|  |
| --- |
| **МИНОБРНАУКИ РОССИИ**  **САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  **ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  **«ЛЭТИ» ИМ. В. И. УЛЬЯНОВА (ЛЕНИНА)**  **Кафедра РЭС** |
| **ОТЧЕТ по лабораторной работе №6  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

* разрабатываться на языке C++,
* разрабатываться в среде MS Visual Studio,
* работать в операционных системах MS Windows 7 и выше,
* использовать хотя бы 2 пользовательских класса,
* принимать данные контактов телефонной книги (имя и номер), сортировать их и выводить в виде таблицы с выравниванием.

# Описание алгоритма

Для хранения телефонной книги создается экземпляр класса Phonebook, который содержит всего одно поле типа std::vector<PhoneRecord>. В этом классе определены функция бинарного поиска по записям в книге (методом сдвига границ поиска по результатам сравнения записи в середине отрезка с искомой), функции добавлений записей в книгу, которые при помощи функции двоичного поиска находят место для вставки (реализуя таким образом сортировку вставками), функции ввода телефонной книги из потока ввода, использующие функцию добавления записи, пока не считают терминальный элемент (нулевой номер), функция распечатки, которая вызывает распечатку для каждой записи в векторе.

Класс PhoneRecord содержит член – структуру, содержащую два поля – телефонный номер (выделено 50 бит, так как этого хватит для представления 15-значного числа) и индикатор того, начинается ли номер с символа ‘+’. Для этого класса определены дружественная функция сравнения int8\_t compair, которая сравнивает строки имен двух контактов без учета регистра, функция фильтрации строки телефонного номера, которая проверяет каждый символ строки на то, является он ведущим плюсом или цифрой, «get’теры» и «set’теры» для имени и номера телефона (так как они находятся в поле спецификатора доступа protected), конструктор по умолчанию и инициализирующий конструктор, функция ввода записи из потока ввода, функция вывода записи в поток вывода, использующая функцию расширения строки inflate\_string.

inflate\_string добавляет указанные символы к строке так, чтобы исходная строка выравнивалась либо по левому, либо по правому краю, любо по центру и занимала указанный объем.

При старте программа создает телефонную книгу, вызывает функцию ввода через стандартный поток ввода из консоли, затем вызывает функцию вывода телефонной книги в стандартный поток вывода на консоль.

# 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: number

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 = findRecord(r);

получить PhoneRecord r

начало

Алгоритм Phonebook:: FindRecord:

нет

да

да

нет

да

нет

m=(left+right)/2;

конец

right=m;

left=m;

phbook[m]>=r

вернуть phbook.begin()+1+left

phbook.size() == 0

вернуть phbook.begin()

size\_t left=-1,

right = phbook.size();

size\_t m=(left+right)/2;

right-left>1

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:

вернуть s

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

Программа использует пользовательский тип Phonebook – класс определяющий работу с телефонной книгой. Внутри него содержится контейнер std::vector<PhoneRecord>, являющий автоматическим динамическим массивом. Случайный доступ к элементам позволяет использовать двоичный поиск. В качестве полей имени и для строкового представления номера используется тип std::string (std::basic\_string<char>), так как это позволяет более удобно вводить и выводить данные, а также обеспечивает автоматический контроль за памятью. Для хранения телефонного номера используется анонимная структура, содержащая два поля uint64\_t num : 50 (для хранения цифр номера телефона, 50 бит обеспечивает представление 15-значного числа), и uint64\_t plus : 1 (индикатор того, начинается ли номер с плюса).

# Вводимые и выводимые параметры и их типы

На ввод программе через стандартный поток ввода консоли после приглашения ко вводу подаются имена контактов и телефонные номера. Для окончания ввода необходимо в качестве телефонного номера ввести «0» или строку, не содержащую цифр или знака «+». Эти данные записываются в переменные типа std::string. После завершения ввода необходимо ввести «y», если вы подтверждаете окончание ввода, и «n» если вы хотите продолжить ввод.

После подтверждения окончания ввода начнется вывод записей в виде таблицы. В качестве шапки таблицы выводятся char[25], size\_t, char[11], а также строки, длины которых задаются параметрически. Затем выводятся строки таблицы (состоящие из номера записи size\_t, имени контакта std::string и номера телефона в строковом представлении std::string), длину выводимого имени можно задать, в программе она равна 40 символов. Имя выравнивается по левому краю. После имени в 20 символов с выравниванием по правому краю выводится телефонный номер.

# Структура проекта

Файлы проекта располагаются в двух директориях.

Labs\Lab6:

* 24.11.2021 14:04 409 Lab6.cpp
* 22.11.2021 15:37 7 326 Lab6.vcxproj
* 22.11.2021 15:37 1 318 Lab6.vcxproj.filters
* 22.11.2021 15:23 168 Lab6.vcxproj.user

Labs\libs:

* 28.11.2021 18:27 9 203 PhoneNumerCPP.cpp
* 28.11.2021 00:51 7 999 PhoneNumerCPP.h

# Текст программы

// Lab6.cpp

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

#include <iostream> // for std::cout

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

int main()

{

Phonebook pb; // phonebook declaration (default constructor called)

pb.input(); // input the phonebook

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

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

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

}

return 0; // Exit success

}

//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 <vector> // for vector container

#include <fstream> // for file stream

/// <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);

[[deprecated("Use int8\_t compair(const PhoneRecord&, const PhoneRecord&) instead of this")]]

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

[[deprecated("Use int8\_t compair(const PhoneRecord&, const PhoneRecord&) instead of this")]]

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

[[deprecated("Use int8\_t compair(const PhoneRecord&, const PhoneRecord&) instead of this")]]

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

[[deprecated("Use int8\_t compair(const PhoneRecord&, const PhoneRecord&) instead of this")]]

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

[[deprecated("Use int8\_t compair(const PhoneRecord&, const PhoneRecord&) instead of this")]]

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

/// <summary>

/// Class explaning phonebook record

/// </summary>

class PhoneRecord {

protected:

// There is "friending" of comparison operators below

// NOTA BENE: case-sensitive

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>

/// case-insensitive record names compairing

/// </summary>

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

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

/// <returns>

/// -1 if first record name is more than second,

/// 0 if records names are equal,

/// 1 if second record name is more than first.

/// </returns>

friend int8\_t compair(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 = std::cout,

char filler=' ') const;

};

/// <summary>

/// Class explaining phonedook

/// </summary>

class Phonebook {

protected:

/// <summary>

/// The member of Standard Template Library,

/// that storages my phone records

/// </summary>

std::vector<PhoneRecord> phbook;

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>

/// Input records from file till the end of it

/// </summary>

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

void inputFromFile(std::ifstream & fin);

/// <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 = std::cout,

char filler = ' ');

/// <summary>

/// Binary search

/// Find nearest record (lexicographically) in the book

/// </summary>

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

/// <returns>iterator to the record</returns>

std::vector<PhoneRecord>::iterator findRecord(const PhoneRecord& r);

};

#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

// NOTA BENE: case-sensitive

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; }

int8\_t compair(const PhoneRecord & a, const PhoneRecord & b)

{

size\_t al = a.nickname.size(), bl = b.nickname.size(); // characters amount

for (size\_t ai = 0, bi = 0; ai < al && bi < bl; ai++, bi++) { // for each char

char ac = tolower(a.nickname[ai]), bc = tolower(b.nickname[bi]); // changing case if it needs

if (ac > bc) {

// if current a char is more than current b char

return 1;

} else if (ac < bc) {

// if current b char is more than current a char

return -1;

}

}

if (al == bl) { return 0; } // equal strings

if (al < bl) { return -1; } // a is substr of b

return 1; // b is substr of a

}

// Phonebook methods:

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

{

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

std::vector<PhoneRecord>::iterator i = findRecord(r);

//find place for record in the book

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

}

void Phonebook::addRecord(PhoneRecord r)

{

std::vector<PhoneRecord>::iterator i = findRecord(r);

//find place for record in the book

phbook.emplace(i, r);

// adds new record to the vector 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 vector

}

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

// continue if user don't want to exit

}

void Phonebook::inputFromFile(std::ifstream & fin)

{

PhoneRecord r; // buffer PhoneRecord

while (fin.good()) { // while there is no EOF

r.input(fin, std::cout, "", ""); // input record

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

}

}

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 vector

}

}

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::vector<PhoneRecord>::const\_iterator i = phbook.cbegin();

i != phbook.cend();

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

}

}

std::vector<PhoneRecord>::iterator Phonebook::findRecord(const PhoneRecord & rec)

{

if (!phbook.size()) { return phbook.begin(); }

// if phonebook is empty - push first

size\_t l = static\_cast<size\_t>(-1ll), r = phbook.size();

// l=0xFFF...F=-1 - left border, r - right border

size\_t m; // middle

while (r - l > 1) { // while element is not found

m = r + l;

m /= 2; // middle index calculated

if (compair(phbook[m], rec) != -1) { // if searching placed before

r = m; // move right border

} else { // else

l = m; // move left border

}

}

return phbook.begin() + 1 + l; // return iterator to this element

}

# Копии экрана при работе программы

\*скриншоты (лучше win+shift+s в режиме окна)\*

# Контрольный пример, сравнение с эталоном

Ну тут типа информация по прохождению теста / тестов

# 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