Полиморфизм

М. А. Ложников

24 октября 2019 г.
Ревизия: 1

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Предварительная информация

1.1 Строковый поток std::ostringstream

Помимо стандартного потока вывода `std::cout` — объекта типа `std::ostream` в C++ можно создавать свои собственные потоки вывода, например, поток вывода в строку или в файл. Поток вывода в строку создаётся при помощи типа данных `std::ostringstream`, определённого в файле `<sstream>`. Класс `std::ostringstream` является производным от класса `std::ostream`.

```
#include <iostream>
#include <sstream>

int main() {
  std::ostringstream strm; // Объявляем строковый поток strm.
  int a = 3;
  double pi = 3.14;

  /* С этими потоками работают как с std::cout, только вывод происходит не на экран,
     а в строку. */
  strm << "a = " << a << " pi = " << pi;

  // Метод str() позволяет получить записанную строку.
  std::string writtenStr = strm.str();

  // Выводим полученную строку на экран.
  std::cout << "written string = '" << writtenStr << "'" << std::endl;
  return 0;
}
```

2 Полиморфизм

Предположим, что есть некоторый интерфейс, который работает с классом `Base`, а также с любыми его наследниками (производными классами), например,

```
void Function(Base* base) {
  base->IdentifyYourself();
}
```

Для того, чтобы всё работало корректно, нужно определить метод `IdentifyYourself()` с доступом `public` в классе `Base`, например,

```
class Base {
 public:
  void IdentifyYourself() {
    std::cout << "I am the base class!" << std::endl;
  }
};
```

Однако в некоторых случаях хотелось бы, чтобы производные классы могли переопределять поведение некоторых методов базовых классов. Для этого существует механизм виртуальных функций. Для того, чтобы функция базового класса замещалась функцией производного класса, функция базового класса должна быть объявлена с ключевым словом `virtual` и в производном классе должна быть функция с точно таким же прототипом. То есть

```
class Base {
 public:
  virtual void IdentifyYourself() {
    std::cout << "I am the base class!" << std::endl;
  }
};

/* В C++11 рекомендуется помечать методы, которые должны замещать соответствующие методы
   базового класса ключевым словом override для того, чтобы компилятор выдавал ошибку
   в том случае, если в базовом классе отсутствует метод, который необходимо
   переопределить. Это помогает избежать ошибок при рефакторинге кода. В старых версиях
   C++ этого ключевого слова нет. */
class Derived : public Base {
 public:
  void IdentifyYourself() override {
    std::cout << "I am a derived class!" << std::endl;
  }
};

/* В таком случае в функции Function() будет вызываться метод IdentifyYourself()
   производного класса. Как быть, если всё-таки хочется вызвать виртуальный метод
   базового класса? */
void AnotherFunction(Base* base) {
  /* Если бы мы не пометили метод IdentifyYourself() базового класса как virtual,
     то следующие две строчки были бы эквивалентны, то есть всегда вызывался бы метод
     базового класса. */

  // Вызовется метод производного класса, если он переопределён.
  base->IdentifyYourself();
```

```
30
31   /* Следующая конструкция позволяет всегда вызывать метод базового класса, даже если
32      он помечен как виртуальный. */
33   base->Base::IdentifyYourself();
34 }
```

Определение 2.1 *Бьерн Страуструп: «"Правильное" поведение виртуальных функций* `IdentifyYourself()` *для объектов типа* `Base`, `Derived` *и прочих производных классов называется полиморфизмом (polymorphism). Тип с виртуальными функциями называется полиморфным типом (polymorphic type). Для практической реализации полиморфного поведения в языке C++ тип объектов должен быть полиморфным, а сами объекты должны адресоваться указателями или ссылками» (цитата отредактирована автором под данный конкретный пример).*

Замечание 2.1 *Отметим, что следует избегать определения всех виртуальных функций класса внутри его объявления из-за того, что для функций, определённых внутри срабатывает inline подстановка. Таким образом для каждого случая inline подстановки будет генерироваться своя таблица виртуальных функций. Иными словами, хотя бы одну виртуальную функцию следует определять out-of-line, то есть снаружи объявления класса. В данных примерах этот недочёт проигнорирован в целях экономии места. Программа, приведённая в конце лекции лишена данного недостатка.*

3 Абстрактные классы

В некоторых случаях реализация виртуального метода в базовом класса не имеет смысла с точки зрения логики. Однако, своя собственная реализация этого метода необходима в каждом производном классе. То есть базовый класс по замыслу представляет некоторый абстрактный интерфейс, который в чистом виде никогда не должен использоваться, однако, вместо него будут использоваться производные классы.

```
1  // Абстрактный класс Base.
2  class Base {
3   public:
4    /* = 0 в конце говорит о том, что метод является чисто виртуальным (абстрактным)
5       Он не должен быть определён в базовом классе, однако, производный класс
6       обязан его определить. */
7    virtual void PureVirtualMethod() = 0;
8  };
9  /* Класс, содержащий абстрактные методы, называется виртуальным.
10    Попытка создать объект абстрактного класса вызывает ошибку компиляции.
11 // Base base; вызовет ошибку компиляции */
12
13 class Derived : public Base {
14  public:
15   virtual void PureVirtualMethod() {
16     // Определение метода
17   }
18 };
19
20 /* Можно создавать объекты производных классов, в которых определены абстрактные методы
21    базовых классов, а потом приводить их к типу базового класса по ссылке или
```

```
22    по указателю. */
23 void Function(Base& base) {
24   base.PureVirtualMethod(); // Корректно
25 }
26
27 // Derived derived;
28 // Function(derived); -- Корректно
```

3.1 Виртуальный деструктор

Рассмотрим следующий пример

```
1  #include <iostream>
2  #include <string>
3
4  // Класс для вывода сообщений о срабатывании конструктора и деструктора.
5  class Log {
6   public:
7    Log(const std::string& id = "") :
8      id(id) {
9        std::cerr << "Log constructed (id = " << id << ")!" << std::endl;
10     }
11   ~Log() {
12     std::cout << "Log destructed (id = " << id << ")!" << std::endl;
13   }
14  private:
15   const std::string id;
16 };
17
18 // Плохая реализация базового класса.
19 class BadBase {
20  public:
21   BadBase() :
22     log("bad base")
23   { }
24  private:
25   Log log;
26 };
27
28 // Реализация наследника.
29 class BadDerived : public BadBase {
30  public:
31   BadDerived() :
32     log("bad derived")
33   { }
34  private:
35   Log log;
36 };
37
38 // Хорошая реализация базового класса, у неё есть виртуальный деструктор
```

```
39 class Base {
40  public:
41   Base() :
42     log("base")
43   { }
44
45   virtual ~Base() {
46   }
47  private:
48    Log log;
49 };
50
51 // Реализация наследника.
52 class Derived : public Base{
53  public:
54   Derived() :
55     log("derived")
56   { }
57  private:
58   Log log;
59 };
60
61 void BadFunction(BadBase* badBase) {
62   std::cout << "Calling function BadFunction()" << std::endl;
63   delete badBase;  // Удаление объекта.
64   std::cout << "Function BadFunction() finished!" << std::endl;
65 }
66
67 void Function(Base* base) {
68   std::cout << "Calling function Function()" << std::endl;
69   delete base;  // Удаление объекта.
70   std::cout << "Function Function() finished!" << std::endl;
71 }
72
73 int main() {
74   BadDerived* badDerived = new BadDerived();
75   Derived* derived = new Derived();
76
77   BadFunction(badDerived);
78   Function(derived);
79   return 0;
80 }
```

В данном примере не отработает деструктор производного класса `BadDerived`. Оператор `delete` в функции `BadFunction()` вызовет лишь деструктор базового класса `BadBase`, поскольку он не является виртуальным и не обеспечивает полиморфного поведения. Однако, в функции `Function()` будет вызван деструктор производного класса `Derived`, который в свою очередь вызовет деструктор базового класса `Base`. Это достигается за счёт того, что в базовом классе `Base` деструктор объявлен как виртуальный, следовательно, он ведёт себя полиморфно. Таким образом, в базовых классах, необходимо объявлять виртуальный деструктор.

Замечание 3.1 *Имеет смысл делать деструктор виртуальным и в тех классах, которые в дан-*

ный момент не являются базовыми, поскольку в дальнейшем на основе них могут быть созданы производные классы.

Замечание 3.2 *Если некоторый метод производного класса переопределяет виртуальный метод базового класса, то соответствующий метод производного класса неявно помечается как виртуальный. Таким образом, для таких методов не нужно указывать слово* `virtual`.

4 Программа шахматы

4.1 Файл cell.hpp

```
#ifndef CELL_HPP
#define CELL_HPP

#include <sstream>    // Нужно для работы класса std::ostringstream.
#include <stdexcept>  // Здесь определены стандартные классы исключений.

const int boardSize = 8; // Константа, задающая размер шахматной доски.

// Этот класс нужен для того, чтобы хранить разность клеток шахматной доски.
class Delta {
 public:
  Delta(int x, int y) :
    x(x),
    y(y)
  { }

  int X() const { return x; } // Методы для чтения private полей.
  int Y() const { return y; }

 private:
  int x;
  int y;
};

// Этот класс предназначен для описания клетки шахматной доски.
class Cell {
 public:
  Cell(int x, int y) {
    if (x < 0 || x >= boardSize ||
        y < 0 || y >= boardSize) {
      /* Выбрасываем исключение std::runtime_error, если оказались за пределами доски.
         std::ostringstream --- класс, описывающий поток вывода в строку.
         С объектами типа std::ostringstream работают так же, как с std::cout.
         Класс std::ostringstream является производным от класса std::ostream. */
      std::ostringstream reasonStream;
      reasonStream << "Incorrect coordinates (" << x << ", " << y << ")!";

      // Записанную строку можно получить с помощью метода str().
      throw std::runtime_error(reasonStream.str());
```

```
40      }
41
42      this->x = x;
43      this->y = y;
44    }
45
46    /* Разность двух клеток описывается классом Delta. Отличие в том, что класс
47       Delta не осуществляет проверок, т.к. разность клеток может быть отрицательной. */
48    Delta operator-(const Cell& other) const {
49      return Delta(x - other.x, y - other.y);
50    }
51
52    // Сумма клетки и разницы равняется клетке.
53    Cell operator+(const Delta& delta) const {
54      return Cell(x + delta.X(), y + delta.Y());
55    }
56
57    int X() const { return x; } // Методы для чтения private полей.
58    int Y() const { return y; }
59
60   private:
61    int x;  // Координаты клетки.
62    int y;
63  };
64
65  /* Хотим упростить себе жизнь и выводить клетку с помощью любого потока вывода.
66     Можно использовать объект std::cout (тип std::ostream), а также объекты типа
67     std::ostringstream (запись в строку) и std::ofstream (запись в файл),
68     поскольку они являются производными от базового класса std::ostream. */
69  inline std::ostream& operator<<(std::ostream& out, const Cell& cell) {
70    /* С базовыми потоками вывода работают как с std::cout, который является потоком
71       вывода, предназначенным для вывода на экран. Объект std::cout имеет тип
72       std::ostream. */
73    out << "(" << cell.X() << ", " << cell.Y() << ")";
74    return out;
75  }
76
77  #endif // CELL_HPP
```

4.2 Файл chess.hpp

```
1  #ifndef CHESS_HPP
2  #define CHESS_HPP
3
4  #include <ostream>
5  #include <string>
6
7  #include "cell.hpp"
8
```

```
// Базовый класс для любой шахматной фигуры.
class ChessPiece {
 public:
  ChessPiece(const std::string& type, const std::string& team, const Cell& pos) :
    type(type), // Тип фигуры (пешка, ладья, слон и т.д.).
    team(team), // Команда (белые, чёрные).
    pos(pos)    // Клетка, в которой она расположена.
  { }

  /* Виртуальный деструктор нужен для того, чтобы при удалении указателя на объект
     базового класса посредством оператора delete вызывался деструктор производного
     класса, который автоматически вызывает деструктор базового класса. В противном
     случае деструктор производного класса не отработает, что может привести к утечкам
     памяти. */
  virtual ~ChessPiece() {

  }

  void Move(const Cell& newPos);     // Переставить фигуру на свободную клетку.

  void Eat(const ChessPiece& other); // Съесть фигуру.

  /* Абстрактная функция для проверки корректности хода. Она своя у каждого
     производного класса. */
  virtual bool IsMoveCorrect(const Cell& newPos) const = 0;

  /* Виртуальная функция для проверки возможности съесть другую фигуру. Эта функция
     одинаковая у всех фигур кроме пешки. Поэтому мы определим общий вариант в базовом
     классе, а класс пешки переопределит его своим собственным вариантом. */
  virtual bool CanEat(const ChessPiece& other) const;

  const std::string& Type() const { return type; }
  std::string& Type() { return type; }

  const std::string& Team() const { return team; }
  std::string& Team() { return team; }

  const Cell& Pos() const { return pos; }
  Cell& Pos() { return pos; }

 protected:
  std::string type;
  std::string team;
  Cell pos;
};

// Класс пешки.
class Pawn : public ChessPiece
{
 public:
  Pawn(const std::string& team, const Cell& pos) :
```

```
60      ChessPiece("pawn", team, pos)
61    { }
62
63    /* В C++11 рекомендуется помечать методы, которые должны замещать соответствующие
64       методы базового класса ключевым словом override для того, чтобы компилятор
65       выдавал ошибку в том случае, если в базовом классе отсутствует метод, который
66       необходимо переопределить. Это помогает избежать ошибок при рефакторинге кода.
67       В старых версиях C++ этого ключевого слова нет. */
68    bool IsMoveCorrect(const Cell& newPos) const override;
69
70    bool CanEat(const ChessPiece& other) const override;
71  };
72
73  // Класс коня.
74  class Knight : public ChessPiece
75  {
76  public:
77    Knight(const std::string& team, const Cell& pos) :
78      ChessPiece("knight", team, pos)
79    { }
80
81    bool IsMoveCorrect(const Cell& newPos) const override;
82  };
83
84  // Класс слона.
85  class Bishop : public ChessPiece
86  {
87  public:
88    Bishop(const std::string& team, const Cell& pos) :
89      ChessPiece("bishop", team, pos)
90    { }
91
92    bool IsMoveCorrect(const Cell& newPos) const override;
93  };
94
95  // Класс ладьи.
96  class Castle : public ChessPiece
97  {
98  public:
99    Castle(const std::string& team, const Cell& pos) :
100     ChessPiece("castle", team, pos)
101   { }
102
103   bool IsMoveCorrect(const Cell& newPos) const override;
104 };
105
106 // Класс ферзя.
107 class Queen : public ChessPiece
108 {
109 public:
110   Queen(const std::string& team, const Cell& pos) :
```

```
111     ChessPiece("queen", team, pos)
112   { }
113
114   bool IsMoveCorrect(const Cell& newPos) const override;
115 };
116
117 // Класс короля.
118 class King : public ChessPiece
119 {
120 public:
121   King(const std::string& team, const Cell& pos) :
122     ChessPiece("king", team, pos)
123   { }
124
125   bool IsMoveCorrect(const Cell& newPos) const override;
126 };
127
128 // Оператор для вывода произвольной фигуры в поток.
129 std::ostream& operator<<(std::ostream& out, const ChessPiece& piece);
130
131 // Класс шахматной доски
132 class ChessBoard {
133  public:
134   ChessBoard();
135
136   /* Поскольку класс хранит указатели, то он требует правильной реализации конструктора
137      копирования и операции присваивания. Также потребуется правильная реализация
138      конструктора перемещения (пока не проходили) и операции перемещающего присваивания
139      (также пока не проходили). В целях экономии места код этих операций в данном
140      примере не приводится. Вместо этого, мы явно указываем компилятору посредством
141      = delete (C++11), чтобы он не создавал соответствующие методы по-умолчанию. Попытка
142      воспользоваться "удалёнными" методами приведёт к ошибке компиляции. Таким образом,
143      код класса ChessBoard останется корректным. */
144   ChessBoard(const ChessBoard&) = delete;
145   ChessBoard(ChessBoard&&) = delete;
146   ChessBoard& operator=(const ChessBoard&) = delete;
147   ChessBoard& operator=(ChessBoard&&) = delete;
148
149   /* Метод для чтения элементов массива (фигур на доске) по позиции. */
150   const ChessPiece* GetCell(const Cell&) const;
151   /* Метод для записи элементов массива (фигур на доске) по позиции. Метод возвращает
152      ссылку на указатель для того, чтобы можно было модифицировать этот указатель. */
153   ChessPiece*& GetCell(const Cell&);
154
155   /* Добавляем фигуру. Здесь передаётся указатель на класс для того, чтобы обеспечить
156      полиморфное поведение. Ссылку передать не получится, поскольку массив не может
157      хранить ссылки. Данный класс сам освобождает все указатели при необходимости. */
158   void Add(ChessPiece* piece);
159
160   // Метод для передвижения фигуры
161   void Move(const Cell& src, const Cell& dest);
```

```
162
163   // Вывод всех фигур в переданный поток out.
164   void Print(std::ostream& out) const;
165
166   // Нам понадобится деструктор для освобождения указателей, хранящихся в массиве.
167   ~ChessBoard();
168
169  private:
170   /* Двумерный массив указателей на фигуры размером с шахматную доску. Указатели
171      нужны для того, чтобы фигуры, хранящиеся в массиве вели себя полиморфно. Если
172      в клетке отсутствует фигура, то соответствующий указатель будет равен nullptr. */
173   ChessPiece* board[boardSize][boardSize];
174 };
175
176 #endif // CHESS_HPP
```

4.3 Файл chess.cpp

```
1  #include <iostream>
2  #include <sstream>    // Нужно для работы класса std::ostringstream.
3  #include <stdexcept>  // Здесь определены стандартные классы исключений.
4  #include <cmath>
5  #include "cell.hpp"
6  #include "chess.hpp"
7
8  /************       Методы класса ChessPiece          ************************/
9
10 /* Виртуальный метод базового класса проверяет, можно ли съесть фигуру.
11    У пешки метод отличается, именно поэтому в базовом классе он виртуальный. */
12 bool ChessPiece::CanEat(const ChessPiece& other) const {
13   return (team != other.team) && IsMoveCorrect(other.pos);
14 }
15
16 // Этот метод перемещает фигуру.
17 void ChessPiece::Move(const Cell& newPos) {
18   /* Вызов абстрактного метода, проверяющего корректность хода. Этот метод для каждого
19      типа фигуры свой. */
20   if (IsMoveCorrect(newPos))
21     pos = newPos;
22   else {
23     std::ostringstream reasonStream;
24     // Кидаем исключение, если не получилось сделать ход.
25     reasonStream << "Can't move " << type << " from " << pos << " to "
26                  << newPos << "!";
27     throw std::runtime_error(reasonStream.str());
28   }
29 }
30
31 // Этот метод для того, чтобы съесть фигуру.
```

```
32 void ChessPiece::Eat(const ChessPiece& other) {
33   // Вызываем виртуальный метод для проверки, можно ли съесть.
34   if (!CanEat(other)) {
35     // Кидаем исключение, если не получилось.
36     std::ostringstream reasonStream;
37     reasonStream << *this << "' can't eat '" << other << "!";
38     throw std::runtime_error(reasonStream.str());
39   }
40 
41   // Перемещаем фигуру на новую позицию.
42   pos = other.pos;
43 }
44 
45 /* Этот оператор служит для вывода фигуры в поток вывода. Подробный комментарий
46    см. в cell.hpp для аналогичного оператора для класса Cell. */
47 std::ostream& operator<<(std::ostream& out, const ChessPiece& piece) {
48   out << "Piece ('" << piece.Type() << "', '" << piece.Team() << "', "
49       << piece.Pos() << ")";
50   return out;
51 }
52 
53 /******        Методы класса Pawn                ****************/
54 
55 // Определяем метод, который был абстрактным в базовом классе.
56 bool Pawn::IsMoveCorrect(const Cell& newPos) const {
57   Delta delta = newPos - pos; // Сохраняем разницу нового и старого положения.
58 
59   return (delta.X() == 0 &&   // Проверка корректности хода.
60       ((delta.Y() == 1 && team == "white") ||
61        (delta.Y() == -1 && team == "black")));
62 }
63 
64 /* Пешка ест по-другому. Поэтому нам нужно переопределить виртуальный метод
65    CanEat() базового класса. */
66 bool Pawn::CanEat(const ChessPiece& other) const {
67   if (team == other.Team())
68     return false;
69 
70   Delta delta = other.Pos() - pos;
71 
72   return (std::abs(delta.X()) == 1 &&
73       ((delta.Y() == 1 && team == "white") ||
74        (delta.Y() == -1 && team == "black")));
75 }
76 
77 /******        Методы класса Knight              ****************/
78 
79 bool Knight::IsMoveCorrect(const Cell& newPos) const {
80   Delta delta = newPos - pos;
81 
82   return ((std::abs(delta.X()) == 1 && std::abs(delta.Y()) == 2) &&
```

```
83        (std::abs(delta.X()) == 2 && std::abs(delta.Y()) == 1));
84  }
85
86  /******        Методы класса Bishop                ***************/
87
88  bool Bishop::IsMoveCorrect(const Cell& newPos) const {
89    Delta delta = newPos - pos;
90
91    return (std::abs(delta.X()) == std::abs(delta.Y()));
92  }
93
94  /******        Методы класса Castle                ***************/
95
96  bool Castle::IsMoveCorrect(const Cell& newPos) const {
97    Delta delta = newPos - pos;
98
99    return (std::abs(delta.X()) == 0 || std::abs(delta.Y()) == 0);
100 }
101
102 /******        Методы класса Queen                ***************/
103
104 bool Queen::IsMoveCorrect(const Cell& newPos) const {
105   Delta delta = newPos - pos;
106
107   return (std::abs(delta.X()) == 0 || std::abs(delta.Y()) == 0 ||
108           (std::abs(delta.X()) == std::abs(delta.Y())));
109 }
110
111 /******        Методы класса King                ***************/
112
113 bool King::IsMoveCorrect(const Cell& newPos) const {
114   Delta delta = newPos - pos;
115
116   return (std::abs(delta.X()) == 1 || std::abs(delta.Y()) == 1);
117 }
118
119 /******        Методы класса ChessBoard                ***************/
120
121 ChessBoard::ChessBoard() {
122   /* Зануляем все клетки доски. */
123   for (int row = 0; row < boardSize; row++) {
124     for (int col = 0; col < boardSize; col++)
125       board[row][col] = nullptr;
126   }
127 }
128
129 /* Метод для чтения элементов массива (фигур на доске) по позиции. */
130 const ChessPiece* ChessBoard::GetCell(const Cell& pos) const {
131   return board[pos.Y()][pos.X()];
132 }
133
```

```
134 /* Метод для записи элементов массива (фигур на доске) по позиции. Метод возвращает
135    ссылку на указатель для того, чтобы можно было модифицировать этот указатель. */
136 ChessPiece*& ChessBoard::GetCell(const Cell& pos) {
137   return board[pos.Y()][pos.X()];
138 }
139
140
141 // Метод для добавления фигуры.
142 void ChessBoard::Add(ChessPiece* piece) {
143   // Проверяем, занята ли клетка.
144   if (GetCell(piece->Pos()) != nullptr) {
145     std::ostringstream reasonStream;
146     reasonStream << "Can't add " << *piece << " since the position is busy!";
147
148     // Удаляем фигуру поскольку она дальше не используется.
149     delete piece;
150
151     // Кидаем исключение, если клетка занята.
152     throw std::runtime_error(reasonStream.str());
153   }
154
155   // Добавляем фигуру piece на позицию piece->Pos().
156   GetCell(piece->Pos()) = piece;
157 }
158
159 // Метод для перемещения фигуры из клетки src в клетку dest.
160 void ChessBoard::Move(const Cell& src, const Cell& dest) {
161   // Проверяем, есть ли в клетке src фигура.
162   if (GetCell(src) == nullptr) {
163     // Кидаем исключение, если клетка пустая.
164     std::ostringstream reasonStream;
165     reasonStream << "Can't move from " << src << " to " << dest << "!"
166                  << " Cell " << src << " is empty!";
167     throw std::runtime_error(reasonStream.str());
168   }
169
170   if (GetCell(dest) == nullptr) {
171     // Перемещаем фигуру, класс фигуры выполняет все проверки.
172     GetCell(src)->Move(dest); // Выбросит исключение в случае ошибки.
173
174     // Теперь эта фигура будет храниться на позиции dest.
175     GetCell(dest) = GetCell(src);
176
177     // Удаляем фигуру с доски с позиции src.
178     GetCell(src) = nullptr;
179   }
180   else {
181     // Фигура, находящаяся в клетке src ест фигуру в клетке dest.
182     GetCell(src)->Eat(*GetCell(dest)); // Выбросит исключение в случае ошибки.
183
184     // Удаляем объект, описывающий съеденную фигуру (освобождаем память).
```

```
185     delete GetCell(dest);
186
187     // Теперь оставшаяся фигура будет храниться на позиции dest.
188     GetCell(dest) = GetCell(src);
189
190     // Удаляем фигуру с доски с позиции src.
191     GetCell(src) = nullptr;
192   }
193 }
194
195 void ChessBoard::Print(std::ostream& out) const {
196   // Проходим по всей доске и выводим в поток фигуры, которые на ней стоят.
197   for (int row = 0; row < boardSize; row++) {
198     for (int col = 0; col < boardSize; col++) {
199       if (board[row][col] != nullptr)
200         out << *board[row][col] << std::endl;
201     }
202   }
203 }
204
205 ChessBoard::~ChessBoard() {
206   // Проходим по всей доске и освобождаем память фигур, которые стоят на доске.
207   for (int row = 0; row < boardSize; row++) {
208     for (int col = 0; col < boardSize; col++) {
209       if (board[row][col] != nullptr)
210         delete board[row][col];
211     }
212   }
213 }
214
215 int main()
216 {
217   ChessBoard board; // Наша шахматная доска.
218   std::string command; // Команда.
219
220   std::cout << "Enter command -> "; // Приглашение для ввода.
221
222   /* Вспомним, что операция >> для потока вывода возвращает ссылку на этот поток.
223      А для самого потока переопределена неявная операция приведения типа к типу данных
224        1) void* (до C++11)
225        2) bool (начиная с C++11)
226      Если результат операции приведения равен NULL (или false), то произошла ошибка
227      чтения. Таким образом, следующий цикл будет считывать "пока считывается". */
228   while (std::cin >> command) { // Считываем команды покуда считываются.
229     if (command == "exit") // Получили команду для выхода.
230       break;
231     else if (command == "help") { // Выводим справку.
232       std::cout << "help - print help information" << std::endl;
233       std::cout << "add <type> <team> <pos_x> <pos_y> - add chess piece"
234                 << std::endl;
235       std::cout << "    type = pawn|knight|bishop|rock|queen|king" << std::endl;
```

```
236       std::cout << "    team = white|black" << std::endl;
237       std::cout << "    x, y belong to [0, " << boardSize - 1 << "]" << std::endl;
238       std::cout << "move <src x> <src y> <dest x> <dest y> - move chess piece"
239                 << std::endl;
240       std::cout << "print - print chess board" << std::endl;
241       std::cout << "ext - exit" << std::endl;
242     }
243     else if (command == "add") { // Получили команду для добавления фигуры.
244       std::string type; // Тип фигуры.
245       std::string team; // Белые/чёрные.
246       int x, y;
247
248       if (!(std::cin >> type >> team >> x >> y)) {
249         std::cout << "Incorrect argument!" << std::endl; // Если не считалось...
250         continue;
251       }
252
253       if (team != "white" && team != "black") {
254         std::cout << "Incorrect team!" << std::endl; // Неправильная команда.
255         continue;
256       }
257
258       try { // Метод Add() класса ChessBoard может выбросить исключение.
259         if (type == "pawn") {
260           /* Мы создаём указатель на объект класса Pawn для того, чтобы объект
261              существовал после того, как программа выйдет из этого блока. То есть
262              в функции main() он будет потерян, однако класс ChessBoard будет о нём
263              помнить. Использование оператора new потребует вызова оператора delete для
264              освобождения памяти. Это будет сделано в классе ChessBoard. Использование
265              указателя обязательно для обеспечения полиморфного поведения, поскольку
266              функция Add() класса ChessBoard принимает базовый класс ChessPiece. Таким
267              образом, простое копирование не подойдёт, иначе не будет полиморфного
268              поведения. Передача ссылки не сработает из-за выхода из области видимости.
269              Кроме того, фигуры в классе доски хранятся в массиве, а массив не может
270              хранить ссылки. */
271           board.Add(new Pawn(team, Cell(x, y)));
272         }
273         else if (type == "knight")
274           board.Add(new Knight(team, Cell(x, y)));
275         else if (type == "bishop")
276           board.Add(new Bishop(team, Cell(x, y)));
277         else if (type == "castle")
278           board.Add(new Castle(team, Cell(x, y)));
279         else if (type == "queen")
280           board.Add(new Queen(team, Cell(x, y)));
281         else if (type == "king")
282           board.Add(new King(team, Cell(x, y)));
283         else
284           std::cout << "Incorrect piece type: '" << type << "'" << std::endl;
285         }
286       catch(std::runtime_error& e) { // Обрабатываем исключение.
```

```
287         /* Метод what(), возвращающий const char*, позволяет узнать причину
288            исключения, то есть ту строку, которая указывалась в конструкторе
289            класса std::runtime_error при выбросе исключения. */
290         std::cout << "Caught an exception. Reason: '" << e.what() << "'."
291                   << std::endl;
292       }
293     }
294     else if (command == "move") { // Получили команду на перемещение фигуры.
295       int x, y, nx, ny;
296 
297       if (!(std::cin >> x >> y >> nx >> ny)) {
298         std::cout << "Incorrect argument!" << std::endl; // Не считалось...
299         continue;
300       }
301 
302       try {
303         board.Move(Cell(x, y), Cell(nx, ny)); // Может выбросить исключение.
304       }
305       catch(std::runtime_error& e) {
306         std::cout << "Caught an exception. Reason: '" << e.what() << "'."
307                   << std::endl;
308       }
309     }
310     else if (command == "print") {
311       board.Print(std::cout); // Печатаем в поток std::cout информацию о всех фигурах.
312     }
313     else {
314       std::cout << "Incorrect command. Type 'help' for help." << std::endl;
315     }
316 
317     std::cout << "Enter command -> ";
318   }
319   return 0;
320 }
```