|  |
| --- |
| **МИНОБРНАУКИ РОССИИ**  **САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  **ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  **«ЛЭТИ» ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)**  **Кафедра РЭС** |
| **ОТЧЕТ по лабораторной работе №5 по дисциплине «Информатика» Тема: использование классов для создания телефонной книги.** |
| |  |  | | --- | --- | | Студент гр. 1183 | Чаминов Д. А. | | Преподаватель | Ситников И. Ю. | |

# Содержание

[Содержание 2](#_Toc86833440)

[Спецификация задания 3](#_Toc86833441)

[Описание алгоритма 4](#_Toc86833442)

[Блок-схема алгоритма 5](#_Toc86833443)

[Выбор и обоснование типов переменных 8](#_Toc86833444)

[Вводимые и выводимые параметры и их типы 9](#_Toc86833445)

[Структура проекта 10](#_Toc86833446)

[Текст программы 11](#_Toc86833447)

[Копии экрана при работе программы 12](#_Toc86833448)

[Контрольный пример, сравнение с эталоном 13](#_Toc86833449)

[Результаты 14](#_Toc86833450)

# Спецификация задания

Разрабатываемая программа должна отвечать следующим требованиям:

* Быть разработана на языке C++ в среде MS Visual Studio.
* Работать в операционных системах Windows 7 и выше.
* Принимать данные контактов, сортировать их и выводить в виде таблицы.
* Содержать хотя бы 2 класса.

# Описание алгоритма

Для хранения данных создается объект класса Phonebook, который содержит в себе всего один объект – контейнер std::list<PhoneRecord> - двусвязный список объектов класса PhoneRecord. Для экземпляров этого класса определена функция поиска места в двусвязном списке для сортировки записей, возвращающая итератор – объект, позволяющий осуществлять доступ к содержимому контейнера std::list<PhoneRecord>, std::list<PhoneRecord>::iterator findPlaceForRecord( const PhoneRecord& r ), функции добавления записи (с созданием по имени и номеру и по самой записи void addRecord), функция позволяющая считать 1 запись с потока ввода с приглашением по потоку вывода и добавить ее в список void inputRecord, функции позволяющие считать произвольное количество записей (до получения записи с номером телефона равным заранее заданному или заданное наперед количество записей) void input, функция распечатки телефонной книги void print.

Двусвязный список – это упорядоченная структура данных, каждый элемент которой обязательно имеет три поля – поле, в котором содержатся хранимые данные, указатель на предыдущий и на следующий элементы. Таким образом std::list является контейнером, обеспечивающим последовательный доступ, и, соответственно, не имеющем реализации случайного доступа.

PhoneRecord является классом, в который инкапсулирована безымянная структура, описывающая телефонный номер. Каждый объект этого класса содержит одну такую структуру number, структура состоит из двух полей: unsigned \_\_int64 number:50 (50 бит хватит для представления 15-значного положительного целого числа) и unsigned \_\_int64 plus:1 – индикатор того был ли в начале номера «+». Также объекты этого класса содержат в себе std::string – объект строки. Для полей записи определены «set’теры» и «get’теры». Также для объектов класса определена функция фильтрации строк std::string filter(const std::string& s), операторы сравнения (которые используют соответствующие операторы сравнение для строк std::string), функция ввода void input(std::istream&, std::ostream&, std::string, std::string), которая инициализирует поля объекта, используя запрос по потоку вывода и поток ввода, конструкторы по умолчанию и по имени и номеру, функция распечатки записи void print.

При старте программа создает объект телефонной книги, вызывает метод ввода для стандартных потоков ввода-вывода через консоль, пока пользователь не введет в качестве телефонного номера «0».

После завершения ввода, при котором, по мере ввода записей, они сортируются, вызывается метод распечатки телефонной книги с максимальной длиной имени контакта 50, через стандартный поток консольного вывода с разделителем ‘.’.

# Блок-схема алгоритма

Конец

system(“pause”);

pb.input(“0”);

pb.print(50,std::cout,’.’)

Phonebook pb;

Начало

Алгоритм Phonebook::input:

да

нет

нет

да

конец

начало

r->input(input, output, prompt\_name. prompt\_number)

PhoneRecord r;

Получить std::string terminator, std::istream & input, std::ostream & output, std::string prompt\_name, std::string prompt\_number

did\_user\_accept()

r->getName() == terminator

addRecord(r)

Алгоритм PhoneRecord::input:

начало

конец

setName(name);

setNumber(number);

std::string name, number;

Получить std::string terminator, std::istream & input, std::ostream & output, std::string prompt\_name, std::string prompt\_number

output: prompt\_name

input: name

output: prompt\_number

input: numbera

Алгоритм PhoneRecord::setName:

Конец

получить std::string name

Начало

this->name=name

Алгоритм PhoneRecord::setNumber:

нет

да

нет

да

начало

конец

number.plus = (s[0]==’+’);

number.num = std::stoll(s);

s.size()==0

number.num=0;

number.plus=0;

s=filter(s)

filterstr

получение std::string s, bool filterstr

Алгоритм PhoneRecord::filter:

нет

нет

да

да

конец

вернуть result

i = i+1;

is\_first=false;

result=result+s[i];

(s[i]>=’0’ И s[i]<=’9’) ИЛИ (is\_first И s[i]==’+’)

i<n

bool is\_first=true;

size\_t n = s.size();

std::string result=””;

size\_t i=0;

получение const std::string& s

начало

Алгоритм Phonebook::addRecord:

конец

phbook.emplace(i, r);

std::list<PhoneRecord>::iterator i = findPlaceForRecord(r);

получить PhoneRecord r

начало

Алгоритм Phonebook::findPlaceForRecord:

нет

нет

да

да

конец

вернуть i

i ++;

\*i>r

i==phbook.end()

std::list<PhoneRecord>::iterator i = phbook.begin();

начало

получить const PhonrRecord& r

Алгоритм Phonebook::print:

нет

да

конец

(\*i).print(name\_length, output, filler)

output: inflate\_string(std::to\_string(n+1),4)

size\_t n =0;

std::list<PhoneRecord>::iterator i = phbook.begin

i == phbook.end()

начало

Получить size\_t name\_length, std::ostream & output, char filler

Алгоритм PhoneRecord::print:

начало

std::string name = inflate\_string(nickname, name\_length, StrAlign::left, filler);

std::string n = inflate\_string(getNumber(), 20, StrAlign::right, filler);

output: name, n

конец

Получить size\_t name\_length, std::ostream & output, char filler

Алгоритм PhoneRecord::getNumber:

нет

да

конец

вернуть ‘+’ + std::to\_string(number.num)

вернуть std::to\_string(number.num)

number.plus

начало

Алгоритм inflate\_string:

конец

нет

нет

нет

нет

нет

нет

нет

да

да

да

да

да

да

да

s = inflate\_string(s, (w - n) / 2 + n, StrAlign::left, filler);

s = inflate\_string(s, (w - n) / 2 + (w - n) % 2 + s.size(), StrAlign::right, filler);

s.erase(0, (n - w) / 2);

s.erase(w, (n - w) / 2 + (n - w) % 2);

n>w

n>w

n<w

s.erase(w,n-w);

s=s+filler;

n++;

a==left

s.erase(0,n-w);

n>w

s=filler+s;

n++;

n<w

a==right

size\_t n = s.size();

получить std::string s, size\_t w, StrAlign:: a, char filler

начало

# Выбор и обоснование типов переменных

Используется пользовательские классы Phonebook и PhoneRecord.

Phonebook содержит в себе двусвязный список std::list<PhoneRecord>. Использование двусвязного списка обеспечивает быструю вставку в середину телефонной книги, что полезно для используемой сортировки вставками. Случайный доступ к элементам книги не необходим, поэтому двусвязный список подходит для этой задачи.

PhoneRecord содержит в себе строку std::string, которая представляет имя контакта, безымянную структуру, которая содержит два беззнаковых целых – телефонный номер (50 бит достаточно для представления 15-значного числа) и индикатор того, был ли в исходном номере ‘+’ (1 бит).

В качестве счетчиков используется тип «size\_t», являющийся аналогичным «unsigned long long int».

В качестве буферов для считывания с потока ввода и для вывода на поток вывода используются «std::string».

# Вводимые и выводимые параметры и их типы

В программу подаются строки «std::string», содержащие имена и номера контактов телефонной книги, или подтверждение конца ввода.

Программа выводит строки:

* приглашения ввода номера или телефона,
* подтверждение окончания ввода,
* шапку таблицы,
* строки таблицы, состоящие из строки – номера записи, строки – имени контакта и строки – номера телефона.

# Структура проекта

Файлы проекта располагаются в 2х папках:

* Libs:
  + 21.11.2021 13:52 7 336 PhoneNumerCPP.cpp
  + 19.11.2021 11:57 7 283 PhoneNumerCPP.h
* Lab5
  + 19.11.2021 12:34 436 Lab5.cpp
  + 19.11.2021 11:57 7 326 Lab5.vcxproj
  + 19.11.2021 11:57 1 318 Lab5.vcxproj.filters
  + 10.11.2021 14:50 168 Lab5.vcxproj.user

# Текст программы

//Lab5.cpp

#include <iostream> // input-output streams library

#include "../libs/PhoneNumerCPP.h" // phonebook lib

#include <process.h> // for 'system' functions

int main()

{

Phonebook pb; // create phonebook

pb.input("0"); // input phonebook until "0"

pb.print(50,std::cout,'.'); // print phonebook

if (system(NULL)){ // if 'system' function is available

system("pause"); // do not close console

}

return 0;

}//PhoneNumberCPP.h

#ifndef PHONE\_NUMBER\_CPP\_INCLUDED

#define PHONE\_NUMBER\_CPP\_INCLUDED

#include <string> // for std::string class and string functions

#include <iostream> // input-output streams lib

#include <list> // for list container

/// <summary>

/// Types of string align

/// </summary>

enum StrAlign { left, middle, right };

/// <summary>

/// Inflates string. If string is longer than 'w' characters, cuts the string

/// </summary>

/// <param name="s">the string</param>

/// <param name="w">the width of the string</param>

/// <param name="a">- string align</param>

/// <param name="filler">- character for filling space</param>

/// <returns>inflated string</returns>

std::string inflate\_string(std::string s, size\_t w, StrAlign a=StrAlign::left, char filler=' ');

/// <summary>

/// Asks user.

/// Filters input

/// </summary>

/// <param name="question">- the question</param>

/// <returns>Returns true if user agree, else - false</returns>

bool did\_user\_accept(std::string question);

/// <summary>

/// Class explaning phonebook record

/// </summary>

class PhoneRecord {

protected:

// There is "friending" of comparison operators below

friend bool operator<(const PhoneRecord& a, const PhoneRecord& b);

friend bool operator>(const PhoneRecord& a, const PhoneRecord& b);

friend bool operator==(const PhoneRecord& a, const PhoneRecord& b);

friend bool operator<=(const PhoneRecord& a, const PhoneRecord& b);

friend bool operator>=(const PhoneRecord& a, const PhoneRecord& b);

/// <summary>

/// Filters string (excludes all symbols except digits and '+'

/// if it's the first one)

/// </summary>

/// <param name="s">- filtering string</param>

/// <returns>filtered string</returns>

std::string filter(const std::string& s);

/// <summary>

/// String for nickname

/// </summary>

std::string nickname;

/// <summary>

/// struct that explains phone number

/// </summary>

struct {

/// <summary>

/// phone number

/// </summary>

uint64\_t num : 50;

/\*

According to E.164 (standard phone number recording format)

phone numbers should be not longer than 15 digits.

So greatest phone number is: 999 999 999 999 999.

Then 50 bits are enough for recording this number,

because 2^50 = 1 125 899 906 842 624.

\*/

/// <summary>

/// Does number includes plus flag

/// </summary>

uint64\_t plus : 1;

} number; // object of the nameless struct

public:

/// <summary>

/// Number getter

/// </summary>

/// <returns>phone number as an std::string</returns>

std::string getNumber() const;

/// <summary>

/// Name getter

/// </summary>

/// <returns>nickname</returns>

std::string getName() const;

/// <summary>

/// Constructor with no parameters

/// </summary>

PhoneRecord();

/// <summary>

/// Constructor with initialization

/// </summary>

/// <param name="name">- nickname</param>

/// <param name="number">- phone number as a std::string</param>

/// <param name="checknum">= true, flag if we need filter string</param>

PhoneRecord(std::string name, std::string number, bool checknum=true);

/// <summary>

/// Inputs phone record from the stream

/// </summary>

/// <param name="input">- input stream</param>

/// <param name="output">- output stream for prompt</param>

/// <param name="prompt\_name">- prompt for input</param>

/// <param name="prompt\_number">- prompt for input</param>

void input(

std::istream& input = std::cin,

std::ostream& output = std::cout,

std::string prompt\_name = "Input nickname\n>>> ",

std::string prompt\_number = "Input number\n>>> ");

/// <summary>

/// Number setter with filtering

/// </summary>

/// <param name="number">- std::string, that includes phone number an any format with missprints</param>

/// <param name="filterstr">= true, flag if we need filter string</param>

void setNumber(std::string number, bool filterstr = true);

/// <summary>

/// Name setter

/// </summary>

/// <param name="name">- new nickname</param>

void setName(std::string name);

/// <summary>

/// Prints formated record

/// </summary>

/// <param name="name\_length">- length of the nickname</param>

/// <param name="output">- output stream</param>

/// <param name="filler">fills space with it</param>

void print(size\_t name\_length, std::ostream& output, char filler) const;

};

/// <summary>

/// Class explaining phonedook

/// </summary>

class Phonebook {

protected:

/// <summary>

/// The member of Standard Template Library,

/// that represents an two-linked list.

/// </summary>

std::list<PhoneRecord> phbook;

/// <summary>

/// find lexicographically correct place for record

/// </summary>

/// <param name="r">- record</param>

/// <returns>iterator to the place</returns>

std::list<PhoneRecord>::iterator findPlaceForRecord(const PhoneRecord& r);

public:

/// <summary>

/// adds record by name and number

/// </summary>

/// <param name="name"></param>

/// <param name="number"></param>

/// <param name="checknum">- should we filter number</param>

void addRecord(std::string name, std::string number, bool checknum = true);

/// <summary>

/// adds the record to the phonebook

/// </summary>

/// <param name="r"></param>

void addRecord(PhoneRecord record);

/// <summary>

/// Uses streams to create and add a new record

/// </summary>

/// <param name="input">stream</param>

/// <param name="output">stream</param>

/// <param name="prompt\_name">prompt for name input</param>

/// <param name="prompt\_number">prompt for number input</param>

void inputRecord(

std::istream& input = std::cin,

std::ostream& output = std::cout,

std::string prompt\_name = "Input nickname\n>>> ",

std::string prompt\_number = "Input number\n>>> ");

/// <summary>

/// Adds elements from inputs stream, until

/// reaches record with number "terminator"

/// </summary>

/// <param name="terminator"></param>

/// <param name="input">stream</param>

/// <param name="output">stream</param>

/// <param name="prompt\_name">prompt for name input</param>

/// <param name="prompt\_number">prompt for number input</param>

void input(

std::string terminator = "0",

std::istream& input = std::cin,

std::ostream& output = std::cout,

std::string prompt\_name = "Input nickname\n>>> ",

std::string prompt\_number = "Input number\n>>> ");

/// <summary>

/// Adds "amount" elements from inputs stream

/// </summary>

/// <param name="amount"></param>

/// <param name="input">stream</param>

/// <param name="output">stream</param>

/// <param name="prompt\_name">prompt for name input</param>

/// <param name="prompt\_number">prompt for number input</param>

void input(

size\_t amount,

std::istream& input = std::cin,

std::ostream& output = std::cout,

std::string prompt\_name = "Input nickname\n>>> ",

std::string prompt\_number = "Input number\n>>> ");

/// <summary>

/// prints phonebook in the table

/// </summary>

/// <param name="name\_length">- length of the name column</param>

/// <param name="output">stream</param>

/// <param name="filler">stream</param>

void print(size\_t name\_length, std::ostream& output, char filler);

};

#endif

//PhoneNumberCPP.cpp

#include "PhoneNumerCPP.h" // lib header

// Functions:

std::string inflate\_string(std::string s, size\_t w, StrAlign a, char filler)

{

size\_t n = s.size(); // 's' length

switch (a) { // choose align

case StrAlign::left:

if (n > w) { // if 's' is longer 'w' chars

s.erase(w, n - w); // cut it

}

for (; n < w; n++) { // while 's' is shorter 'w' chars

s = s + filler; // add filler

}

break;

case StrAlign::right:

if (n > w) { // if 's' is longer

s.erase(0, n - w); // cut it

}

for (; n < w; n++) { // while 's' is shorter

s = filler + s; // add filler

}

break;

case StrAlign::middle:

if (n > w) { // if 's' longer

s.erase(0, (n - w) / 2); // cut the head

s.erase(w, (n - w) / 2 + (n - w) % 2); // cut the tail

} else {

s = inflate\_string(s, (w - n) / 2 + n, StrAlign::left, filler);

// add fillers to the right

s = inflate\_string( s, (w - n) / 2 + (w - n) % 2 + s.size(),

StrAlign::right, filler);

// add to the left

}

}

return s; // return result

}

bool did\_user\_accept(std::string question = "Are you sure?")

{

std::string answer; // place for answer

while (1) { // always

std::cout << question << "(y/n)\n>>> "; // prompt for input

getline(std::cin, answer, '\n'); // input

if (answer == "y") { // if user agree

return true; // return true

} else if (answer == "n") { // if user disagree

return false; // return false

} // else repeat

}

}

// PhoneRecord methods:

PhoneRecord::PhoneRecord() = default; // default constructor

PhoneRecord::PhoneRecord(std::string name, std::string number, bool checknum)

{

setName(name); // set name

setNumber(number, checknum); // set number

}

std::string PhoneRecord::filter(const std::string & s)

{

bool is\_it\_first = true;

std::string result; // place for result

size\_t n = s.size(); // source string size

for (size\_t i = 0; i < n; i++) { // for each character in source

char c = s[i]; // place for character

if (('0' <= c and c <= '9') or (is\_it\_first and c == '+')) {

// if 'c' is good (digit or '+' as the first)

result += c; // add 'c' to result string

is\_it\_first = false; // it's not first

}

}

return result; // return result

}

void PhoneRecord::setNumber(std::string s, bool filterstr)

{

if (filterstr) { //if we need filter string

s = filter(s); // filter string

}

if (s.size()) {

number.plus = (s[0] == '+'); // set plus if it's necessary

// else set no plus

number.num = std::stoll(s); // set number

} else {

number = { 0,0 }; // fill number with 0

}

}

void PhoneRecord::setName(std::string s) { nickname = s; } // set nickname

std::string PhoneRecord::getNumber()const

{

std::string res; // string for result

if (number.plus) { res = "+"; } else { res = ""; } // if plus add '+' to res

res = res + std::to\_string(number.num);

// convert number to string and add it to res

return res; // return res

}

std::string PhoneRecord::getName()const { return nickname; } //return nickname

void PhoneRecord::input(std::istream & input, std::ostream & output, std::string prompt\_name, std::string prompt\_number)

{

std::string name, number; // variables for name and number

output << prompt\_name; // prompt for input

getline(input, name, '\n'); // input

output << prompt\_number; // prompt for input

getline(input, number, '\n'); // input

setName(name); // set name

setNumber(number); // set number

}

void PhoneRecord::print(size\_t name\_length, std::ostream & output, char filler) const

{

std::string n = inflate\_string(nickname, name\_length, StrAlign::left, filler);

// format name

std::string num = inflate\_string(getNumber(), 20, StrAlign::right, filler);

// format number

output << n << num << '\n'; // output

}

// There is definition of comparison operators below

// comparison by nickname with standard std::string comparison operators

bool operator<(const PhoneRecord & a, const PhoneRecord & b)

{ return a.nickname < b.nickname; }

bool operator>(const PhoneRecord & a, const PhoneRecord & b)

{ return a.nickname > b.nickname; }

bool operator==(const PhoneRecord & a, const PhoneRecord & b)

{ return a.nickname == b.nickname; }

bool operator<=(const PhoneRecord & a, const PhoneRecord & b)

{ return a.nickname <= b.nickname; }

bool operator>=(const PhoneRecord & a, const PhoneRecord & b)

{ return a.nickname >= b.nickname; }

// Phonebook methods:

std::list<PhoneRecord>::iterator Phonebook::findPlaceForRecord(const PhoneRecord & r)

{

for (std::list<PhoneRecord>::iterator i = phbook.begin(); i != phbook.end(); i++){

if (\*i > r) { //if i becomes less (lexicographically) than new record

return i;

}

}

return phbook.end();

}

void Phonebook::addRecord(std::string name, std::string number, bool checknum)

{

PhoneRecord r(name, number); // create record

std::list<PhoneRecord>::iterator i = findPlaceForRecord(r);

//find place for record in the book

phbook.emplace(i, r); // adds new record to the list of records

}

void Phonebook::addRecord(PhoneRecord r)

{

std::list<PhoneRecord>::iterator i = findPlaceForRecord(r);

//find place for record in the book

phbook.emplace(i, r); // adds new record to the list of records

}

void Phonebook::inputRecord(std::istream & input, std::ostream & output, std::string prompt\_name, std::string prompt\_number)

{

PhoneRecord r; // place for input

r.input(); // input

addRecord(r); // adding result

}

void Phonebook::input(std::string terminator, std::istream & input, std::ostream & output, std::string prompt\_name, std::string prompt\_number)

{

PhoneRecord r; // place for result

do {

while (true) { // always

r.input(input, output, prompt\_name, prompt\_number);

// input record

if (r.getNumber() == terminator) { break; }

// if it's end od the input ask user if he really wanto to exit

addRecord(r); // add record to the list

}

} while (!did\_user\_accept("Do you really want to exit?"));

// continue if user don't want to exit

}

void Phonebook::input( size\_t amount,

std::istream & input,

std::ostream & output,

std::string prompt\_name,

std::string prompt\_number

)

{

PhoneRecord r; // place for input

for (size\_t i = 0; i < amount; i++) { // for 'amount' records

r.input(input, output, prompt\_name, prompt\_number); // input

addRecord(r); // add record to the list

}

}

void Phonebook::print(size\_t name\_length, std::ostream & output, char filler)

{

output << "Printing phonebook with " << phbook.size() << " records:\n";

// message

output << " # " << " name"

<< inflate\_string("number ", name\_length + 16, StrAlign::right) << '\n';

// table header

size\_t n = 0; // counter

for (std::list<PhoneRecord>::iterator i = phbook.begin(); i != phbook.end(); i++) { // for each record in records

output << ' ' << inflate\_string(std::to\_string(++n), 4);

// print # of element

(\*i).print(name\_length, output, filler);

// print table row

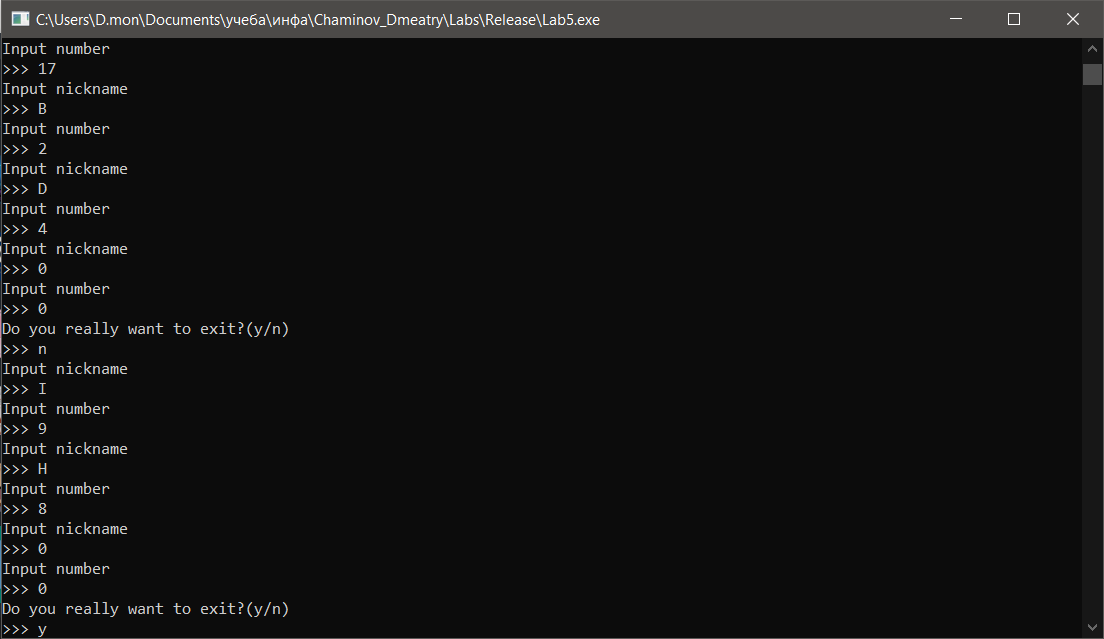
}

}

# Копии экрана при работе программы

Изображение выглядит как текст

Автоматически созданное описание



Изображение выглядит как текст, электроника, компьютер, снимок экрана

Автоматически созданное описание

# Контрольный пример, сравнение с эталоном

# Результаты

К результатам работы относятся характеристики разработанной программы, такие, как: • тип ОС, для которой разработано приложение • тип приложения (консольное, оконное…) • путь, имя и размер исполняемого .exe файла • ограничения на исходные данные • результаты проверки выходных значений по независимому источнику • достоинства и недостатки программы