Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Чувашский государственный университет им. И.Н.Ульянова».

Кафедра вычислительной техники.

Предмет: Объектно-ориентированное программирование

Курсовая работа

по дисциплине «Объектно-ориентированное программирование»

на тему:

**Игровая программа «Ханойская башня»**

Вариант 5.

Выполнил: Васильев Егор Юрьевич

студент группы ИВТ-41-20

Проверил: Обломов И.А.

Оценка:

Дата:

Чебоксары, 2021.

# Введение

**Цель работы:** закрепление теоретических знаний, полученных по данному курсу, и приобретение практических навыков формализации поставленной задачи, создания и использования эффективных структур данных и алгоритмов.

**Объектно-ориентированное программирование** (ООП) это совокупность понятий (класс, объект, инкапсуляция, полиморфизм, наследование), приемов их использования при проектировании программ, а Си++ - инструмент этой технологии

Программирование "от класса к классу" включает в себя ряд новых понятий. Прежде всего, это - **инкапсуляция данных**, то есть логическое связывание данных с конкретной операцией. Инкапсуляция данных означает, что данные являются не глобальными - доступными всей программе, а локальными - доступными только малой ее части. Инкапсуляция автоматически подразумевает защиту данных. Для этого в структуре class используется спецификатор раздела private, содержащий данные и методы, доступные только для самого класса. Если данные и методы содержатся в разделе public, они доступны извне класса. Раздел protected содержит данные и методы, доступные из класса и любого его производного класса. Наличие последних позволяет говорить об иерархии классов, где есть классы - родители - шаблоны для создания классов - потомков. Объекты, полученные из описания класса, называют экземплярами этого класса.

Вторым по значимости понятием является **наследование**. Новый, или производный класс может быть определен на основе уже имеющегося, или базового. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки. Если объект наследует свои свойства от одного родителя, то говорят об одиночном наследовании. Если же объект наследует атрибуты от нескольких базовых классов, то говорят о множественном наследовании.

Третьим по значимости понятием является **полиморфизм**. Он основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Принципиально важно, что такой объект становится "самодостаточным". Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о его типе, он всегда может корректно вызвать свойственные ему методы. Полиморфной называется функция, независимо определенная в каждом из группы производных классов и имеющая в них общее имя. Полиморфная функция обладает тем свойством, что при отсутствии полной информации о том, объект какого из производных классов в данный момент обрабатывается, она тем не менее корректно вызывается в том виде, к каком она была определена для данного конкретного класса. Практический смысл полиморфизма заключается в том, что он позволяет посылать общее сообщение о сборе данных любому классу, причем и родительский класс, и классы-потомки ответят на сообщение соответствующим образом, поскольку производные классы содержат дополнительную информацию.

# Постановка задачи

**Легенда**

В одном из буддийских монастырей монахи уже тысячу лет занимаются перекладыванием колец. Они располагаются тремя пирамидами, на которых надеты кольца разных размеров. В начальном состоянии 64 кольца были надеты на первую пирамиду и упорядочены по размеру. Монахи должны переложить все кольца с первой пирамиды на вторую, выполняя единственное условие – кольцо нельзя положить на кольцо меньшего размера. При перекладывании можно использовать все три пирамиды. Монахи перекладывают одно кольцо за одну секунду. Как только они закончат свою работу, наступит конец света. Количество перекладываний в зависимости от количества колец вычисляется по формуле 2n-1. Для 64 колец это 18 446 744 073 709 551 615 перекладываний, и, если учесть скорость одно перекладывание в секунду, получится около 584 942 417 355 лет, то есть апокалипсис наступит нескоро.

На самом деле эту известную игру придумал французский математик Эдуард Люка, в 1883 году её продавали как забавную игрушку. Первоначально она называлась «Профессор Клаус из Колледжа Ли-Су-Стьян», но вскоре обнаружилось, что таинственный профессор из несуществующего колледжа — не более чем анаграмма фамилии изобретателя игры — профессора Люка из колледжа Сен-Луи.

Даны три стержня из N дисков разного диаметра, которые надеты на стержень в порядке убывания диаметра. Надо переместить N дисков за наименьшее число шагов на стержень (3), так чтобы они остались в таком же порядке. При этом требуется соблюдать правила:

* На каждом шаге ровно один диск перемещается с одного диска на другой; ·
* Диск большего диаметра нельзя помещать на диск меньшего диаметра; ·
* Стержень (2) можно использовать как промежуточный.

Количество перекладываний в зависимости от количества колец вычисляется по формуле **2n-1.**

Планируется реализовать 2 класса. Первый – алгоритм, который сам найдет лучшее решение и выведет его в консоль. Второй класс – игрок. При помощи второго класса можно будет самому попытаться решить задачу о Ханойской башне.

# 

# Выбор метода

Многим из тех людей, которые играют в эту игру, практически никогда не удается обнаружить весьма простую **стратегию**, позволяющую успешно играть в Ханойские башни с тремя штырями и N дисками.

\* Граничное условие выполняется в случае, когда на исходном (левом) штыре нет дисков.

\* Переместить N-1 дисков с исходного штыря на запасной (правый) штырь, используя итоговый штырь как запасной; отметим, что это перемещение осуществляется рекурсивно.

\* Переместить один диск с исходного штыря на итоговый штырь. В этом месте наша программа будет выдавать сообщение об этом перемещении.

\* Наконец, переместить N-1 дисков с запасного на итоговый, используя исходный штырь в качестве запасного.

Начнем с самого маленького кольца и переложим его на любую отметку. В дальнейшем это кольцо нужно перемещать в том же направлении, что и при первом перекладывании. Затем произведем единственно возможное перемещение оставшихся колец, после чего снова переложим самое маленькое кольцо и т. д. (Интересно заметить, что, перенумеровав «кольца» по порядку, мы добьёмся неожиданного эффекта: четные кольца будут перемещаться из одной вершины треугольника в другую в одном направлении, а нечётные - в противоположном направлении.)

Задача о ханойских башнях - это классический пример применения **рекурсии** для описания эффективного алгоритма.

В программировании **рекурсия** - вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция вызывает функцию , а функция - функцию . Количество вложенных вызовов функции или процедуры называется глубиной рекурсии.

Преимущество **рекурсивного** определения объекта заключается в том, что такое конечное определение теоретически способно описывать бесконечно большое число объектов. С помощью рекурсивной программы же возможно описать бесконечное вычисление, причём без явных повторений частей программы.

Займемся теперь непосредственно методом, реализующим алгоритм и перекладывающим кольца в соответствии с правилами игры. Написать **нерекурсивный** вариант ханойских башен совсем не просто. Можно, конечно, написать цикл, завершающийся по достижению требуемой конфигурации, на каждом шаге которого выполняется очередной ход. Но даже первый ход не тривиален. Поскольку фиксирован столб, где должны быть собраны кольца, то неясно, куда нужно переложить первое кольцо - на второй или третий столб? Сложность **нерекурсинвого** алгоритма является его главным минусом, поэтому использоваться **не будет.**

**Рекурсивный** вариант решения задачи прозрачен, хотя и напоминает некоторый род фокуса, что характерно для рекурсивного стиля мышления. Базис рекурсии прост. Для перекладывания одного кольца задумываться о решении не нужно - оно делается в один ход. Если есть базисное решение, то оставшаяся часть также очевидна. Нужно применить рекурсивно алгоритм, переложив n-1 кольцо с первой пирамиды на третью пирамиду. Затем сделать очевидный ход, переложив последнее самое большое кольцо с первой пирамиды на вторую. Затем снова применить рекурсию, переложив n-1 кольцо с третьей пирамиды на вторую пирамиду. Задача решена.

**Выбран рекурсивный метод**, т. к. он наиболее понятен и не требует долгой и сложной реализации.

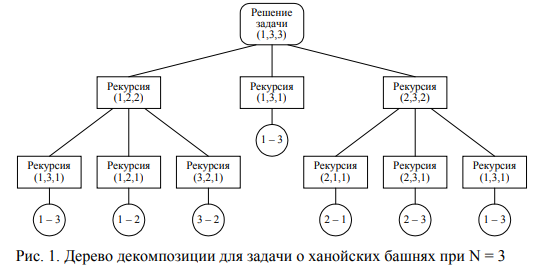
Для реализации данного метода была выбрана среда разработки Visual Studio. Каждый ход будет выводиться в консоль. Для этого определим класс HumanPlayer, позволяющий решать задачу человеку, выбирая в консоли стержни для передвижения колец. Создадим новый класс Bot, который будет наследоваться от класса HumanPlayer, и дадим ему возможность решить задачу с помощью рекурсии.

# Теоретический раздел

В программировании **рекурсия** - вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция вызывает функцию, а функция - функцию. Количество вложенных вызовов функции или процедуры называется **глубиной рекурсии**.

Пронумеровав «кольца» по порядку, четные кольца будут перемещаться из одной вершины треугольника в другую в одном направлении, а нечётные - в противоположном направлении.

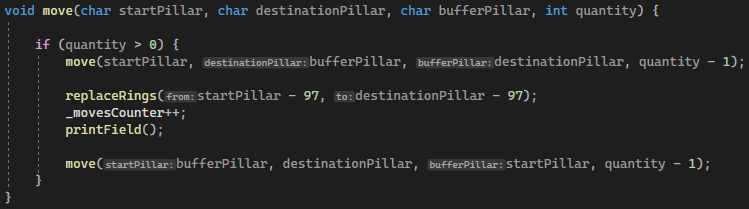
Займемся теперь непосредственно методом, **реализующим алгоритм** и перекладывающим кольца в соответствии с правилами игры.



На нечетном шаге мы всегда перемещаем верхний круг: с 1-ой башни на 3-ю, со 2-ой на 1-ю или с 3-ей на вторую. На четном шаге мы находим номер кольца, которое нужно передвинуть и перемещаем его на единственную доступную башню.

В первой строке вызова move() (п.1) делаются ходы № 1, 4, 7… Ni-1 + 3

Во второй (п.2) - 2 хода после п.1 (№ 2, 3, 5, 6, 8, 9…)

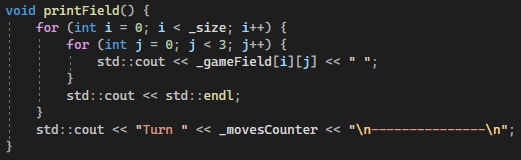


**2**

**1**

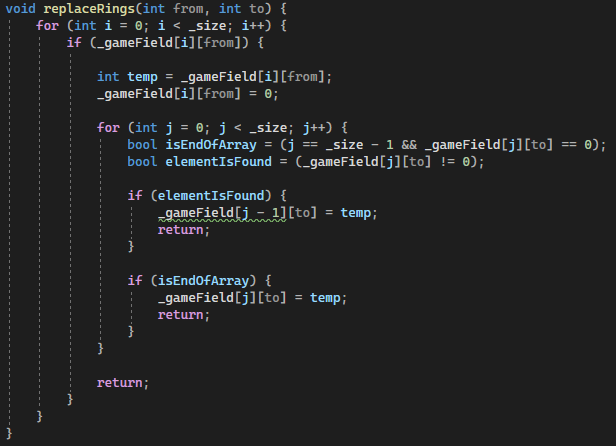
При работе данного метода проверка на ошибки не требуется, т.к данные всегда будут верны. При игре человека потребуется проверка на ошибки.

Для вывода используется метод printField(), который выводит массив с информацией о текущей ситуации на «игровом поле».

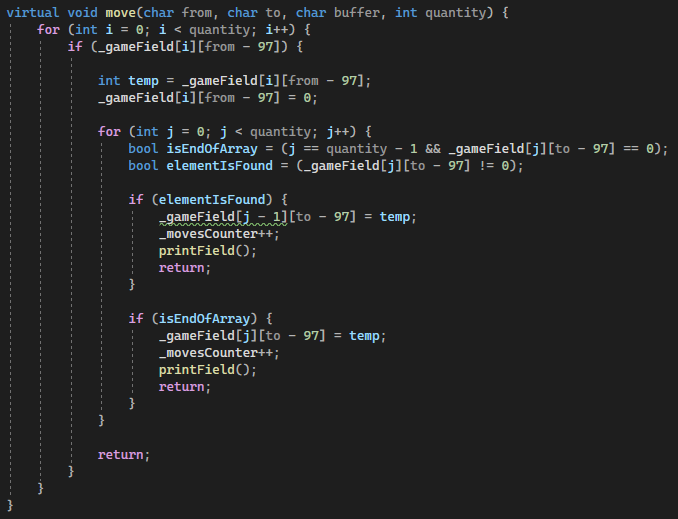


Метод replaceRings() используется для перестановки дисков в массиве. Т.к метод принимает только тип int, при передаче параметров типа char, вычитаем 97 из переменных, чтобы получить номер столбцов.

Метод проходится по массиву, где хранится информация о игровом «поле», находит верхний элемент в столбце, откуда нужно забрать диск. Запоминает этот элемент, обнуляет значение, где он раньше находился. Дальше проходится сверху-вниз по столбцу, куда нужно переместить диск. Если там уже есть какой-то элемент, помещает сверху новый, если нет, то элемент помещается в конец (основание башни).

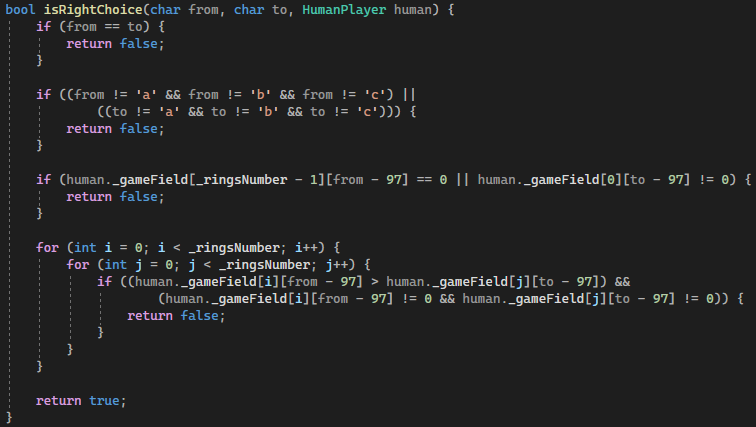


Метод move() класса HumanPlayer, от которого унаследован класс Bot, работает по такому же принципу что и ReplaceRings(). По информации от пользователя определяются стержни, для которых нужно сделать перемещение дисков.



Правильность ввода данных пользователем проверяется методом isRightChoice().

Если названия стержней совпадают, введены неправильные буквы, при вводе значений будут нарушены правила игры, то возвращается значение false, пользователь вводит новые значения.



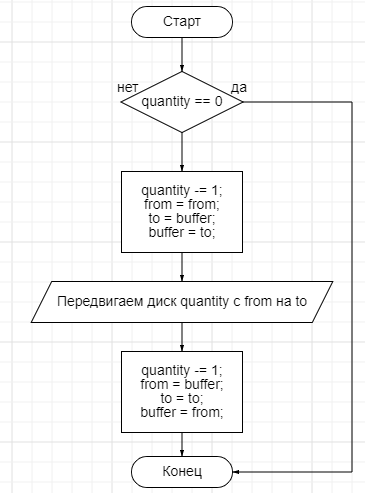
# Блок-схема алгоритма

quantity - кол-во дисков

from - пирамида с дисками

to - пирамида, куда будем перемещать диски

buffer - свободная пирамида для использования как буфер



# Выводы по работе

В данной курсовой работе спроектирована и разработана программа на языке C++, которая на основе запрошенных у пользователя входных данных моделирует «Ханойские башни» или позволяет разложить их вручную. Программа обладает достаточно устойчивой защитой от ввода некорректных данных, удобным интерфейсом для работы пользователей.

Итогом курсовой работы можно считать закрепление знаний в области создания алгоритмов навыков программирования на языке высокого уровня C++, полученных в ходе обучения по курсу «Объектно-ориентированное программирование».

Использование объектно-ориентированного программирования удобно, а в ряде случаев практически необходимо в создании больших программных продуктов.

# 

# Список использованной литературы

<http://ru.wikipedia.org/wiki/Ханойская_башня>

<http://ru.wikipedia.org/wiki/Рекурсия>

Оглавление

[Введение 2](#_Toc104932886)

[Постановка задачи 3](#_Toc104932887)

[Выбор метода 4](#_Toc104932888)

[Теоретический раздел 6](#_Toc104932889)

[Блок-схема алгоритма 10](#_Toc104932890)

[Выводы по работе 11](#_Toc104932891)

[Список использованной литературы 12](#_Toc104932892)

[Приложение 14](#_Toc104932893)

# Приложение

**Текст программы.**

**HanoiTowers.cpp**

#include <iostream>

#include "Hanoi.h"

#include "Bot.h"

#include "HumanPlayer.h"

using namespace std;

int main() {

char startPillar = 'a',

destinationPillar = 'c',

bufferPillar = 'b';

int numberOfRings = 3;

Hanoi game(startPillar, destinationPillar, bufferPillar, numberOfRings);

int choice;

cout << "Who is playing?\n(1 - Bot, 2 - Human):\n";

cin >> choice;

cout << endl;

switch (choice) {

case 1: {

Bot bot(game.gameField, numberOfRings);

cout << "Bot is playing:\n\n";

game.start(bot);

break;

}

case 2: {

HumanPlayer human(game.gameField, numberOfRings);

cout << "Human is playing:\n";

game.start(human);

break;

}

default:

break;

}

}

**HumanPlayer.h**

#pragma once

#include "Hanoi.h"

class HumanPlayer {

protected:

int \_size;

int \_movesCounter;

public:

int \_gameField[20][3];

HumanPlayer(int gameField[][3], int size) : \_size(size), \_movesCounter(0) {

for (int i = 0; i < \_size; i++) {

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

\_gameField[i][j] = gameField[i][j];

}

}

}

virtual void move(char from, char to, char buffer, int quantity) {

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

if (\_gameField[i][from - 97]) {

int temp = \_gameField[i][from - 97];

\_gameField[i][from - 97] = 0;

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

bool isEndOfArray = (j == quantity - 1 && \_gameField[j][to - 97] == 0);

bool elementIsFound = (\_gameField[j][to - 97] != 0);

if (elementIsFound) {

\_gameField[j - 1][to - 97] = temp;

\_movesCounter++;

printField();

return;

}

if (isEndOfArray) {

\_gameField[j][to - 97] = temp;

\_movesCounter++;

printField();

return;

}

}

return;

}

}

}

void printField() {

for (int i = 0; i < \_size; i++) {

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

std::cout << \_gameField[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << "Turn " << \_movesCounter << "\n---------------\n";

}

};

**Bot.h**

#pragma once

#include <iostream>

#include "HumanPlayer.h"

class Bot : public HumanPlayer {

public:

Bot(int gameField[][3], int size) : HumanPlayer(gameField, size) {

}

void move(char startPillar, char destinationPillar, char bufferPillar, int quantity) {

if (quantity > 0) {

move(startPillar, bufferPillar, destinationPillar, quantity - 1);

replaceRings(startPillar - 97, destinationPillar - 97);

\_movesCounter++;

printField();

move(bufferPillar, destinationPillar, startPillar, quantity - 1);

}

}

private:

void replaceRings(int from, int to) {

for (int i = 0; i < \_size; i++) {

if (\_gameField[i][from]) {

int temp = \_gameField[i][from];

\_gameField[i][from] = 0;

for (int j = 0; j < \_size; j++) {

bool isEndOfArray = (j == \_size - 1 && \_gameField[j][to] == 0);

bool elementIsFound = (\_gameField[j][to] != 0);

if (elementIsFound) {

\_gameField[j - 1][to] = temp;

return;

}

if (isEndOfArray) {

\_gameField[j][to] = temp;

return;

}

}

return;

}

}

}

};

**Hanoi.h**

#pragma once

#include <iostream>

#include "HumanPlayer.h"

#include "Bot.h"

class Hanoi {

private:

const char \_startPillar,

\_destinationPillar,

\_bufferPillar;

const int \_ringsNumber;

const int \_pillarsCount;

int \_movesNumber;

public:

int gameField[20][3];

Hanoi(char startPillar, char destPillar, char buffPillar, int ringsNum) :

\_startPillar(startPillar), \_destinationPillar(destPillar), \_bufferPillar(buffPillar), \_ringsNumber(ringsNum), \_pillarsCount(3) {

\_movesNumber = 0;

nullArray();

fillPillar();

}

void fillPillar() {

for (int i = 0; i < \_ringsNumber; i++) {

gameField[i][\_startPillar - 97] = i + 1;

}

}

void printField() {

for (int i = 0; i < \_ringsNumber; i++) {

for (int j = 0; j < \_pillarsCount; j++) {

std::cout << gameField[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << "Turn " << \_movesNumber << "\n---------------\n";

}

void nullArray() {

for (int i = 0; i < \_ringsNumber; i++) {

for (int j = 0; j < \_pillarsCount; j++) {

gameField[i][j] = 0;

}

}

}

void start(Bot bot) {

nullArray();

fillPillar();

printField();

int quantity = \_ringsNumber;

bot.move(\_startPillar, \_destinationPillar, \_bufferPillar, quantity);

printWinMessage();

}

void start(HumanPlayer human) {

nullArray();

fillPillar();

printField();

char from, to;

while (gameContinues(human)) {

do {

std::cout << "Choose start pillar and destination pillar (a - c):\n";

std::cin >> from >> to;

std::cout << std::endl;

} while (!isRightChoice(from, to, human));

human.move(from, to, \_bufferPillar, \_ringsNumber);

}

printWinMessage();

}

bool isRightChoice(char from, char to, HumanPlayer human) {

if (from == to) {

return false;

}

if ((from != 'a' && from != 'b' && from != 'c') ||

((to != 'a' && to != 'b' && to != 'c'))) {

return false;

}

if (human.\_gameField[\_ringsNumber - 1][from - 97] == 0 || human.\_gameField[0][to - 97] != 0) { // if circles on start pillar are exists and enough space in destination pillar

return false;

}

for (int i = 0; i < \_ringsNumber; i++) {

for (int j = 0; j < \_ringsNumber; j++) {

if ((human.\_gameField[i][from - 97] > human.\_gameField[j][to - 97]) &&

(human.\_gameField[i][from - 97] != 0 && human.\_gameField[j][to - 97] != 0)) {

return false;

}

}

}

return true;

}

bool gameContinues(HumanPlayer human) {

if (human.\_gameField[0][\_destinationPillar - 97] != 0) {

return false;

}

return true;

}

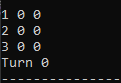
void printWinMessage() {

std::cout << "You are won!\n";

}

};

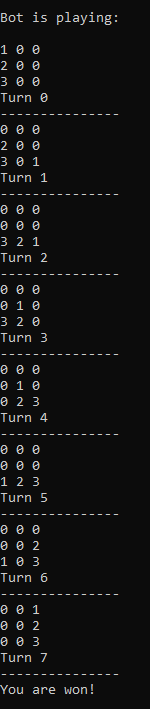
**Результаты выполнения программы и тесты**

1, 2, 3 – диски, расположенные в порядке убывания диаметра от основания.

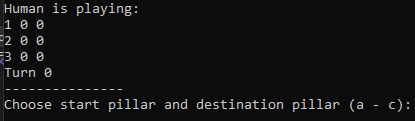
На старте программа спрашивает, кто будет играть: бот или человек.

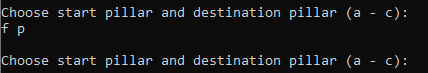


Демонстрация работы рекурсивного алгоритма, который сработает, если будет играть бот.

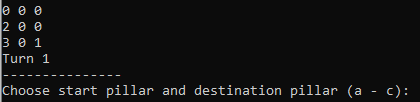


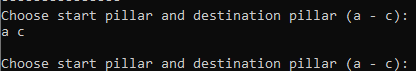
Демонстрация работы алгоритма, который сработает, если будет играть человек.

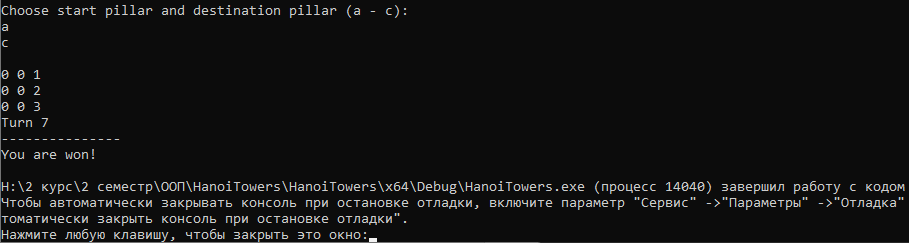
Программа просит сделать следующий ход. Для примера введем названия несуществующих стержней, например, f и p.



Как итог программа просит ввести данные повторно. На этот раз сделаем все правильно.

Как видим, кольцо переместилось по нашей команде. Т.к игра не закончилась, требуется повторный ввод. Попробуем поставить кольцо большего диаметра (2) на кольцо с диаметром 1. Это запрещено правилами.

Снова сработала защита, нужно ввести корректные данные. Пропустим до момента, когда нужно будет сделать последний ход.



Как только все кольца оказались в нужном порядке на последнем стержне, программа завершила работу, вывелось соответствующее сообщение и количество затраченных ходов.