r1)
```

Итак, ход короля.




Король может перемещаться на одну клетку в любом направлении. Выше мы уже рассчитали приращение координат фигуры в соответствии со строками и столбцами начальной и конечной клеток хода. И тогда ход будет корректным, если каждое из приращений по модулю не превышает единицы (а стояние фигуры на месте мы уже отфильтровали раньше).

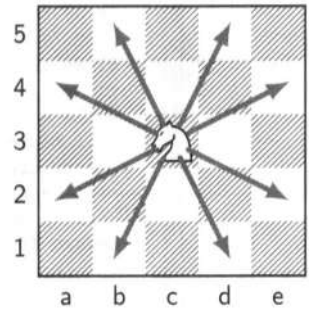
C++

```
if (f == 'K')
{
  return dx <= 1 && dy <= 1;
}
```

Python

```
if f == 'K':
  return dx <= 1 and dy <= 1
```

### 10.2.11. Ход коня




Проверка корректности хода коня тоже несложная. Так как он перепрыгивает фигуры на своём пути, нужно проверить только, что ход похож на букву «Г»: модуль приращения одной координаты равняется единице, а второй — двойке.

C++

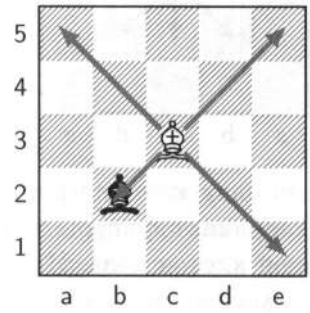
```
if (f == 'N')
{
  return (dx == 2 && dy == 1)
      || (dx == 1 && dy == 2);
}
```

Python

```
if f == 'N':
  return (dx == 2 and dy == 1)
      or (dx == 1 and dy == 2)
```

### 10.2.12. Ход слона

Все остальные фигуры (слон, ладья и ферзь) ходят по прямой, причём на несколько клеток сразу.




Поэтому для них уже недостаточно проверки только конечной клетки и направления движения. Нужно ещё убедиться, что на пути ничего нет.

Создадим для этого функцию `isFreeWay`. Она пройдёт по всем клеткам, от начальной до конечной (но не включая их), и проверит, что каждая из них пустая.

C++

```
bool isFreeWay(const Board &board, int
↪  c1, int r1, int c2, int r2)
{
  int dx = c2 - c1;
  if (dx != 0)
  {
    dx /= std::abs(dx);
  }
  int dy = r2 - r1;
  if (dy != 0)
  {
    dy /= std::abs(dy);
  }
  c1 += dx;
  r1 += dy;
  while (c1 != c2 || r1 != r2)
  {
    if (board[r1][c1] != '.')
    {
      return false;
    }
    c1 += dx;
    r1 += dy;
  }
  return true;
}
```

Python

```
def isFreeWay(board, c1, r1, c2, r2):
  dx = c2 - c1
  if dx != 0:
    dx //= abs(dx)
  dy = r2 - r1
  if dy != 0:
    dy //= abs(dy)
  c1 += dx
  r1 += dy
  while c1 != c2 or r1 != r2:
    if board[r1][c1] != '.':
      return False
    c1 += dx
    r1 += dy
  return True
```

У этой функции есть особенность — она никогда не завершится, если движение не идёт по горизонтали, вертикали или диагонали. Поэтому при проверке

возможности хода для каждой из оставшихся фигур нужно сначала убедиться, что начальная и конечная клетка соответствуют правилам движения фигуры, а потом уже проверять это движение.

Для слона, который двигается по диагонали, модули изменения его положения по горизонтали и вертикали должны совпадать (это как раз и будет прямая, наклонённая под 45°).

А дальше уже можно и функцию `isFreeWay` вызвать.

C++

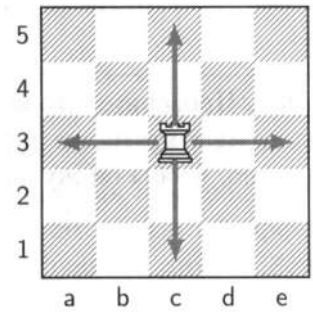
```
if (f == 'B')
{
  if (dx != dy)
  {
    return false;
  }
  return isFreeWay(board, c1, r1, c2,
  ↪  r2);
}
```

Python

```
if f == 'B':
  if dx != dy:
    return False
  return isFreeWay(board, c1, r1, c2, r2)
```

## 10.2.13. Ход ладьи

Чем отличается ладья от слона? Она тоже ходит по прямой, только прямая должна быть либо горизонтальной, либо вертикальной.




Это то же самое, что и утверждение «одна из координат не меняется». А если меняются обе, то ход некорректный.

C++

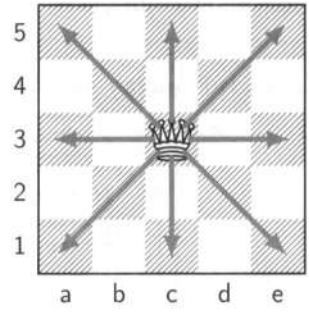
```
if (f == 'R')
{
  if (dx && dy)
  {
    return false;
  }
  return isFreeWay(board, c1, r1, c2,
  ↪  r2);
}
```

Python

```
if f == 'R':
  if dx and dy:
    return False
  return isFreeWay(board, c1, r1, c2, r2)
```

### 10.2.14. Ход ферзя

Ферзь может ходить и как слон, и как ладья.




Поэтому и проверка координат будет сложнее. Тут исключаются случаи, когда меняются обе координаты, но ход при этом не идёт по диагонали.

C++

```
if (f == 'Q')
{
  if (dx != dy && dx && dy)
  {
    return false;
  }
  return isFreeWay(board, c1, r1, c2,
  ↪  r2);
}
```

Python

```
if f == 'Q':
  if dx != dy and dx and dy:
    return False
  return isFreeWay(board, c1, r1, c2, r2)
```

Поиграйте уже по правилам. Проверьте, что все фигуры могут ходить как надо и не могут как не надо. Ведь этими же правилами будет пользоваться компьютер. Нам же не нужно, чтобы он жульничал?

### 10.2.15. Случайный ход компьютера

Казалось бы, сделать случайные ходы сложнее, чем просчитанные. Нужно выбрать фигуру (случайно), потом найти клетки, куда она может пойти. Потом снова случайно выбрать одну из этих возможностей.

Но оказывается, что всё намного проще. Ведь мы можем проверить любой ход на корректность (человек-то вводит как раз случайные ходы).

Следовательно, достаточно лишь сгенерировать четыре координаты — для начальной и конечной клеток, и проверить, возможен ли такой ход.

C++

```
int computerMove()
{
  int c1, r1, c2, r2;
  do
  {
    c1 = rand() % N;
    r1 = rand() % N;
    c2 = rand() % N;
    r2 = rand() % N;
  }
  while (!isValidMove(board, 1, c1, r1,
  ↪  c2, r2));

  std::cout << "Мой ход: "
    << (char)(c1 + 'a')
    << (char)(r1 + '1')
    << '-' << (char)(c2 + 'a')
    << (char)(r2 + '1')
    << "\n\n";

  return makeMove(board, c1, r1, c2, r2);
}
```

Python

```
import random

def computerMove():
  c1 = 0
  r1 = 0
  c2 = 0
  r2 = 0
  while not isValidMove(board, 1, c1, r1,
  ↪  c2, r2):
    c1 = random.randint(0, N - 1)
    r1 = random.randint(0, N - 1)
    c2 = random.randint(0, N - 1)
    r2 = random.randint(0, N - 1)

  print(f'Мой ход: {chr(c1 +
  ↪  ord('a'))}{r1 + 1}-{chr(c2 +
  ↪  ord('a'))}{r2 + 1}\n')

  return makeMove(board, c1, r1, c2, r2)
```

Ну вот и компьютер тоже научился делать ходы. Самое время насладиться своими победами над ним (вдруг дальше делать это не получится).

### 10.2.16. Перебор первого хода

Перебор ходов можно организовать так же, как это делалось в игре Калах функция `computerMove` ищет первый ход, имеющий максимальную оценку по мнению рекурсивной функции, проверяющей всё дерево игры.

Сам ход описывается координатами начальной и конечной клетки, все комбинации которых перебираются. Это не самый эффективный способ, но довольно удобный, так как у нас есть функция `isValidMove`, благодаря которой можно пропускать некорректные варианты.

Вообще для поиска хода было бы удобнее просто запустить рекурсию, а она бы сразу и нужный ход вернула. Но тут придётся возвращать сразу 4 значения (координаты клеток), рекурсия станет более запутанной. Мы сделаем как в прошлый раз, но взамен усложнится вызывающая перебор функция, так как ход придётся выбирать прямо в ней.

Ещё в прошлый раз мы вызывали `makeMove` на входе в рекурсивный перебор, а сейчас делаем это перед вызовом. Разница невелика, но зато не нужно передавать все координаты в виде параметров. Однако результат хода (стоимость «съеденной» фигуры) мы всё же передаём в рекурсивную функцию, чтобы получить полную оценку позиции, включая лучший вариант развития событий.

**C++**

```
const int MAX = KING * 2;

int checkMove(const Board &board, int
↪  player, int depth, int score)
{
  return score;
}

int computerMove()
{
  int mc1, mr1, mc2, mr2;
  int max = -MAX;
  for (int c1 = 0 ; c1 < N ; ++c1)
  {
    for (int r1 = 0 ; r1 < N ; ++r1)
    {
      for (int c2 = 0 ; c2 < N ; ++c2)
      {
        for (int r2 = 0 ; r2 < N ; ++r2)
        {
          if (isValidMove(board, 1, c1,
          ↪  r1, c2, r2))
          {
            Board b(board);
            int m = makeMove(b, c1, r1,
            ↪  c2, r2);
            m = checkMove(b, 0, 5, m);
            if (m > max)
            {
              max = m;
              mc1 = c1;
              mr1 = r1;
              mc2 = c2;
              mr2 = r2;
            }
          }
        }
      }
    }
  }

  std::cout << "Мой ход: "
    << (char)(mc1 + 'a')
    << mr1 + 1
    << '-' << (char)(mc2 + 'a')
    << mr2 + 1 << "\n\n";

  return makeMove(board, mc1, mr1, mc2,
  ↪  mr2);
}
```

**Python**

```
import copy

MAX = KING * 2

def checkMove(board, player, depth,
↪  score):
  return score

def computerMove():
  max = -MAX
  for c1 in range(N):
    for r1 in range(N):
      for c2 in range(N):
        for r2 in range(N):
          if isValidMove(board, 1, c1,
          ↪  r1, c2, r2):
            b = copy.deepcopy(board)
            m = makeMove(b, c1, r1, c2,
            ↪  r2)
            m = checkMove(b, 0, 3, m)
            if m > max:
              max = m
              mc1 = c1
              mr1 = r1
              mc2 = c2
              mr2 = r2

  print(f'Мой ход: {chr(mc1 +
  ↪  ord('a'))}{mr1 + 1}-{chr(mc2 +
  ↪  ord('a'))}{mr2 + 1}\n')

  return makeMove(board, mc1, mr1, mc2,
  ↪  mr2)
```

Вместо того чтобы менять игровое поле `board`, мы создаём его копию `b`, чтобы после проверки очередного хода вернуться к начальной позиции.

Тут, как и в варианте для C++, нужно создавать копию игрового поля. Но в Python встроенного оператора для этого нет, поэтому используется функция из модуля `copy`. Важно, чтобы это была функция `deepcopy`, потому что надо продублировать не только сам массив из строк, но и те строки, что хранятся внутри. Иначе все массивы-копии будут ссылаться на одни и те же экземпляры строк, и исходная позиция будет меняться при перемещении фигур в скопированных полях.



Новая программа тоже работоспособна. Только теперь компьютер будет стараться взять самую дорогую фигуру (ведь мы перебираем только один ход).

### 10.2.17. Рекурсивный перебор ходов

А чем рекурсивный перебор ходов отличается от функции, которая выбирает один ход? Во-первых, здесь запоминать сам ход не требуется, ведь нам нужна только его оценка. А во-вторых, нужно учитывать, что игроком, для которого оценка считается, будет не только компьютер, но и человек.

Завершается рекурсия, когда достигнута заданная глубина перебора либо последний ход был взятием короля. Тогда функция возвращает накопленную оценку пройденной цепочки ходов (стоимость всех фигур, которые были «съедены»).

Стоимость последнего хода мы тут проверить не можем, но если текущий счёт достаточно велик по модулю, значит, какого-то из королей на доске больше нет.

C++

```
const int END = KING / 2;

int checkMove(const Board &board, int
↪  player, int depth, int score)
{
  if (depth == 0
    || std::abs(score) >= END)
  {
    return score;
  }
  ...
}
```

Python

```
END = KING // 2

def checkMove(board, player, depth,
↪  score):
  if depth == 0 or abs(score) >= END:
    return score
```

Оставшийся фрагмент похож на тот, что был в функции `computerMove`. Перебираются все пары клеток, для подходящего варианта делается ход, а потом вызывается функция `checkMove`.

Разница здесь в том, что не запоминаются параметры найденного хода, а только максимизируется или минимизируется оценка. Для компьютера (`player == 1`) она будет максимизироваться, а для игрока (`player == 0`) — минимизироваться.

Ещё из-за этого придётся инвертировать результат функции `makeMove`, когда ходит игрок. Ведь «съеденные» им фигуры должны уменьшать оценку позиции (так как она ухудшается для компьютера, а он стремится максимизировать оценку).

C++

```
int res = player == 0 ? MAX : -MAX;
for (int c1 = 0 ; c1 < N ; ++c1)
{
  for (int r1 = 0 ; r1 < N ; ++r1)
  {
    for (int c2 = 0 ; c2 < N ; ++c2)
    {
      for (int r2 = 0 ; r2 < N ; ++r2)
      {
        if (isValidMove(board, player,
        ↪  c1, r1, c2, r2))
        {
          Board b(board);
          int r = makeMove(b, c1, r1, c2,
          ↪  r2);
          if (player == 0)
          {
            r = -r;
          }
          r = checkMove(b, 1 - player,
          ↪  depth - 1, score + r);
          if (player == 0)
          {
            res = std::min(r, res);
          }
          else
          {
            res = std::max(r, res);
          }
        }
      }
    }
  }
}
return res;
```

Python

```
res = MAX if player == 0 else -MAX
for c1 in range(N):
  for r1 in range(N):
    for c2 in range(N):
      for r2 in range(N):
        if isValidMove(board, player, c1,
        ↪  r1, c2, r2):
          b = copy.deepcopy(board)
          r = makeMove(b, c1, r1, c2, r2)
          if player == 0:
            r = -r
          r = checkMove(b, 1 - player,
          ↪  depth - 1, score + r)
          if player == 0:
            res = min(r, res)
          else:
            res = max(r, res)
return res
```

### 10.2.18. Альфа-бета отсечение

Перебор с использованием альфа-бета отсечения совсем не отличается от аналогичного в игре Калах. Всего-то надо добавить два параметра рекурсивной функции.

Один параметр (альфа) используется для отбрасывание позиций, которые заведомо не будут выбраны компьютером, потому что для текущего поддерева игры уже есть слишком плохая позиция, и всё это поддерево не будет учитываться.

Например, после одного варианта хода компьютера он выигрывает слона, а в самом начале проверки другого хода (когда рассматривается ход человека) становится известно, что компьютер теряет коня, и (очевидно) ход, ведущий к такому развитию событий, рассматривать не нужно, потому что, как минимум, одна ветка там выгодна противнику (и её-то он наверняка и выберет).

Второй параметр (бета) нужен для противоположной ситуации. Когда алгоритм оценивает позицию с точки зрения игрока, поддеревья можно отбрасывать, если они получаются заведомо для него невыгодными, потому что у компьютера там будет слишком хороший ход.

**C++**

```
int checkMove(const Board &board, int
↪  player, int depth, int score, int
↪  alpha, int beta)
{
  ...
  r = checkMove(b, 1 - player, depth - 1,
  ↪  score + r, alpha, beta);
  if (player == 0)
  {
    res = std::min(r, res);
    if (r < alpha)
    {
      return res;
    }
    beta = std::min(beta, r);
  }
  if (player == 1)
  {
    res = std::max(r, res);
    if (r > beta)
    {
      return res;
    }
    alpha = std::max(alpha, r);
  }
  ...
}

int computerMove()
{
  ...
  m = checkMove(b, 0, 7, m, -MAX, MAX);
  ...
}
```

**Python**

```
def checkMove(board, player, depth,
↪  score, alpha, beta):
  ...
  r = makeMove(b, c1, r1, c2, r2)
  if player == 0:
    r = -r
  r = checkMove(b, 1 - player, depth - 1,
  ↪  score + r, alpha, beta)
  if player == 0:
    res = min(r, res)
    if r < alpha:
      return res
    beta = min(beta, r)
  else:
    res = max(r, res)
    if r > beta:
      return res
    alpha = max(alpha, r)
  ...

def computerMove():
  ...
  m = checkMove(b, 0, 5, m, -MAX, MAX)
  ...
```

Вот и всё. Теперь можно увеличивать глубину перебора по сравнению с обычным рекурсивным алгоритмом оценки позиций. Стал ли компьютер играть лучше?

### 10.2.19. Задания для самостоятельной работы

**1.** Попробуйте ускорить перебор с помощью отсечения заведомо бесполезных вариантов в циклах, которые выбирают первую клетку для хода. Например, если на исходной клетке стоит фигура противника (или она пустая), перебирать финальную клетку хода не нужно. Ещё множество клеток для перебора ходов фигуры можно выбирать, исходя из её вида вместо сканирования всего поля.

**2.** Иногда компьютер почти добивается цели, оставаясь с одной-двумя пешками и слоном. Реализуйте превращение пешек в более сильные фигуры, чтобы компьютер мог довести партию до конца и победить.

**3.** При оценке ходов не учитывается скорость достижения цели. Компьютер может выиграть за один ход, а может за несколько, но выберет первый попавшийся из этих вариантов. Доработайте оценку позиции, чтобы минимизировать число ходов для победы.

**4.** Сделайте так, чтобы программа определяла шах и мат и не нужно было «съедать» короля для завершения игры. Вероятно, при этом понадобится ещё и ввести понятие «пат», когда противник не может никуда сходить. К примеру, если король остался в одиночестве, а все клетки вокруг него под боем фигур противника.

**Листинг 10.3. minichess.cpp**

```
#include <iostream>
#include <string>
#include <cctype>
#include <array>

const int N = 5;
const int KING = 1000;
const int END = KING / 2;
const int MAX = KING * 2;

typedef std::array<std::string, N> Board;

Board board = {
    "RNBQK",
    "PPPPP",
    ".....",
    "ppppp",
    "rnbqk"
```

```
};

void printBoard()
{
    std::cout << "  abcde\n";
    for (int i = N - 1 ; i >= 0 ; --i)
    {
        std::cout << i + 1 << " " << board[i] << "\n";
    }
    std::cout << "\n";
}

bool isOwnFigure(int player, char f)
{
    if (player == 0 && std::isupper(f))
    {
        return true;
    }
    if (player == 1 && std::islower(f))
    {
        return true;
    }
    return false;
}

bool isFreeWay(const Board &board, int c1, int r1, int c2, int r2)
{
    int dx = c2 - c1;
    if (dx != 0)
    {
        dx /= std::abs(dx);
    }
    int dy = r2 - r1;
    if (dy != 0)
    {
        dy /= std::abs(dy);
    }
    c1 += dx;
    r1 += dy;
    while (c1 != c2 || r1 != r2)
    {
```

```
        if (board[r1][c1] != '.')
        {
            return false;
        }
        c1 += dx;
        r1 += dy;
    }
    return true;
}

bool isValidMove(const Board &board, int player, int c1, int r1, int
↪ c2, int r2)
{
    if (c1 < 0 || c1 >= N || r1 < 0 || r1 >= N
        || c2 < 0 || c2 >= N || r2 < 0 || r2 >= N)
    {
        return false;
    }
    if (c1 == c2 && r1 == r2)
    {
        return false;
    }
    char f = board[r1][c1];
    if (f == '.')
    {
        return false;
    }
    char t = board[r2][c2];
    if (!isOwnFigure(player, f) || isOwnFigure(player, t))
    {
        return false;
    }
    if (f == 'P' || f == 'p')
    {
        int dy = f == 'P' ? 1 : -1;
        if (r1 + dy != r2)
        {
            return false;
        }
        if (c1 == c2)
        {
```

```
            return t == '.';
        }
        if (c1 + 1 == c2 || c1 - 1 == c2)
        {
            return isOwnFigure(1 - player, t);
        }
        return false;
    }
    f = std::toupper(f);
    int dx = std::abs(c2 - c1);
    int dy = std::abs(r2 - r1);
    if (f == 'K')
    {
        return dx <= 1 && dy <= 1;
    }
    if (f == 'N')
    {
        return (dx == 2 && dy == 1) || (dx == 1 && dy == 2);
    }
    if (f == 'R')
    {
        if (dx && dy)
        {
            return false;
        }
        return isFreeWay(board, c1, r1, c2, r2);
    }
    if (f == 'B')
    {
        if (dx != dy)
        {
            return false;
        }
        return isFreeWay(board, c1, r1, c2, r2);
    }
    if (f == 'Q')
    {
        if (dx != dy && dx && dy)
        {
            return false;
        }
```

```
        return isFreeWay(board, c1, r1, c2, r2);
    }
    return false;
}

int makeMove(Board &board, int c1, int r1, int c2, int r2)
{
    char f = board[r1][c1];
    char t = std::toupper(board[r2][c2]);
    board[r1][c1] = '.';
    board[r2][c2] = f;
    switch (t)
    {
    case 'P':
        return 1;
    case 'R':
        return 5;
    case 'N':
        return 3;
    case 'B':
        return 3;
    case 'Q':
        return 9;
    case 'K':
        return KING;
    }
    return 0;
}

int playerMove()
{
    char c1, r1, c2, r2;
    do
    {
        std::cout << "Введите ваш ход: ";
        std::cin >> c1 >> r1 >> c2 >> r2;
        c1 -= 'a';
        r1 -= '1';
        c2 -= 'a';
        r2 -= '1';
    }
```

```
    while (!isValidMove(board, 0, c1, r1, c2, r2));
    return makeMove(board, c1, r1, c2, r2);
}

int checkMove(const Board &board, int player, int depth, int score,
↪   int alpha, int beta)
{
    if (depth == 0 || std::abs(score) >= END)
    {
        return score;
    }
    int res = player == 0 ? MAX : -MAX;
    for (int c1 = 0 ; c1 < N ; ++c1)
    {
        for (int r1 = 0 ; r1 < N ; ++r1)
        {
            for (int c2 = 0 ; c2 < N ; ++c2)
            {
                for (int r2 = 0 ; r2 < N ; ++r2)
                {
                    if (isValidMove(board, player, c1, r1, c2, r2))
                    {
                        Board b(board);
                        int r = makeMove(b, c1, r1, c2, r2);
                        if (player == 0)
                        {
                            r = -r;
                        }
                        r = checkMove(b, 1 - player, depth - 1, score
                        ↪   + r, alpha, beta);
                        if (player == 0)
                        {
                            res = std::min(r, res);
                            if (r < alpha)
                            {
                                return res;
                            }
                            beta = std::min(beta, r);
                        }
                        else
                        {
```

```
                        res = std::max(r, res);
                        if (r > beta)
                        {
                            return res;
                        }
                        alpha = std::max(alpha, r);
                    }
                }
            }
        }
    }
    return res;
}

int computerMove()
{
    int mc1, mr1, mc2, mr2;
    int max = -MAX;
    for (int c1 = 0 ; c1 < N ; ++c1)
    {
        for (int r1 = 0 ; r1 < N ; ++r1)
        {
            for (int c2 = 0 ; c2 < N ; ++c2)
            {
                for (int r2 = 0 ; r2 < N ; ++r2)
                {
                    if (isValidMove(board, 1, c1, r1, c2, r2))
                    {
                        Board b(board);
                        int m = makeMove(b, c1, r1, c2, r2);
                        m = checkMove(b, 0, 7, m, -MAX, MAX);
                        if (m > max)
                        {
                            max = m;
                            mc1 = c1;
                            mr1 = r1;
                            mc2 = c2;
                            mr2 = r2;
                        }
                    }
```

```
                }
            }
        }
    }
    std::cout << "Мой ход: " << (char)(mc1 + 'a') << mr1 + 1
        << '-' << (char)(mc2 + 'a') << mr2 + 1 << "\n\n";
    return makeMove(board, mc1, mr1, mc2, mr2);
}

int main()
{
    printBoard();
    while (true)
    {
        int m = playerMove();
        printBoard();
        if (m == KING)
        {
            std::cout << "Вы выиграли!\n";
            break;
        }
        m = computerMove();
        printBoard();
        if (m == KING)
        {
            std::cout << "Я победил!\n";
            break;
        }
    }
}
```

**Листинг 10.4. minichess.py**

```
#!/usr/bin/python3
import copy

N = 5
KING = 1000
END = KING // 2
MAX = KING * 2

board = [
    list('RNBQK'),
    list('PPPPP'),
    list('.....'),
    list('ppppp'),
    list('rnbqk')
]

def printBoard():
    print('  abcde')
    for i in range(N - 1, -1, -1):
        print(f'{i + 1} {"".join(board[i])}')
    print()

def isOwnFigure(player, f):
    if player == 0 and f.isupper():
        return True
    if player == 1 and f.islower():
        return True
    return False

def isFreeWay(board, c1, r1, c2, r2):
    dx = c2 - c1
    if dx != 0:
        dx //= abs(dx)
    dy = r2 - r1
    if dy != 0:
        dy //= abs(dy)
    c1 += dx
    r1 += dy
    while c1 != c2 or r1 != r2:
        if board[r1][c1] != '.':
```

```
            return False
        c1 += dx
        r1 += dy
    return True

def isValidMove(board, player, c1, r1, c2, r2):
    if c1 < 0 or c1 >= N or r1 < 0 or r1 >= N or c2 < 0 or c2 >= N or
    ↪ r2 < 0 or r2 >= N:
        return False
    if c1 == c2 and r1 == r2:
        return False
    f = board[r1][c1]
    if f == '.':
        return False
    t = board[r2][c2]
    if not isOwnFigure(player, f) or isOwnFigure(player, t):
        return False
    if f == 'P' or f == 'p':
        dy = 1 if f == 'P' else -1
        if r1 + dy != r2:
            return False
        if c1 == c2:
            return t == '.'
        if c1 + 1 == c2 or c1 - 1 == c2:
            return isOwnFigure(1 - player, t)
        return False
    f = f.upper()
    dx = abs(c2 - c1)
    dy = abs(r2 - r1)
    if f == 'K':
        return dx <= 1 and dy <= 1
    if f == 'N':
        return (dx == 2 and dy == 1) or (dx == 1 and dy == 2)
    if f == 'R':
        if dx and dy:
            return False
        return isFreeWay(board, c1, r1, c2, r2)
    if f == 'B':
        if dx != dy:
            return False
        return isFreeWay(board, c1, r1, c2, r2)
```

```
    if f == 'Q':
        if dx != dy and dx and dy:
            return False
        return isFreeWay(board, c1, r1, c2, r2)
    return False

def makeMove(board, c1, r1, c2, r2):
    f = board[r1][c1]
    t = board[r2][c2].upper()
    board[r1][c1] = '.'
    board[r2][c2] = f
    if t == 'P':
        return 1
    if t == 'R':
        return 5
    if t == 'N':
        return 3
    if t == 'B':
        return 3
    if t == 'Q':
        return 9
    if t == 'K':
        return KING
    return 0

def playerMove():
    c1 = 0
    r1 = 0
    c2 = 0
    r2 = 0
    while not isValidMove(board, 0, c1, r1, c2, r2):
        line = input('Введите ваш ход: ')
        c1, r1, c2, r2 = list(line)
        c1 = ord(c1) - ord('a')
        r1 = ord(r1) - ord('1')
        c2 = ord(c2) - ord('a')
        r2 = ord(r2) - ord('1')
    return makeMove(board, c1, r1, c2, r2)

def checkMove(board, player, depth, score, alpha, beta):
    if depth == 0 or abs(score) >= END:
```

```
            return score
        res = MAX if player == 0 else -MAX
        for c1 in range(N):
            for r1 in range(N):
                for c2 in range(N):
                    for r2 in range(N):
                        if isValidMove(board, player, c1, r1, c2, r2):
                            b = copy.deepcopy(board)
                            r = makeMove(b, c1, r1, c2, r2)
                            if player == 0:
                                r = -r
                            r = checkMove(b, 1 - player, depth - 1, score
                            ↪ + r, alpha, beta)
                            if player == 0:
                                res = min(r, res)
                                if r < alpha:
                                    return res
                                beta = min(beta, r)
                            else:
                                res = max(r, res)
                                if r > beta:
                                    return res
                                alpha = max(alpha, r)
        return res

def computerMove():
    max = -MAX
    for c1 in range(N):
        for r1 in range(N):
            for c2 in range(N):
                for r2 in range(N):
                    if isValidMove(board, 1, c1, r1, c2, r2):
                        b = copy.deepcopy(board)
                        m = makeMove(b, c1, r1, c2, r2)
                        m = checkMove(b, 0, 5, m, -MAX, MAX)
                        if m > max:
                            max = m
                            mc1 = c1
                            mr1 = r1
                            mc2 = c2
                            mr2 = r2
```

```
    print(f'Мой ход: {chr(mc1 + ord('a'))}{mr1 + 1}-{chr(mc2 +
    ↪ ord('a'))}{mr2 + 1}\n')
    return makeMove(board, mc1, mr1, mc2, mr2)

printBoard()
while True:
    m = playerMove()
    printBoard()
    if m == KING:
        print('Вы выиграли!')
        break
    m = computerMove()
    printBoard()
    if m == KING:
        print('Я победил!')
        break
```

# Глава 11

# Самообучающиеся игры

## 11.1. Игра «Угадай животное»

В этой игре компьютер пытается поразить игрока своим «интеллектом». Пользователь задумывает какое-нибудь животное, а компьютер задаёт вопросы о нём, на которые можно отвечать «да» или «нет».

Если в конце компьютер отгадывает, то остаётся лишь порадоваться за него. Если же ему отгадать не удалось, он пополняет свою базу знаний новым вопросом, а также запоминает то животное, которое было задумано. Подходящий вопрос, конечно же, придумывает пользователь (тяжёлое бремя победителя партии).

**Экран 1. Пример игры с программой «Угадай животное»**

```
Загадайте животное...
Оно умеет плавать? (д/н) н
Это птица? (д/н) н
Вы победили!
Что за животное вы загадали? слон
С помощью какого вопроса можно отличить птица от слон?
Это млекопитающее?
Хотите сыграть ещё? (д/н) д

Загадайте животное...
Оно умеет плавать? (д/н) н
Это млекопитающее? (д/н) д
Это слон? (д/н) н
Вы победили!
Что за животное вы загадали? кот
С помощью какого вопроса можно отличить слон от кот?
У него есть шерсть?
Хотите сыграть ещё? (д/н) д

Загадайте животное...
Оно умеет плавать? (д/н) н
Это млекопитающее? (д/н) д
У него есть шерсть? (д/н) д
Это кот? (д/н) д
Ура, я отгадал!
Хотите сыграть ещё? (д/н) н
```

### 11.1.1. Алгоритм игры

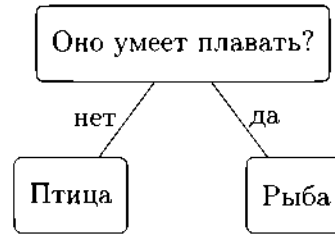
Для того чтобы начать придумывать программу, нужно решить, как именно она будет отгадывать.

Программа задаёт вопросы, на которые можно отвечать «да» или «нет», причём порядок вопросов строго определён. Следовательно, их можно расположить в виде дерева:

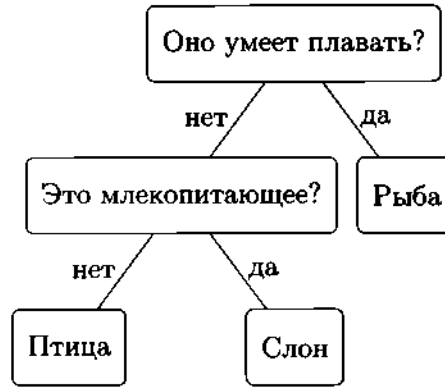

В промежуточных узлах дерева расположены вопросы, а в листьях — ответы. Игра начинается с корневого узла. Игрок отвечает на вопросы, и программа «идёт» по дереву вниз. В какую она пойдет сторону, зависит от ответа игрока. Если достигнут листовой узел, значит, вопросов больше нет и отгадка найдена.

Если компьютеру отгадать не удалось, дерево расширяется вниз. Вместо листа-ответа появляется узел с новым вопросом, потомками которого будут новый ответ (который был загадан) и старый (который был в дереве до этого).

Посмотрим, как будет меняться база знаний для приведённой в начале главы партии. Сначала в программе есть только один вопрос и два ответа:



После первого поражения программа дополняет базу вопросом «Это млекопитающее?» и ответом «Слон»:

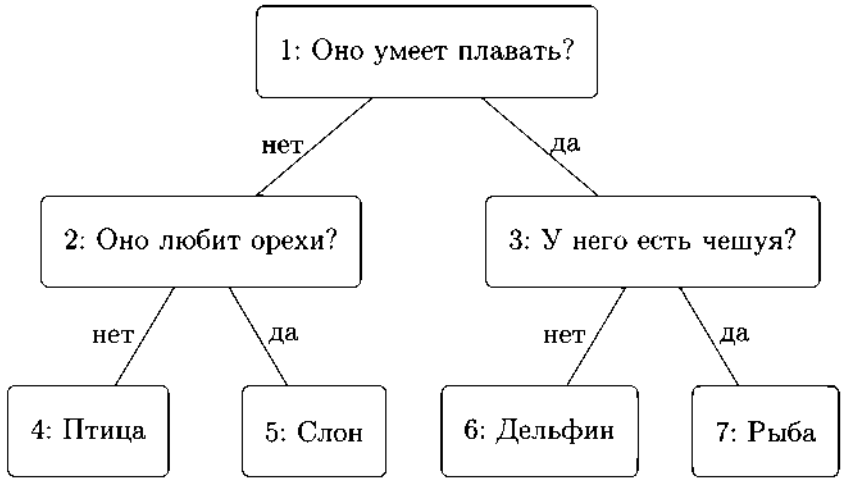



После второго раунда в базе появляются вопрос «У него есть шерсть?» и ответ «Кот»:


### 11.1.2. Игровая позиция

Игровая позиция состоит из базы знаний (вопросов и ответов), а также какой-то ссылки на текущий узел дерева, чтобы следующий ход игрока перемещал его вниз в левое или правое поддерево.

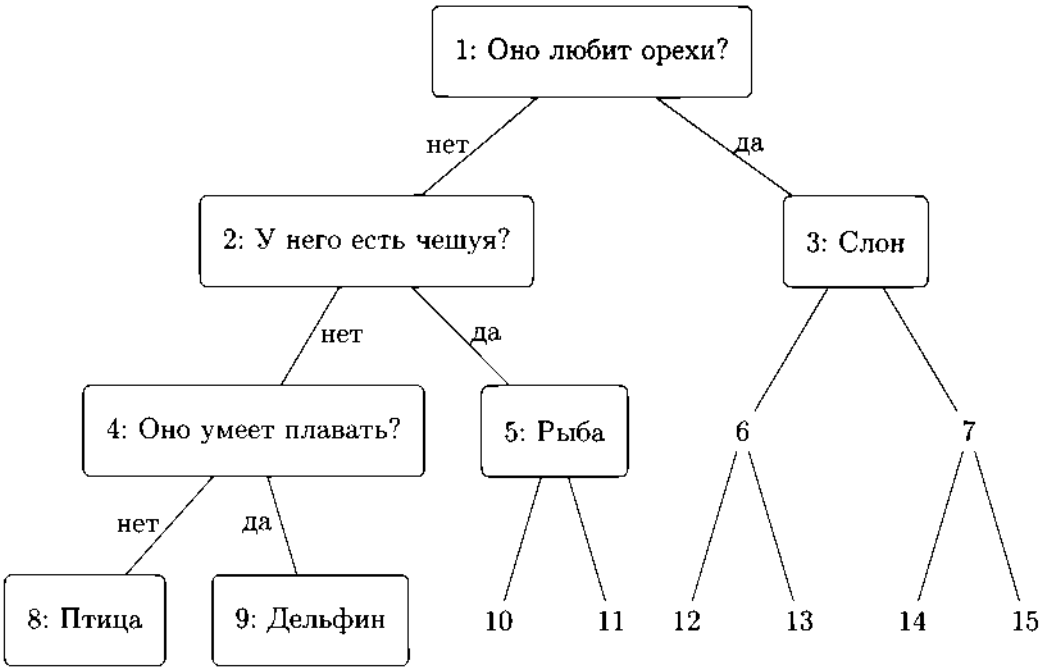
Если пронумеровать узлы дерева, приведённого выше, слева направо и сверху вниз, получится вот что:




Легко заметить, что номера дочерних узлов равняются $2p$ и $2p+1$, где $p$ — номер родительского узла.

Получается, что такая база знаний удобно разместится в массиве, все ячейки будут последовательно заниматься по мере роста дерева, а текущее положение в дереве будет определяться одним числом — номером узла (или же ячейки массива).

Но кто сказал, что дерево будет расти «правильно», то есть все уровни будут заполнены? Ведь после нескольких партий может получиться и такое:



Выходит, что если игрок не будет попадать в ветку со слоном, почти половина массива будет простаивать. Поэтому обычный массив это не лучший выбор.

Но вместо обычного массива можно использовать ассоциативный массив (словарь). Принцип формирования номера вопроса или ответа останется прежним, но обращаться по этому номеру будем уже к словарю. Тогда лишнее место не будет пропадать зря.

Конечно, ещё более правильно было бы строить дерево с помощью специально созданной структуры данных. Выделять память под его узлы, следить за ссылками на левого и правого потомков. Но в этом случае программы на C++ и Python слишком сильно бы отличались и возникла бы дополнительная путаница.

Поэтому воспользуемся словарём:

C++

```
#include <unordered_map>
#include <string>

std::unordered_map<long long,
↪ std::string> data = {{1, "Оно умеет
↪ плавать?"}, {2, "птица"}, {3,
↪ "рыба"}};
```

Здесь используется тип `long long`, а не `int`, чтобы можно было выстроить как можно более длинную цепочку вопросов. Ведь индексы узлов дерева растут очень быстро из-за удвоений.

Python

```
data = {1: 'Оно умеет плавать?', 2:
↪ 'птица', 3: 'рыба'}
```

В Python целочисленные переменные автоматически увеличиваются (с потерей эффективности, конечно) настолько, насколько необходимо.

В самом начале в базе знаний только один вопрос и две отгадки, для ответов «да» и «нет».

### 11.1.3. Главный игровой цикл

Как идёт игра? Цикл перемещается по дереву базы знаний и задаёт вопросы, пока не будет найдена отгадка.

Все вопросы и отгадки у нас пронумерованы, поэтому их легко получить, используя этот номер. Игра начинается с узла дерева $i = 1$, а на каждой итерации переходит или к узлу $2i$ (если ответ игрока «нет»), или к узлу $2i + 1$ (для ответа «да»).

Отличать вопрос от отгадки можно по наличию знака вопроса в конце. В итоге получится такая структура игрового цикла:

C++

```
#include <iostream>

int main()
{
  std::cout << "Загадайте
  ↪ животное...\n";
  long long i = 1;
  while (true)
  {
    std::string current = data[i];
    long long no = i * 2;
    long long yes = i * 2 + 1;
    if (current.back() == '?')
    {
      // нужно задать вопрос
    }
    else
    {
      // нужно вывести отгадку
    }
  }
}
```

Python

```
print('Загадайте животное...')
i = 1
while True:
  current = data[i]
  no = i * 2
  yes = i * 2 + 1
  if current[-1] == '?':
    # нужно задать вопрос
  else:
    # нужно вывести отгадку
```

### 11.1.4. Задаём игроку вопрос

Чтобы задать вопрос, нужно вывести на экран строку, которую мы прочитали в переменную `current`, а затем ввести с клавиатуры ответ на него.

За это будет отвечать функция `yesNo`. Почему сразу целая функция? Потому что спрашивать что-то с ответом «да» или «нет» понадобится ещё пару раз.

Для ответа «да» функция будет возвращать «истину». При этом игра должна перейти к левому поддереву текущего узла:

C++

```
...
if (current.back() == '?')
{
  if (yesNo(current))
  {
    i = yes;
  }
  else
  {
    i = no;
  }
}
...
```

Python

```
...
if current[-1] == '?':
  if yesNo(current):
    i = yes
  else:
    i = no
...
```

Сама эта функция тоже несложная. Сначала выводится вопрос, потом вводится строка с ответом игрока. Если он не ошибся с набором ответа (ввёл букву «д» или «н»), функция возвращает результат. В противном случае всё повторяется.

C++

```
bool yesNo(const std::string &question)
{
  while (true)
  {
    std::cout << question << " (д/н) ";
    std::string s;
    std::getline(std::cin, s);
    if (s == "д" || s == "Д")
    {
      return true;
    }
    if (s == "н" || s == "Н")
    {
      return false;
    }
  }
}
```

Python

```
def yesNo(question):
  while True:
    s = input(f'{question} (д/н) ')
    if s == 'д' or s == 'Д':
      return True
    if s == 'н' or s == 'Н':
      return False
```

### 11.1.5. Когда достигнут узел-отгадка

Когда вопросы кончились, игра достигла листа в дереве. В листьях у нас хранятся отгадки, поэтому нужно спросить у игрока, это ли животное было задумано. Если так оно и есть, игровой цикл просто завершается.

C++

```
...
if (yesNo("Это " + current + "?"))
{
  std::cout << "Ура, я отгадал!\n";
}
else
{
  // нужно добавить новый вопрос и новое
  ↪ животное
}
break;
...
```

Python

```
...
if yesNo(f'Это {current}?'):
  print('Ура, я отгадал!')
else:
  # нужно добавить новый вопрос и новое
  ↪ животное
break
...
```

### 11.1.6. Добавление нового вопроса

Если же животное отгадано неверно, придётся расширить базу знаний, чтобы программа научилась отгадывать задуманное в текущем раунде животное.

Сначала нужно спросить у игрока, кого именно он задумал, и записать название животного в переменную `animal`.

Получается, что последовательность вопросов, которая привела к текущей отгадке (`current`), подходит и для животного `animal`.

Поэтому дальше нужно получить от игрока вопрос, который программа в дальнейшем будет задавать, чтобы отличить животное в переменной `current` от животного в переменной `animal`.

И только после этого новый вопрос помещается в текущий узел дерева, `current` записывается в базу знаний в качестве его левого потомка, а `animal` — в качестве правого.

**C++**

```
std::cout << "Вы победили! \nЧто за
↪ животное вы загадали? ";
std::string animal;
std::getline(std::cin, animal);
std::cout << "С помощью какого вопроса
↪ можно отличить "
    << current << " от " << animal << "?
    ↪ ";
std::string question;
std::getline(std::cin, question);
data[i] = question;
data[no] = current;
data[yes] = animal;
```

**Python**

```
animal = input('Вы победили! \nЧто за
↪ животное вы загадали? ')
question = input(f'С помощью какого
↪ вопроса можно отличить {current} от
↪ {animal}? ')
data[i] = question
data[no] = current
data[yes] = animal
```

⌨ | Теперь попробуйте сыграть в игру. Всё работает? Или чего-то не хватает?

### 11.1.7. Цикл для последовательности игр

Сила этой игры в том, что программа постоянно улучшает свои знания, становясь всё более умным соперником.

Но сейчас, после окончания раунда, программа завершается, и база данных исчезает. Поэтому нужно продолжать работу программы, пока пользователь не наиграется. То есть следует проводить множество партий подряд.

Для этого заключим наш игровой цикл в другой цикл, который будет отвечать за последовательность игр. Завершаться этот цикл будет в случае, если пользователь не захочет больше играть (и тут снова пригодится функция `yesNo`).

**C++**

```
int main()
{
  do
  {
    std::cout << "\nЗагадайте
    ↪ животное...\n";
    long long i = 1;
    while (true)
    {
      ...
    }
  }
  while (yesNo("Хотите сыграть ещё?"));
}
```

**Python**

```
while True:
  print('\nЗагадайте животное...')
  i = 1
  while True:
    ...
  if not yesNo('Хотите сыграть ещё?'):
    break
```

Вот всё и готово. Можно играть много раз и наблюдать, как программа набирается ума.

### 11.1.8. Задания для самостоятельной работы

**1.** Немного неудобно, что для добавляемого в базу знаний животного ответ на введённый игроком вопрос всегда должен быть «да». И поэтому оно всегда помещается в правое поддерево. Иногда из-за этого сложно придумывать формулировки вопросов. Доработайте программу, чтобы дополнительно узнать у пользователя, какому ответу («да» или «нет») будет соответствовать животное.

**2.** Сохраняйте накопленную базу данных в файл при завершении программы, чтобы потом загружать её при старте. Так программу будет интереснее показывать друзьям, предварительно обучив её.

**3.** Можно придумать несколько выбираемых тем для игры, чтобы отгадывать не только животных. Например, географические названия, минералы, компьютерные игры, книги или фильмы.

**Листинг 11.1. animal.cpp**

```
#include <iostream>
#include <string>
#include <unordered_map>

std::unordered_map<long long, std::string> data = {{1, "Оно умеет
↪  плавать?"}, {2, "птица"}, {3, "рыба"}};

bool yesNo(const std::string &question)
{
    while (true)
    {
        std::cout << question << " (д/н) ";
        std::string s;
        std::getline(std::cin, s);
        if (s == "д" || s == "Д")
        {
            return true;
        }
        if (s == "н" || s == "Н")
        {
            return false;
```

```
        }
    }
}

int main()
{
    do
    {
        std::cout << "\nЗагадайте животное...\n";
        long long i = 1;
        while (true)
        {
            std::string current = data[i];
            long long no = i * 2;
            long long yes = i * 2 + 1;
            if (current.back() == '?')
            {
                if (yesNo(current))
                {
                    i = yes;
                }
                else
                {
                    i = no;
                }
            }
            else
            {
                if (yesNo("Это " + current + "?"))
                {
                    std::cout << "Ура, я отгадал!\n";
                }
                else
                {
                    std::cout << "Вы победили!\nЧто за животное вы
                    ↪ загадали? ";
                    std::string animal;
                    std::getline(std::cin, animal);
                    std::cout << "С помощью какого вопроса можно
                    ↪ отличить "
                        << current << " от " << animal << "? ";
```

```
                    std::string question;
                    std::getline(std::cin, question);
                    data[i] = question;
                    data[no] = current;
                    data[yes] = animal;
                }
                break;
            }
        }
    }
    while (yesNo("Хотите сыграть ещё?"));
}
```

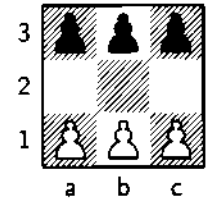
**Листинг 11.2. animal.py**

```
#!/usr/bin/python3

data = {1: 'Оно умеет плавать?', 2: 'птица', 3: 'рыба'}

def yesNo(question):
    while True:
        s = input(f'{question} (д/н) ')
        if s == 'д' or s == 'Д':
            return True
        if s == 'н' or s == 'Н':
            return False

while True:
    print('\nЗагадайте животное...')
    i = 1
    while True:
        current = data[i]
        no = i * 2
        yes = i * 2 + 1
        if current[-1] == '?':
            if yesNo(current):
                i = yes
            else:
                i = no
        else:
            if yesNo(f'Это {current}?'):
```

```
            print('Ура, я отгадал!')
        else:
            animal = input('Вы победили! \n Что за животное вы
            ↪ загадали? ')
            question = input(f'С помощью какого вопроса можно
            ↪ отличить {current} от {animal}? ')
            data[i] = question
            data[no] = current
            data[yes] = animal
        break
if not yesNo('Хотите сыграть ещё?'):
    break
```

## 11.2. Игра «6 пешек»

Игру «Шесть пешек» придумал Мартин Гарднер, чтобы показать, как своими руками сделать самообучающуюся машину из спичечных коробков [2].

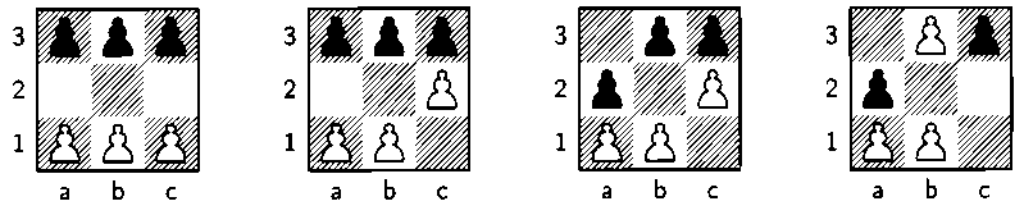



Пешки здесь родом из шахмат, но ходят по упрощённым правилам, без двойных ходов. Побеждает тот, кто проводит свою пешку на последнюю горизонталь либо блокирует возможность хода сопернику.

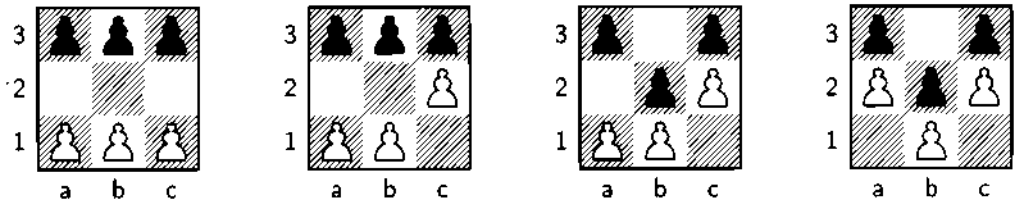
Сама по себе игра слишком простая с точки зрения стратегии. Не сложнее, чем крестики-нолики. Но зато с её помощью можно показать, как делаются простые самообучающиеся программы.

Обучение работает так. Программа просто делает первые попавшиеся ходы. Алгоритму доступны только две вещи — правила, по которым делаются ходы, а также список «запрещённых» позиций. Если программа проигрывает, последний свой ход (точнее, позицию, которая после него возникла) она записывает в «запрещённые». Тогда в следующий раз этот ход выбирать уже будет нельзя.

Например, после следующей партии запрещённым станет ход a3–a2 (потому что из получившейся позиции игрок делает выигрышный ход):




И если в будущих партиях позиция повторится, программа должна будет выбрать что-то другое:



То есть программа учится на своих поражениях, становясь со временем непобедимой.

**Экран 1. Начальная фаза обучения программы**

```
  abc
3 ppp
2 ...
1 PPP

Введите ваш ход: c1c2
  abc
3 ppp
2 ..P
1 PP.

Мой ход: a3-a2

  abc
3 .pp
2 p.P
1 PP.

Введите ваш ход: c2b3
  abc
3 .Pp
2 p..
1 PP.

Вы выиграли!
Сыграем ещё? (y/n) y
```

```
  abc
3 ppp
2 ...
1 PPP

Введите ваш ход: c1c2
  abc
3 ppp
2 ..P
1 PP.

Мой ход: b3-b2

  abc
3 p.p
2 .pP
1 PP.

Введите ваш ход: a1a2
  abc
3 p.p
2 PpP
1 .P.

Вы выиграли!
Сыграем ещё? (y/n)
```

### 11.2.1. Игровая позиция

Поступим так же, как уже делали в мини-шахматах. Игровое поле будет храниться в массиве. Один элемент массива — одна горизонталь игрового поля.

Горизонтали состоят из отдельных клеток, а каждая клетка кодируется одним символом. Точкой, если клетка пустая, и латинской буквой «P», если клетку занимает пешка.

Заглавные буквы соответствуют белым пешкам, а строчные — чёрным.

C++

```
#include <string>
#include <array>

const int N = 3;
typedef std::array<std::string, N> Board;
Board board;
```

Python

```
N = 3

board = []
```

### 11.2.2. Цикл для последовательности партий

Как и в игре «Угадай животное», здесь программа всё время учится. Причём учится на своих поражениях. Следовательно, ей придётся играть множество партий.

Раз так, кроме обычного игрового цикла придётся сделать цикл, в каждой итерации которого будет проходить одна игра. То есть игровой цикл будет вложен в этот.

C++

```
int main()
{
  do
  {
    board = {"PPP", "...", "ppp"};
    ...
  }
  while (yesNo("Сыграем ещё?"));
}
```

Python

```
while True:
  board = [list('PPP'), list('...'),
  ↪ list('ppp')]
  ...
  if not yesNo('Сыграем ещё?'):
    break
```

В этой игре нам не потребуется всё время задавать игроку вопросы. Но функция с вопросами уже была написана для игры «Угадай животное», так почему бы её не переиспользовать?

C++

```
#include <iostream>

bool yesNo(const std::string &question)
{
  while (true)
  {
    std::cout << question << " (y/n) ";
    char c;
    std::cin >> c;
    if (c == 'y' || c == 'Y')
    {
      return true;
    }
    if (c == 'n' || c == 'N')
    {
      return false;
    }
  }
}
```

Python

```
def yesNo(question):
  while True:
    c = input(f'{question} (y/n) ')
    if c == 'y' or c == 'Y':
      return True
    if c == 'n' or c == 'N':
      return False
```

Правда, здесь вместо русских букв «Д» и «Н» используются английские «Y» и «N», чтобы не заставлять игрока переключать раскладку (ведь шахматные ходы тоже обозначаются с использованием латиницы).

### 11.2.3. Игровой цикл

Начнём придумывать игровой цикл. В целом он похож на любую другую игру. Выводится игровое поле, делаются ходы. И ещё как-то надо определять, завершилась ли игра и кто победил. Но к этому мы вернёмся чуть позже.

C++

```
while (true)
{
  printBoard();
  ... = playerMove();
  if (...)
  {
    break;
  }
  printBoard();
  ... = computerMove();
  if (...)
  {
    break;
  }
}
printBoard();
...
```

Python

```
while True:
  printBoard()
  ... = playerMove()
  if ...:
    break
  printBoard()
  ... = computerMove()
  if ...:
    break
printBoard()
...
```

### 11.2.4. Вывод игрового поля

Игровое поле печатается так же, как в мини-шахматах. Клетки нумеруются буквами и цифрами, а поле хранится в виде последовательности символов.

C++

```
void printBoard()
{
  std::cout << "  abc\n";
  for (int i = N - 1 ; i >= 0 ; --i)
  {
    std::cout << i + 1 << " " << board[i]
    ↪  << "\n";
  }
  std::cout << "\n";
}
```

Python

```
def printBoard():
  print('  abc')
  for i in range(N - 1, -1, -1):
    print(f' {i + 1}
    ↪  {"".join(board[i])} ')
  print()
```

### 11.2.5. Проверка окончания игры

А вот определение того, что игра закончилась, отличается от мини-шахмат, так как возможна ситуация, когда нет ни одного доступного хода.

Во время хода компьютера это не проблема, он может просто перебрать все доступные ходы. И если ни одного не нашлось, функция может вернуть какой-то особый код.

Но функция, отвечающая за ход человека, обычно ходы не перебирает. Следовательно, эту часть придётся туда добавить.

Тогда каждая из функций, отвечающей за ходы, может вернуть три возможных статуса: игра продолжается, победил игрок или победил компьютер.

Например, если у игрока нет хода, его функция вернёт код «победил компьютер», а если он провёл пешку на последнюю горизонталь, то «победил игрок».

C++

```
enum MoveResult
{
  OK,
  PLAYER,
  COMPUTER
};

...

MoveResult res = OK;
while (res == OK)
{
    printBoard();
    res = playerMove();
    if (res != OK)
    {
        break;
    }
    printBoard();
    res = computerMove();
}
printBoard();
if (res == PLAYER)
{
  std::cout << "Вы выиграли!\n";
}
else
{
  std::cout << "Я победил!\n";
}
```

Python

```
OK = 0
PLAYER = 1
COMPUTER = 2

...

res = OK
while res == OK:
  printBoard()
  res = playerMove()
  if res != OK:
    break
  printBoard()
  res = computerMove()
printBoard()
if res == PLAYER:
  print('Вы выиграли!')
else:
  print('Я победил!')
```

Одна из проверок окончания игры «переехала» в условие продолжения цикла, чтобы код был чуть компактнее.

### 11.2.6. Ход игрока

Мы уже поняли, что в отличие от других программ в этой придётся проверять, есть ли у игрока доступные ходы. Для этого нужно найти все его пешки и проверить, есть ли хотя бы у одной возможность пойти вперёд (с взятием пешки

соперника или нет). Искать пешки нужно на всех горизонталях, кроме последней. Потому что если пешка там уже оказалась на предыдущем ходу, то игрок победил ещё тогда.

За проверку того, правильный ли получается ход, отвечает функция `isValidMove` Она пригодится и для валидации введённого пользователем хода, и для такого же поиска хода компьютером.

В отличие от мини-шахмат, здесь клетка, куда пойдёт фигура, ищется только среди клеток следующей горизонтали. Потому что пешки в любом случае далеко уйти не могут.

C++

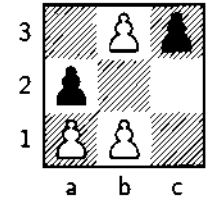
```
MoveResult playerMove()
{
  bool hasMove = false;
  for (int r = 0 ; r < N - 1 ; ++r)
  {
    for (int c = 0 ; c < N ; ++c)
    {
      if (board[r][c] == 'P')
      {
        for (int c2 = c - 1 ; c2 <= c + 1
        ↪  ; ++c2)
        {
          if (isValidMove('P', c, r, c2,
          ↪  r + 1))
          {
            hasMove = true;
          }
        }
      }
    }
  }
  if (!hasMove)
  {
    return COMPUTER;
  }
  ...
}
```

Python

```
def playerMove():
  hasMove = False
  for r in range(N - 1):
    for c in range(N):
      if board[r][c] == 'P':
        for c2 in range(c - 1, c + 2):
          if isValidMove('P', c, r, c2, r
          ↪  + 1):
            hasMove = True
  if not hasMove:
    return COMPUTER
  ...
```

Когда хотя бы один ход найден, дадим возможность игроку ввести его (или любой другой ход) с клавиатуры. Эта часть почти не отличается от мини-шахмат.

А если выбранная пешка дошла до конца, функция `playerMove` должна вернуть константу `PLAYER`, что означает победу игрока:



C++

```
char c1, r1, c2, r2;
do
{
  std::cout << "Введите ваш ход: ";
  std::cin >> c1 >> r1 >> c2 >> r2;
  c1 -= 'a';
  r1 -= '1';
  c2 -= 'a';
  r2 -= '1';
}
while (!isValidMove('P', c1, r1, c2,
↪  r2));
makeMove(board, c1, r1, c2, r2);
if (r2 == N - 1)
{
  return PLAYER;
}
return OK;
```

Python

```
c1 = 0
r1 = 0
c2 = 0
r2 = 0
while not isValidMove('P', c1, r1, c2,
↪  r2):
  line = input('Введите ваш ход: ')
  c1, r1, c2, r2 = list(line)
  c1 = ord(c1) - ord('a')
  r1 = ord(r1) - ord('1')
  c2 = ord(c2) - ord('a')
  r2 = ord(r2) - ord('1')
makeMove(board, c1, r1, c2, r2)
if r2 == N - 1:
  return PLAYER
return OK
```

### 11.2.7. Функция завершения хода

Для того чтобы игрок мог делать ходы, не хватает двух функций: `isValidMove` и `makeMove`. Начнём со второй, она попроще.

Всего-то и нужно скопировать содержимое клетки с пешкой в новое место (это может быть как пешка игрока, так и пешка компьютера), а в старую позицию записать точку (то есть пустоту).

Никакой счёт вести не нужно, потому что оценки позиции в этой игре не будет.

В отличие от других функций, которые мы уже написали, здесь в качестве первого параметра передаётся игровое поле. Для хода игрока это не нужно, можно было бы и глобальную переменную менять, а вот для хода компьютера понадобится.

C++

```
void makeMove(Board &board, int c1, int
↪  r1, int c2, int r2)
{
  board[r2][c2] = board[r1][c1];
  board[r1][c1] = '.';
}
```

Python

```
def makeMove(board, c1, r1, c2, r2):
  board[r2][c2] = board[r1][c1]
  board[r1][c1] = '.'
```

### 11.2.8. Проверка корректности хода

Так как функция должна проверять и почти корректные ходы (полученные в результате перебора), и потенциально некорректные (которые ввёл игрок), нужно проверять ход на все возможные ошибки.

C++
```
bool isValidMove(char player, int c1, int
↪  r1, int c2, int r2)
{
  return false;
}
```

Python
```
def isValidMove(player, c1, r1, c2, r2):
  return False
```

Сначала это будут условия, что начальная и конечная клетки находятся в пределах поля и не совпадают.

C++
```
if (c1 < 0 || c1 >= N
  || r1 < 0 || r1 >= N
  || c2 < 0 || c2 >= N
  || r2 < 0 || r2 >= N)
{
  return false;
}
if (c1 == c2 && r1 == r2)
{
  return false;
}
```

Python
```
if c1 < 0 or c1 >= N
  or r1 < 0 or r1 >= N
  or c2 < 0 or c2 >= N
  or r2 < 0 or r2 >= N:
    return False
if c1 == c2 and r1 == r2:
  return False
```

Следующее условие проверяет, что игрок ходит собственной пешкой. Здесь для простоты переменная **player** хранит не число 0 или 1, а соответствующий символ пешки — заглавную или строчную букву «P». Тогда если этот символ совпадает с выбранной пешкой, то всё в порядке.

C++
```
char f = board[r1][c1];
if (f != player)
{
  return false;
}
```

Python
```
f = board[r1][c1]
if f != player:
  return False
```

Дальше мы проверяем, что пешка перемещается только на одну клетку вперёд. Для пешек игрока это направление увеличения номера горизонтали, а для компьютера — уменьшения.

C++
```
int dy = f == 'P' ? 1 : -1;
if (r1 + dy != r2)
{
  return false;
}
```

Python
```
dy = 1 if f == 'P' else -1
if r1 + dy != r2:
  return False
```

Ну и наконец добавим функции возможность вернуть значение **true**. Это произойдёт, если делается ход в пустую клетку вперёд или в занятую по диагонали. Причём занята она должна быть чужой пешкой.

C++
```
char t = board[r2][c2];
if (c1 == c2)
{
  return t == '.';
}
if (c1 + 1 == c2 || c1 - 1 == c2)
{
  return t != '.' && f != t;
}
```

Python
```
t = board[r2][c2]
if c1 == c2:
  return t == '.'
if c1 + 1 == c2 or c1 - 1 == c2:
  return t != '.' and f != t
```



Кажется, после появления этой функции уже можно поиграть. Протестируйте, правильно ли вводятся ходы и корректно ли работает проверка возможности хода.

### 11.2.9. Ход компьютера

Функция для хода компьютера должна выбирать первый попавшийся ход. Ведь мы решили не придумывать стратегию для программы, а заставить компьютер найти её самостоятельно.

C++
```
MoveResult computerMove()
{
  ...
  return OK;
}
```

Python
```
def computerMove():
  ...
  return OK
```

Поэтому просто перебираем все возможные ходы:

C++
```
int mc = -1, mc2;
for (int r = 1 ; r < N ; ++r)
{
  for (int c = 0 ; c < N ; ++c)
  {
    if (board[r][c] == 'p')
    {
      for (int c2 = c - 1 ; c2 <= c + 1 ;
      ↪  ++c2)
      {
        if (isValidMove('p', c, r, c2, r
        ↪  - 1))
        {
          mc = c;
          mr = r;
          mc2 = c2;
        }
      }
    }
  }
}
```

Python
```
mc = -1
for r in range(1, N):
  for c in range(0, N):
    if board[r][c] == 'p':
      for c2 in range(c - 1, c + 2):
        if isValidMove('p', c, r, c2, r -
        ↪  1):
          mc = c
          mr = r
          mc2 = c2
```

После чего проверяем, что ход нашёлся, и двигаем найденную пешку:

C++

```
if (mc < 0)
{
  return PLAYER;
}
makeMove(board, mc, mr, mc2, mr - 1);
std::cout << "Мой ход: " << (char)(mc +
↪  'a') << mr + 1
  << '-' << (char)(mc2 + 'a') << mr <<
  ↪  "\n\n";
if (mr == 1)
{
  return COMPUTER;
}
```

Python

```
if mc < 0:
  return PLAYER
makeMove(board, mc, mr, mc2, mr - 1)
print(f'Мой ход: {chr(mc + ord('a'))}{mr
↪  + 1}-{chr(mc2 + ord('a'))}{mr}\n')
if mr == 1:
  return COMPUTER
```

Получилась функция почти как для игрока. Только вместо ввода координат для хода запоминаем их во время перебора пешек и их возможных перемещений.

Вот компьютер и научился играть. Удаётся ли ему победить со своей стратегией первого попавшегося хода?

### 11.2.10. Хранение недопустимых позиций

Чтобы программа училась, нужно где-то хранить «плохие» позиции. Такие, куда компьютер будет стараться не попадать после своего хода.

Для этого заведём хранилище типа «множество». Всевозможных позиций немного, поэтому будет легко преобразовать игровое поле в уникальное целое число, а полученные числа уже добавлять в множество.

C++

```
#include <unordered_set>

std::unordered_set<int> bad;

int boardCode(const Board &board)
{
  int res = 0;
  for (auto row : board)
  {
    for (auto c : row)
    {
      res *= 3;
      if (c == 'p')
      {
        res += 1;
      }
      if (c == 'P')
      {
        res += 2;
      }
    }
  }
  return res;
}
```

Python

```
bad = set()

def boardCode(board):
  res = 0
  for row in board:
    for c in row:
      res *= 3
      if c == 'p':
        res += 1
      if c == 'P':
        res += 2
  return res
```

Принцип кодирования позиции примерно такой же, как в программе для игры «Восемь».

Представим описание поля как число в системе счисления с основанием 3. Одна цифра такого числа отвечает за одну клетку поля. Если клетка пустая, то используется цифра 0. Если там пешка компьютера, то цифра 1, а если пешка игрока, то цифра 2.

Тогда, чтобы посчитать уникальный код позиции, нужно перебрать все клетки поля, добавляя в получающееся число нужные цифры. Так как преобразование в обратную сторону нам не нужно, можно даже не задумываться, какой именно порядок клеток используется во время перебора.

### 11.2.11. Добавление и поиск недопустимых позиций

Мы уже договаривались, что в набор «плохих» позиций будут попадать те, которые программа достигла в последнюю очередь. Для этого пригодится новая переменная `last`.

Там функция `computerMove` будет сохранять последнюю достигнутую позицию, которая не считается плохой. А когда игра закончится, то если компьютер проиграл, эту позицию (точнее, её уникальный код) нужно добавить в набор недопустимых, чтобы больше такие ходы не выбирать.

C++

```
Board last;

...
if (res == PLAYER)
{
  std::cout << "Вы выиграли!\n";
  bad.insert(boardCode(last));
}
...
```

Python

```
last = []

...
if res == PLAYER:
  print('Вы выиграли!')
  bad.add(boardCode(last))
...
```

Добавление в множество есть, а самого запоминания позиции в переменной `last` пока нет. Нужно доработать функцию `computerMove`, ведь именно там выбираются ходы.

Тут придётся повозиться побольше. Чтобы не попасть в ситуацию, когда компьютер думает, что у него нет ходов, несмотря на то что они есть (просто все плохие), надо предусмотреть, чтобы плохой ход всё же выбирался на всякий случай.

А вот когда найден ход получше (не приводящий к уже известному поражению), то найденный ход можно заменить новым. Поэтому нам нужна дополнительная переменная `foundBad`. Изначально там записано `true`, а сбрасывается в `false` оно только тогда, когда найдена новая позиция, которой нет в контейнере `bad`.

В самом конце функции, если был найден хороший ход, то есть `foundBad == false`, мы записываем его в переменную `last`. И если в конце игры окажется, что этот ход привёл к поражению, он-то и будет записан в множество `bad`.

C++

```
MoveResult computerMove()
{
  bool foundBad = true;
  ...
    if (isValidMove('p', c, r, c2, r -
    ↪ 1))
    {
      if (foundBad)
      {
        mc = c;
        mr = r;
        mc2 = c2;
      }
      Board b = board;
      makeMove(b, c, r, c2, r - 1);
      if (!bad.contains(boardCode(b)))
      {
        foundBad = false;
      }
    }
  ...
  if (!foundBad)
  {
    last = board;
  }
  ...
}
```

Python

```
import copy

def computerMove():
  global last
  foundBad = True
  ...
    if isValidMove('p', c, r, c2, r - 1):
      if foundBad:
        mc = c
        mr = r
        mc2 = c2
      b = copy.deepcopy(board)
      makeMove(b, c, r, c2, r - 1)
      if boardCode(b) not in bad:
        foundBad = False
  ...
  if not foundBad:
    last = copy.deepcopy(board)
  ...
```

Программа готова. Сколько всего партий вы сможете выиграть у неё, пока она не станет непобедимой?

### 11.2.12. Задания для самостоятельной работы

**1.** При проигрыше компьютера в «плохие» позиции записывайте не только последнюю позицию, но и симметричную ей. Так программа будет учиться быстрее.

**2.** Расширьте игровое поле до размеров 4 × 4. Так компьютер будет учиться дольше, поэтому играть станет интереснее.

**3.** Сделайте так, чтобы компьютер выбирал случайные ходы, а не первые попавшиеся. Тогда станет чуть интереснее, ведь разные сеансы игры будут проходить по-разному.

**4.** Шесть пешек — это очень простая игра. Изначально такой же метод самообучения использовался для игры крестики-нолики [2]. Так как мы пишем программу, а не делаем самообучающуюся машину из спичечных коробков, как в книге, можно рассматривать намного более сложные игры. Попробуйте пере-

работать программу, чтобы она играла в крестики нолики $4 \times 4$ или даже в мини-шахматы.

### Листинг 11.3. hexapawn.cpp

```
#include <iostream>
#include <string>
#include <array>
#include <unordered_set>

const int N = 3;

enum MoveResult
{
    OK,
    PLAYER,
    COMPUTER
};

typedef std::array<std::string, N> Board;
std::unordered_set<int> bad;

Board board;
Board last;

int boardCode(const Board &board)
{
    int res = 0;
    for (auto row : board)
    {
        for (auto c : row)
        {
            res *= 3;
            if (c == 'p')
            {
                res += 1;
            }
            if (c == 'P')
            {
                res += 2;
```

```
            }
        }
    }
    return res;
}


bool yesNo(const std::string &question)
{
    while (true)
    {
        std::cout << question << " (y/n) ";
        char c;
        std::cin >> c;
        if (c == 'y' || c == 'Y')
        {
            return true;
        }
        if (c == 'n' || c == 'N')
        {
            return false;
        }
    }
}

void printBoard()
{
    std::cout << "  abc\n";
    for (int i = N - 1 ; i >= 0 ; --i)
    {
        std::cout << i + 1 << " " << board[i] << "\n";
    }
    std::cout << "\n";
}

bool isValidMove(char player, int c1, int r1, int c2, int r2)
{
    if (c1 < 0 || c1 >= N || r1 < 0 || r1 >= N
        || c2 < 0 || c2 >= N || r2 < 0 || r2 >= N)
    {
        return false;
```

```
    }
    if (c1 == c2 && r1 == r2)
    {
        return false;
    }
    char f = board[r1][c1];
    if (f != player)
    {
        return false;
    }
    int dy = f == 'P' ? 1 : -1;
    if (r1 + dy != r2)
    {
        return false;
    }
    char t = board[r2][c2];
    if (c1 == c2)
    {
        return t == '.';
    }
    if (c1 + 1 == c2 || c1 - 1 == c2)
    {
        return t != '.' && f != t;
    }
    return false;
}

void makeMove(Board &board, int c1, int r1, int c2, int r2)
{
    board[r2][c2] = board[r1][c1];
    board[r1][c1] = '.';
}

MoveResult computerMove()
{
    bool foundBad = true;
    int mc = -1, mr, mc2;
    for (int r = 1 ; r < N ; ++r)
    {
        for (int c = 0 ; c < N ; ++c)
        {
```

```
            if (board[r][c] == 'p')
            {
                for (int c2 = c - 1 ; c2 <= c + 1 ; ++c2)
                {
                    if (isValidMove('p', c, r, c2, r - 1))
                    {
                        if (foundBad)
                        {
                            mc = c;
                            mr = r;
                            mc2 = c2;
                        }
                        Board b = board;
                        makeMove(b, c, r, c2, r - 1);
                        if (!bad.contains(boardCode(b)))
                        {
                            foundBad = false;
                        }
                    }
                }
            }
        }
    }
    if (mc < 0)
    {
        return PLAYER;
    }
    makeMove(board, mc, mr, mc2, mr - 1);
    if (!foundBad)
    {
        last = board;
    }
    std::cout << "Мой ход: " << (char)(mc + 'a') << mr + 1
        << '-' << (char)(mc2 + 'a') << mr << "\n\n";
    if (mr == 1)
    {
        return COMPUTER;
    }
    return OK;
}
```

```
MoveResult playerMove()
{
    bool hasMove = false;
    for (int r = 0 ; r < N - 1 ; ++r)
    {
        for (int c = 0 ; c < N ; ++c)
        {
            if (board[r][c] == 'P')
            {
                for (int c2 = c - 1 ; c2 <= c + 1 ; ++c2)
                {
                    if (isValidMove('P', c, r, c2, r + 1))
                    {
                        hasMove = true;
                    }
                }
            }
        }
    }
    if (!hasMove)
    {
        return COMPUTER;
    }
    char c1, r1, c2, r2;
    do
    {
        std::cout << "Ваедите ваш ход: ";
        std::cin >> c1 >> r1 >> c2 >> r2;
        c1 -= 'a';
        r1 -= '1';
        c2 -= 'a';
        r2 -= '1';
    }
    while (!isValidMove('P', c1, r1, c2, r2));
    makeMove(board, c1, r1, c2, r2);
    if (r2 == N - 1)
    {
        return PLAYER;
    }
    return OK;
}
```

```
int main()
{
    do
    {
        last = Board();
        board = {"PPP", "...", "ppp"};
        MoveResult res = OK;
        while (res == OK)
        {
            printBoard();
            res = playerMove();
            if (res != OK)
            {
                break;
            }
            printBoard();
            res = computerMove();
        }
        printBoard();
        if (res == PLAYER)
        {
            std::cout << "Вы выиграли!\n";
            bad.insert(boardCode(last));
        }
        else
        {
            std::cout << "Я победил!\n";
        }
    }
    while (yesNo("Сыграем ещё?"));
}
```

**Листинг 11.4. hexapawn.py**

```
#!/usr/bin/python3
import copy

N = 3

OK = 0
PLAYER = 1
COMPUTER = 2

bad = set()

board = []
last = []

def boardCode(board):
    res = 0
    for row in board:
        for c in row:
            res *= 3
            if c == 'p':
                res += 1
            if c == 'P':
                res += 2
    return res


def yesNo(question):
    while True:
        c = input(f'{question} (y/n) ')
        if c == 'y' or c == 'Y':
            return True
        if c == 'n' or c == 'N':
            return False

def printBoard():
    print('  abc')
    for i in range(N - 1, -1, -1):
        print(f'{i + 1} {"".join(board[i])}')
    print()
```

```
def isValidMove(player, c1, r1, c2, r2):
    if c1 < 0 or c1 >= N or r1 < 0 or r1 >= N or c2 < 0 or c2 >= N or
    ↪ r2 < 0 or r2 >= N:
        return False
    if c1 == c2 and r1 == r2:
        return False

    f = board[r1][c1]
    if f != player:
        return False

    dy = 1 if f == 'P' else -1
    if r1 + dy != r2:
        return False

    t = board[r2][c2]
    if c1 == c2:
        return t == '.'
    if c1 + 1 == c2 or c1 - 1 == c2:
        return t != '.' and f != t
    return False

def makeMove(board, c1, r1, c2, r2):
    board[r2][c2] = board[r1][c1]
    board[r1][c1] = '.'

def computerMove():
    global last
    foundBad = True
    mc = -1
    for r in range(1, N):
        for c in range(0, N):
            if board[r][c] == 'p':
                for c2 in range(c - 1, c + 2):
                    if isValidMove('p', c, r, c2, r - 1):
                        if foundBad:
                            mc = c
                            mr = r
                            mc2 = c2
                        b = copy.deepcopy(board)
                        makeMove(b, c, r, c2, r - 1)
```

```
                        if boardCode(b) not in bad:
                            foundBad = False
    if mc < 0:
        return PLAYER
    makeMove(board, mc, mr, mc2, mr - 1)
    if not foundBad:
        last = copy.deepcopy(board)
    print(f'Мой ход: {chr(mc + ord('a'))}{mr + 1}-{chr(mc2 +
    ↪ ord('a'))}{mr}\n')
    if mr == 1:
        return COMPUTER
    return OK

def playerMove():
    hasMove = False
    for r in range(N - 1):
        for c in range(N):
            if board[r][c] == 'P':
                for c2 in range(c - 1, c + 2):
                    if isValidMove('P', c, r, c2, r + 1):
                        hasMove = True
    if not hasMove:
        return COMPUTER
    c1 = 0
    r1 = 0
    c2 = 0
    r2 = 0
    while not isValidMove('P', c1, r1, c2, r2):
        line = input('Введите ваш ход: ')
        c1, r1, c2, r2 = list(line)
        c1 = ord(c1) - ord('a')
        r1 = ord(r1) - ord('1')
        c2 = ord(c2) - ord('a')
        r2 = ord(r2) - ord('1')
    makeMove(board, c1, r1, c2, r2)
    if r2 == N - 1:
        return PLAYER
    return OK

while True:
    last = []
```

```
board = [list('PPP'), list('...'), list('ppp')]
res = OK
while res == OK:
    printBoard()
    res = playerMove()
    if res != OK:
        break
    printBoard()
    res = computerMove()
printBoard()
if res == PLAYER:
    print('Вы выиграли!')
    bad.add(boardCode(last))
else:
    print('Я победил!')
if not yesNo('Сыграем ещё?'):
    break
```

# Список литературы


1. *Ahl D. H.* 101 BASIC Computer Games. — Maynard, Massachusetts: Digital Equipment Corporation, 1975.
2. *Гарднер М.* Математические досуги. — М.: Оникс, 1995.
3. *Гик Е. Я.* Занимательные математические игры. — М.: Знание, 1982.
4. *Довгалюк П. М.* Динамическое программирование и все-все-все. — М.: Ленанд, 2025.
5. *Златопольский Д. М.* Занимательная информатика. — М.: БИНОМ. Лаборатория знаний, 2011.
6. Калейдоскоп игр. — Лениздат, 1990.
7. Кибернетика. Микрокалькуляторы в играх и задачах. — М.: Наука, 1986.
8. *Левитин А., Левитина М.* Алгоритмические головоломки. — М.: Лаборатория знаний, 2018.
9. Математические новеллы. — М.: Мир, 1974.
10. *Трохименко Я. К., Любич Ф. Д.* Микрокалькулятор, Ваш ход! — М.: Радио и связь, 1985.
11. *Уэзерелл Ч.* Этюды для программистов. — М.: Мир, 1982.

# БАЗОВЫЕ АЛГОРИТМЫ

## Реализация на Python и C++ на примере классических игр

**Алгоритмическая подготовка, осведомленность и эрудиция — это ключевые достоинства программиста, нацеленного на развитие, карьерный рост, рассчитывающего быстро находить решения как для рутинных, так и для нетривиальных задач. В настоящее время алгоритмы в университетском курсе преподаются в основном на двух языках программирования — C++ и Python. Фундаментальные книги по алгоритмам есть в библиотеке любого серьезного программиста, однако не так много найдется простых книг, дающих базовое представление о «джентльменском наборе» алгоритмов и позволяющих сразу перейти к их реализации. Именно эту нишу и заполняет предлагаемая книга. Вы изучите:**

- генерацию случайных чисел на примере броска игральной кости;
- работу с прямоугольными координатами на примере игры «Ферзя в угол»;
- эвристический выбор на примере игры «Морской бой»;
- рекурсивный перебор на примере игры «Калах»;
- деревья решений на примере игры «Угадай животное»;
- и другое.

Алгоритмы предлагаются в порядке усложнения и могут быть использованы как для обучения и самообучения, так и для подготовки к собеседованиям, решения несложных задач при разработке реальных проектов, например, проверки концепций, прототипирования логики и непосредственно для программирования простых игр с графическим пользовательским интерфейсом.

**Павел Михайлович Довгалюк** работает в Институте системного программирования имени В.П. Иванникова РАН, преподает в Новгородском государственном университете имени Ярослава Мудрого. Имеет 25 лет стажа в программировании, 20 лет — в обучении школьников и студентов олимпиадному программированию. Разработал и построил компьютер на электромагнитных реле. Занимается компиляторными технологиями, анализом кода, инструментами для информационной безопасности.

ISBN 978-5-9775-2100-0

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®