Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет Компьютерных Систем и Сетей

Кафедра электронных вычислительных машин

**ОТЧЕТ**

по учебной практике (ознакомительной)

на тему:

**«Программа для вычисления хэш-суммы логина и пароля. И проверка авторизации»**

|  |  |
| --- | --- |
| Выполнил(а): | Бейнар А.В.  Студент гр.150505 |
| Проверил(а): | Лукьянова И.В |

Минск 2022

**Содержание**

[**ВВЕДЕНИЕ** 3](#_Toc106879999)

[**1** **ОБЩИЕ СВЕДЕНИЯ О ХЭШИРОВАНИИ** 4](#_Toc106880000)

[**2** **ОПИСАНИЕ АЛГОРИТМА РАБОТЫ ПРОГРАММЫ** 14](#_Toc106880001)

[**3** **РАЗРАБОТКА СХЕМ АЛГОРИТМОВ РАБОТЫ ПРОГРАММЫ** 17](#_Toc106880002)

[**4** **ТЕСТИРОВАНИЕ И ПРОВЕРКА РАБОТОСПОСОБНОСТИ ПРОГРАММЫ** 23](#_Toc106880003)

[**5** **ИНСТРУКЦИЯ ПО РАЗВЕРТЫВАНИЮ ПРИЛОЖЕНИЯ И СКВОЗНОЙ ТЕСТОВЫЙ ПРИМЕР** 26](#_Toc106880004)

[**5.1 Модуль администратора** 26](#_Toc106880005)

[**5.2 Модуль пользователя** 29](#_Toc106880006)

[**ЗАКЛЮЧЕНИЕ** 30](#_Toc106880007)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ** 31](#_Toc106880008)

[**ПРИЛОЖЕНИЕ А** 32](#_Toc106880009)

# **ВВЕДЕНИЕ**

Данная работа заключается в вычислении хэш-суммы логина и пароля и проверки авторизации. В данной работе будет показан алгоритм реализации хэш-функции занесения в хэш-таблицу, а также хэширование данных. Хэш-таблица будет реализована с помощью структур, методом цепочек. Также тут будут приведены блок-схемы для функций и листинг кода на языках С/C++.

## **ОБЩИЕ СВЕДЕНИЯ О ХЭШИРОВАНИИ**

Для начала нам необходимо понять зачем вообще нужно правильно хранить пароли. Правильно организованный алгоритм хранения паролей должен:

1. Снизить риск полного взлома системы

2. Предотвратить утечку паролей пользователей

Рассмотрим когда система имеет какие-либо теоретические уязвимости. Неправильный алгоритм позволит в первом случае получить пароль администратора через некую имеющуюся уязвимость, далее попасть в админ-панель, и далее по обстановке, т.е. в 90% случаев это означает полный взлом. Во втором случае если злоумышленник каким-то образом получает базу всех паролей, то это позволит скомпрометировать некоторых пользователей, которые, к примеру, используют один пароль на все ресурсы. Правильный же алгоритм должен вообще предотвратить получение злоумышленником паролей.

На практике это выглядит так. Злоумышленник получает логин-пароли всех пользователей. При правильной системе хранения паролей злоумышленник получает не исходные пароли а только их хеш суммы (см.далее).

**Хеш сумма**

Разработчики пришли к выводу что использовать открытое хранение паролей в своих системах занятие не безопасное и надо придумать что-то другое. И тут на помощь пришли [хеш суммы](http://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

Результат, производимый хеш-функцией, называется «хэш-суммой» или же просто «хэшем», а входные данные часто называют «сообщением». Допустим у пользователя пароль «123456». Придумаем свою хеш-функцию. Сложим все цифры получим число «21» — это и будет являться результатом нашей хеш функции. В этой функции один и тот же хеш может соответствовать большому количеству различных паролей. То есть пароли «654321», «555222», «100299» и т.д будут давать такой же хеш. Это свойство называется [коллизией](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BB%D0%BB%D0%B8%D0%B7%D0%B8%D1%8F_%D1%85%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8).

Идеальная хеш функция должна обладать следующими параметрами:

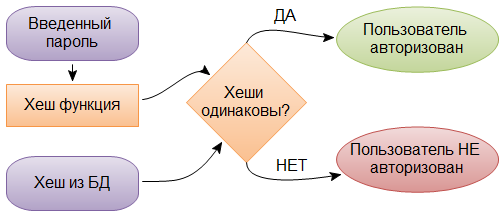
1. Детерминированность – то есть одно и то же сообщение приводит к одному и тому же хэш-значению.

2. Необратимость — хеш сумма не должна «расшифровываться» подобно обычным алгоритмам шифрования.

3. Отсутствие коллизий — для каждых данных проходящих через хеш-функцию должен получится уникальный хеш

4. Лавинный эффект – небольшое изменение в сообщении изменяет хэш очень сильно.

Третий параметр не достижим для хешей с фиксированной длиной (а таких алгоритмов сейчас большинство).

И теперь при регистрации пользователя, указанный им пароль проходит через хеш-функцию и вместо пароля в БД будет занесен полученный хеш. При каждой попытке авторизации указанный пароль будет каждый раз проходить через хеш-функцию и полученный хеш будет сравниваться с хешем хранимым в БД, и если хеши будут соответствовать значит пароль был указан верный. Вообщем как то так:  


Теперь злоумышленнику необходимо будет попытаться восстановить пароль из хеша, и причем если алгоритм хеш-функции полностью необратим, то для злоумышленника останется лишь один метод — [брутофорс](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%BD%D1%8B%D0%B9_%D0%BF%D0%B5%D1%80%D0%B5%D0%B1%D0%BE%D1%80).

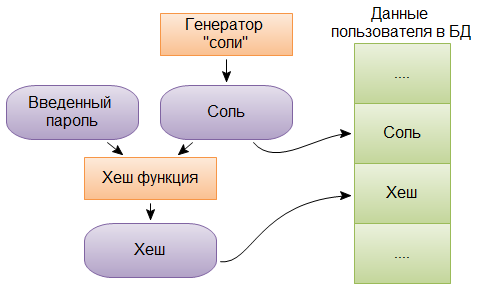
Брутофорс — это перебор всех возможных паролей до тех пор пока хеш от одного из них не совпадет с исходным хешем.

**«Соленые» хеши**

Вообщем все бы хорошо, но если бы не ленивые или забывчивые пользователи которые стремятся использовать максимально короткие и простые пароли… Почему это плохо? Потому что сбрутить короткие пароли можно в считанные минуты, а простые (часто используемые) пароли брутятся по словарям.

Как же нам защитить пользователей и свой ресурс в случае его взлома? На помощь проходит [соль](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BB%D1%8C_(%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F)). Грубо говоря соль — это набор случайный символов который каждый раз перед прохождением через хеш-функцию добавляется к паролю. Таким образом соль используется для усложнения определения прообраза хэш-функции.

Различают статическую соль (одна и та же для всех входных значений) и динамическую (генерируется для каждого входного значения персонально).

При регистрации пользователя генерируется случайная соль, на основе которой и указанного пароля генерируется «соленый» хеш, при этом соль также заносится в БД:  
  
Что дает соль в этом случае? Если подумать, то если злоумышленник имеет доступ к хешам пользователей, то если соль каждого хеша мы храним рядом (в той же таблице/БД), то злоумышленник также будет иметь и доступ к соли. То есть сможет найти исходный пароль методом брутофорса, но словари ему уже не подойдут, так как не существует словарей учитывающих все комбинации паролей с солью.

Теперь рассмотрим второй плюс. Допустим вы владеете ресурсом с десяткой тысяч зарегистрированных пользователей. Каждый пароль пользователя имеет уникальную соль. Что это даст? Это даст невозможность одновременного брута хешей всех пользователей, так как для каждой соли нам придется генерировать новый хеш. И если вдруг злоумышленник сбрутить все аккаунты, то ему придется на каждый вариант пароля генерировать столько хешей сколько пользователей у вас имеется. При этом скорость брутофорса упадет пропорционально.

Вообщем такой алгоритм предотвращает утечку паролей пользователей.

Третий плюс заключается в том, что, при добавлении разных солей к одинаковым паролям у разных пользователей, хэш получится различный.

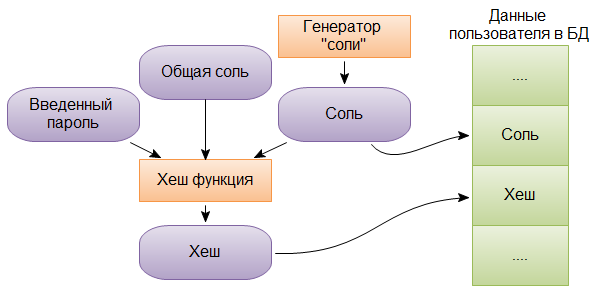
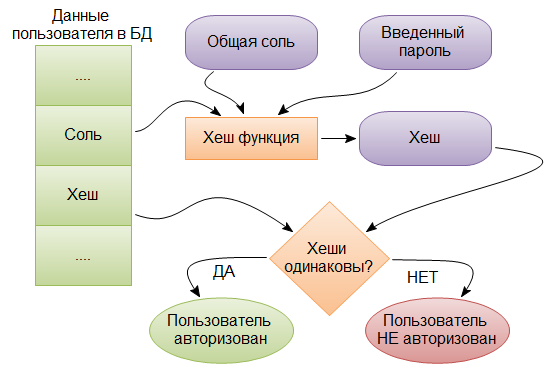
**Дважды «соленый» хеш**

Но вот другая сторона медали. В случае целенаправленной атаки на определенного пользователя (например администратора ресурса), взломщика не остановит такая соль, как написано выше, так как ему придется брутить всего один хеш с уже приемлемой скоростью.

В нашем случае поможет еще одна соль, но уже общая для всех хешей, которая допустим будет хранится отдельно от хеша, т.е в любом другом месте, к примеру в конфиге самого приложения (статическая соль).

Если подвести итог, то смысл данной соли заключается в снижении риска полного взлома системы при наличии у злоумышленника лишь частичного доступа.

Главная идея этого принципа — раздельное хранение хеша и общей соли, т.е. постараться не допустить попадание соли в руки злоумышленника Для практической реализацией в момент настройки системы, создается общая соль, которая будет хранится в конфиге приложения, и далее эта соль будет также добавляться к паролю.

Если схематично то это будет выглядеть вот так:  
  
При этом проверка правильности пароля при авторизации будет выглядеть так:  


**Алгоритмы хеширования**

Конечно полностью защититься от брута невозможно. Но в наших силах сделать брутофорос бессмысленным. В своих системах необходимо применять такой алгоритм хеширования, который требует довольно больших ресурсов и большого количества операций для вычисления хеша.

Таковой является криптографическая хэш-функция **MD5**.

Криптографическая хеш-функция – это математический алгоритм, который отображает данные произвольного размера (например логин и пароль) в битовый массив фиксированного размера (конечный результат).

Например для перебора простенького пароля из 6 симолов латиницы верхнего и нижнего регистров и цифр, то есть это всего около 100 000 000 000 комбинаций, среднее время, которое нам понадобится, ­ ~45 000 секунд или ~12 часов.

**Коллизии**

К сожалению все было бы слишком хорошо если бы все так было бы просто. Не стоит забывать про то что для вашего хеша от супер сложного пароля в 30 символов может быть найдена коллизия длиной в 1 символ. Конечно вероятность эта крайне мала, но она есть.

Поэтому схема хеширования должна включать **алгоритм разрешения конфликтов.**

**Базовые понятия**

Хеширование так же может использоваться как метод адресации данных **по их уникальным ключам (key)** для быстрого поиска нужной информации.

**Хеш-таблица** представляет собой обычный массив со специальной адресацией, задаваемой некоторой функцией.

Хеш-функция в данном случае будет определятся как: функция, которая преобразует ключ элемента данных в некоторый индекс в хеш-таблице, называется функцией хеширования или хеш-функцией:

**i = h(key)** где key- преобразуемый ключ, i- получаемый индекс таблицы, т.е. ключ отображается во множестве, например, целых чисел (**хеш-адреса**), которые впоследствии используются для доступа к данным.

Хеширование таким образом – это способ, который подразумевает использование значения ключа для определения его позиции в специальной таблице.

Итак, **алгоритм разрешения конфликтов** определяет порядок действий, если позиция i=h(key) оказывается уже занятой записью с другим ключом.

Имеется множество схем хеширования, различающихся и используемой хеш функцией h(key) и алгоритмами разрешения конфликтов.

Наиболее распространенный метод задания хеш-функции: **метод деления.**

Исходными данными являются: - некоторый целый ключ key и размер таблицы m. Результатом данной функции является остаток от деления этого ключа на размер таблицы. Общий вид такой функции на языке программирования С/С++:

**int h(int key, int m)**{

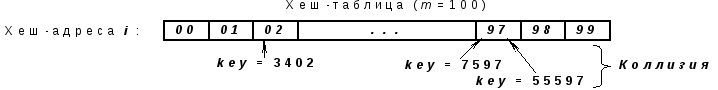
returnkey%m;

}

Для m= 10 хеш-функция возвращает младшую цифру ключа.



Для m= 100 хеш-функция возвращает две младших цифры ключа.



В рассмотренных примерах хеш-функция i=h(key) только определяет позицию, начиная с которой нужно искать (или первоначально - поместить в таблицу) запись с ключом key.

В данной программе эта функция будет иметь вид:

int h(char\* word)

{

int index = 0;

int i = 0;

while (word[i] != '\0') index += word[i++] - '0';

return index % size;

}

При поставленной задаче, от пользователя мы получаем только символьный тип данных. Поэтому функция работает с символами. Она высчитывает суммарный код всего слова и берет остаток от деления на размер таблицы, для получения индекса в этой таблице.

Таким образом для получения хэш-суммы логина и пароля, надо пропустить оба параметра через эту функцию. Тогда для получения финального индекса надо опять найти остаток от деления на размер таблицы. Реализация этого метода в функции user\*\* registration(user\*\* ms) будет иметь вид:

sum += h(st->login);

sum += h(st->password);

sum = sum % size; // получение индекса

Далее необходимо воспользоваться какой – либо схемой (алгоритмом) хеширования.

В большинстве задач два и более ключей хешируются одинаково, но они не могут занимать в хеш-таблице одну и ту же ячейку. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш-таблицы отдельный список, в который помещаются все ключи, отображающиеся в этот индекс.

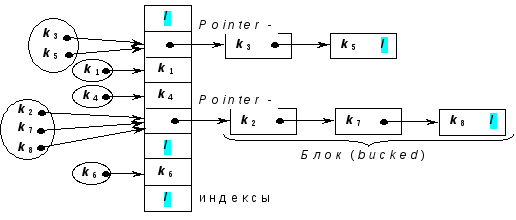
Эти варианты и представляют собой две классические схемы хеширования:

1. Хеширование методом открытой адресацией с линейным опробыванием.

2. Хеширование методом цепочек (со списками), или так называемое, многомерное хеширование.

В этой программе будет использоваться метод цепочек. На нем подробней и остановимся.

**Метод цепочек** является доминирующей стратегией**.**В этом случае i, полученной из выбранной хеш-функцией h(key)=i, трактуется как индекс в хеш-таблице списков, т.е. сначала ключ key очередной записи отображается на позицию i = h(key) таблицы. Если позиция свободна, то в нее размещается элемент с ключом key, если же она занята, то отрабатывается алгоритм разрешения конфликтов, в результате которого такие ключи помещаются в список, начинающийся в i-той ячейке хеш-таблицы. Например



В итоге имеем таблицу массива связных списков.

Процесс заполнения (считывания) хеш-таблицы прост, но доступ к элементам требует выполнения следующих операций:

- вычисление индекса i;

- поиск в соответствующей цепочке.

**Реализация метода цепочек**. Объявляем структурный тип для элемента списка (однонаправленного):

struct zap {

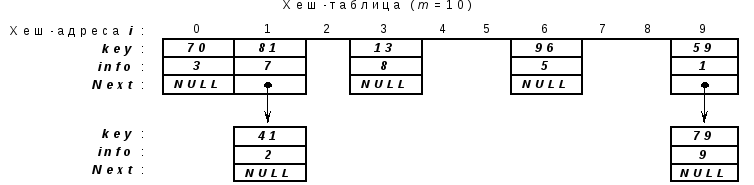
int key; // Ключ

int info; // Информация

zap\*Next; // Указатель на следующий элемент в списке

} data;

На основании исходных данных последовательно заполняем хеш-таблицу, добавляя новый элемент в конец списка, если место уже занято.



Хеширование первых пяти ключей дает различные индексы (хеш-адреса): 9, 0, 6, 1, и 3.

При возникновении коллизии, новый элемент добавляется в конец списка. Поэтому элемент с ключом 41, помещается после элемента с ключом 81, а элемент с ключом 79 - после элемента с ключом 59.

В данной работе структура будет иметь вид:

struct user

{

char login[length]; // Логин

char password[length]; // Пароль

char hash[64]; // Хешированный логин и пароль

char sol[SALT\_SIZE]; // Случайная соль

user\* next;

};

При добавлении пользователя, в структуру записывается логин и пароль. Дальше формируется строка состоящая из логина, пароля, случайной соли, которая записывается отдельно, и общей соли. Эта строка посылается в функцию хэширования MD5. И в итоге на выходе мы имеем строку фиксированной длина, которую и записываем в поле hash.

Как писалось выше, случайную соль надо хранить рядом с пользователем, чтобы потом, при проверке авторизации, можно было получить такой же хэш при таких же входных данных (логине и пароле).

В итоге с сформированной строкой хэша и вычисленному индексу, заносим структуру в конец списка.

При проверке авторизации, надо просматривать каждого пользователя. А точнее их соль и хэш. К введенным данным добавляем соль просматриваемого пользователя, получаем хэш и сравниваем его с хэшем прочитанного пользователя. Если они совпадают, то пользователь авторизован.

Можно сделать несколько выводов:

1. Пароль не должен хранится в БД в открытом виде, а должна хранится лишь хеш-сумма.
2. При регистрации пользователя желательно рекомендовать использовать более сложный пароль.
3. Каждый пользовательский хеш необходимо генерировать с уникальной солью.
4. К пользовательской соли должна быть добавлена общая соль, которая хранится в другом месте (отдельно от пользовательских данных).
5. Соль должна быть достаточно длинной.
6. Алгоритм вычисления хеш суммы должен быть ресурсоемок.
7. Нужен алгоритм разрешения конфликтов.

## **ОПИСАНИЕ АЛГОРИТМА РАБОТЫ ПРОГРАММЫ**

Работа программы включает в себя несколько этапов. Первым из них является авторизация.

Пользователю предоставляется следующая возможность входа:

* в качестве администратора;
* в качестве пользователя.

При выборе пункта входа как администратора, пользователю необходимо ввести свои логин и пароль. На следующем этапе считываются данные из файла с учетными записями администраторов следующего вида:

* login;
* password;
* hash;
* sol.

После ввода пользователем своих логина и пароля, сверяется совпадение для каждого хэша с хэшем, который получается использованием введенных данных от пользователя и соли хэша с которым сравниваем. Если есть совпадение хоть с одним хэшем (даже для разных аккаунтов, а соответственно и данных хэши могут получиться одинаковыми), открывается меню администратора.

При выборе пункта входа как пользователя, открывается меню пользователя. В сравнении с администраторским меню, это меню имеет меньший функционал.

На рисунке 2.1 представлена диаграмма вариантов использования программного продукта. На ней показаны функции, которые доступны пользователю и администратору.

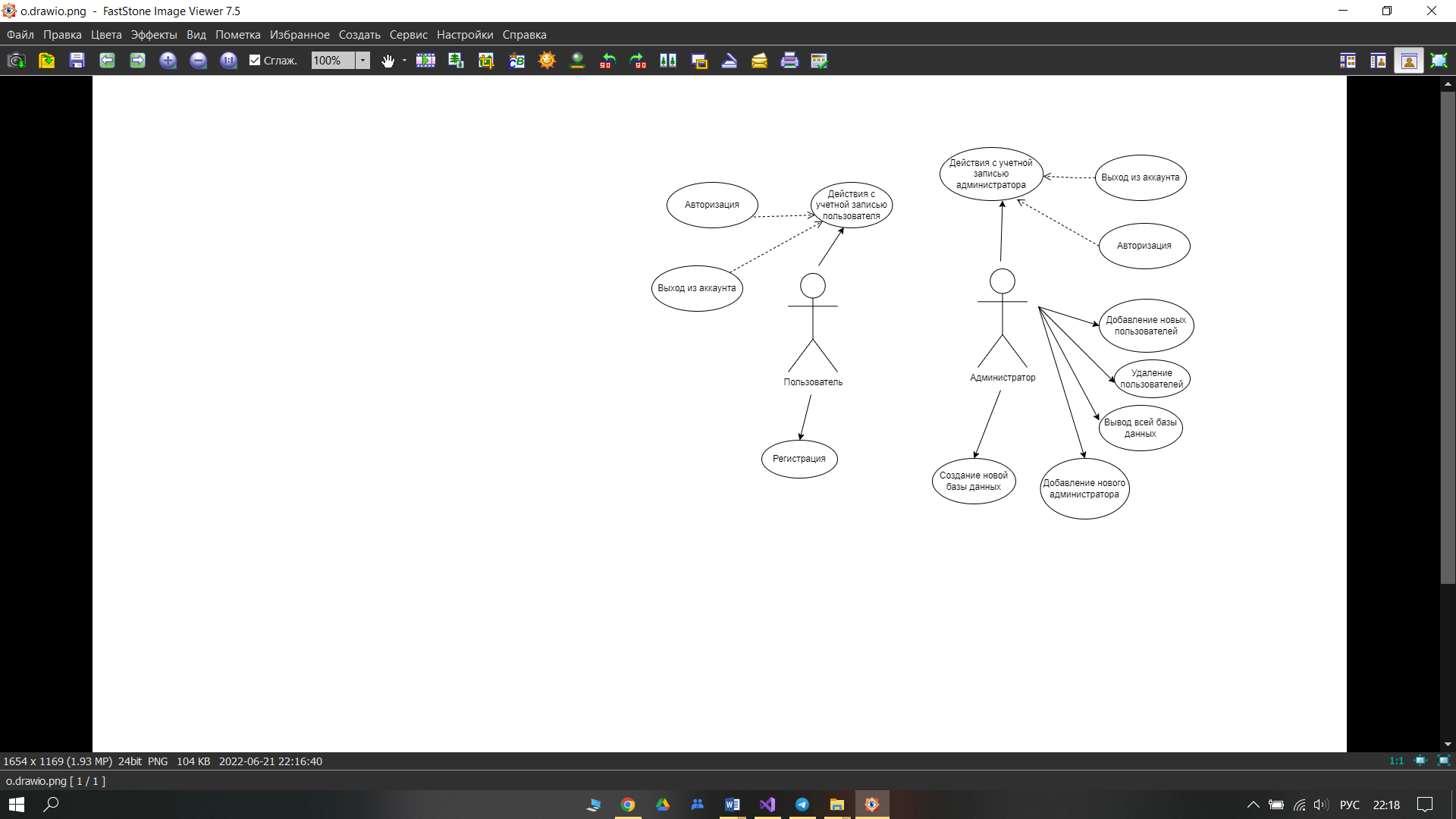


Рисунок 2.1 – Диаграмма вариантов использования

Как видно из рисунка 2.1 регистрация новых пользователей и проверка на авторизацию доступна как администратору, так и самому пользоватею.

При добавлении новых пользователей выполняется проверка потенциальных учетных записей на уникальность логина. Также программа запрещает использование всех символов в логине и пароле, кроме букв латинского алфавита и цифр. При добавлении пароля, он проверяется на сложность и, если был введен очень простой (не использованы в достаточном количестве буквы) или короткий пароль, его именно предлагают поменять, а не вынуждают.

В работе предусмотрены:

* обработка исключительных ситуаций;
* возможность возврата назад (навигация);

Для описания входных данных были выбраны структуры. В качестве способа объединения входных данных используются хэш-таблица и списки (о чем нам говорит поле user\* next в структуре).

Программа содержит следующие функции:

//проверка на администратора

bool admin\_check();

//меню администратора

int admin\_menu();

//меню пользователя

int user\_menu();

//функция хэширования

void hashing(char\* str, user\* st);

// функция получения индекса в хэш таблице

int h(char\* word);

// функция добавления нового пользователя и добавления его по индексу хэш-суммы в массив

user\*\* registration(user\*\* ms);

//функция записи массива в файл

void exit(user\*\* ms, char\* name);

// функция записи в массив из файла

user\*\* write\_from\_file\_to\_array(char\* name, user\*\* ms);

// функция удаления элемента в массиве

user\*\* del(user\*\* ms, user\* st);

// функция вывода массива

void print(user\*\* ms);

// функция очистки памяти использованной под массив

void FREE(user\*\*\* ms);

// функция проверки авторизации пользователя

bool authorization\_check(user\*\* ms, user\* st, int sum);

// функция проверки сложности пароля и логина

bool check\_difficulty(user\* st);

// функция копирования в новую строку только букв из str

char\* get\_letters\_from\_line(char\* str);

// функция для проверки желания продолжить

bool proceed();

// функция проверки наличия такого же логина в базе

bool check\_login(user\*\* ms, char\* str);

// функция проверки использованных символов

bool check\_letters(char\* str).

## **РАЗРАБОТКА СХЕМ АЛГОРИТМОВ РАБОТЫ ПРОГРАММЫ**

Разработка алгоритмов является важной частью создания программы. Они записываются в виде последовательности шагов решения задачи от ввода исходных данных до вывода результатов. Разработка алгоритма включает в себя выбор методов проектирования алгоритма, выбор формы записи алгоритма, выбор метода тестирования. На рисунках 3.1, 3.2, 3.3, 3.4 показаны алгоритмы главной функции «main», функции «hashing», функции «registration» и функции «authorization\_check» соответственно. Листинг кода представлен в приложении А.

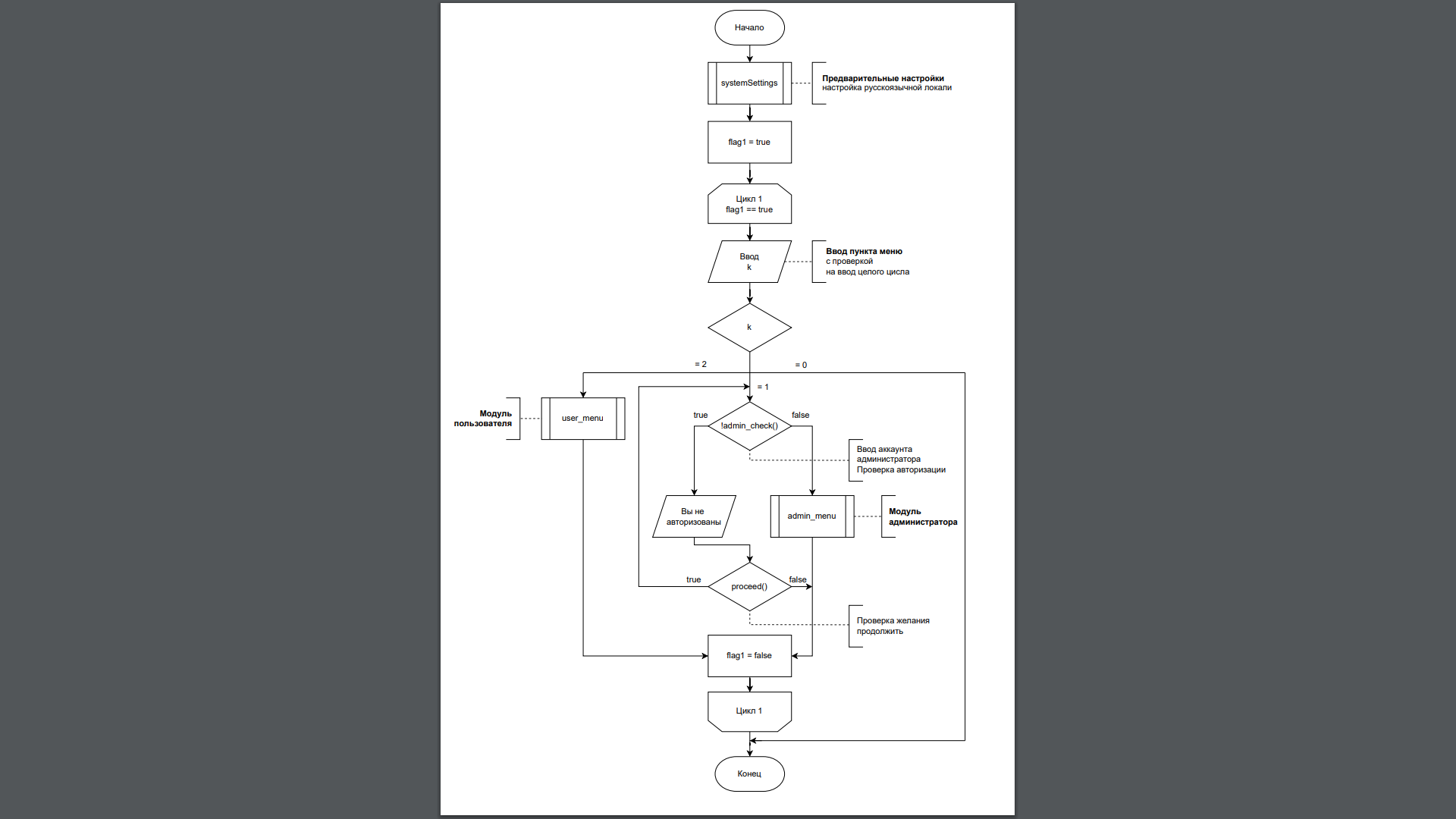


Рисунок 3.1 – Алгоритм главной функции «main»

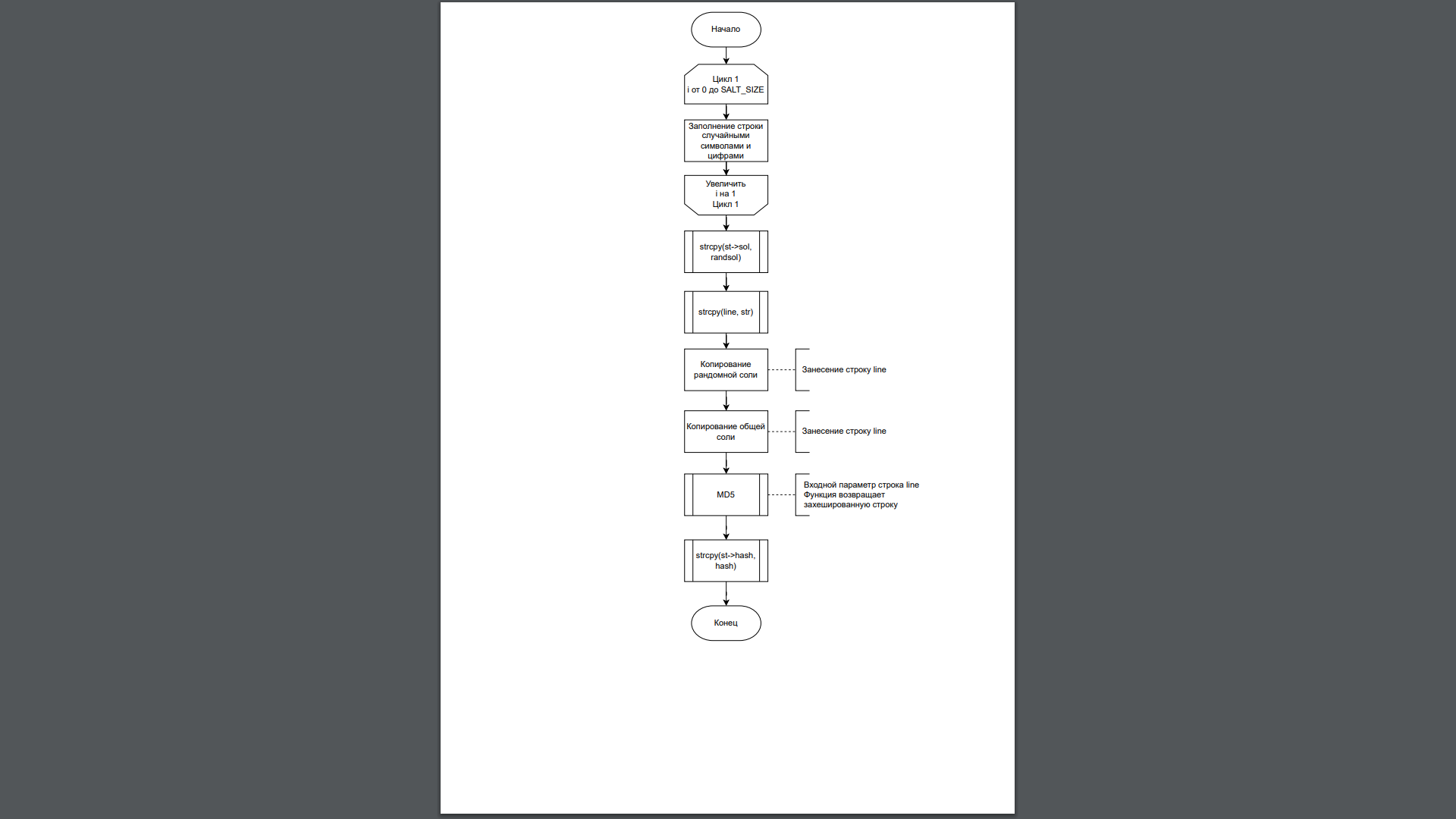
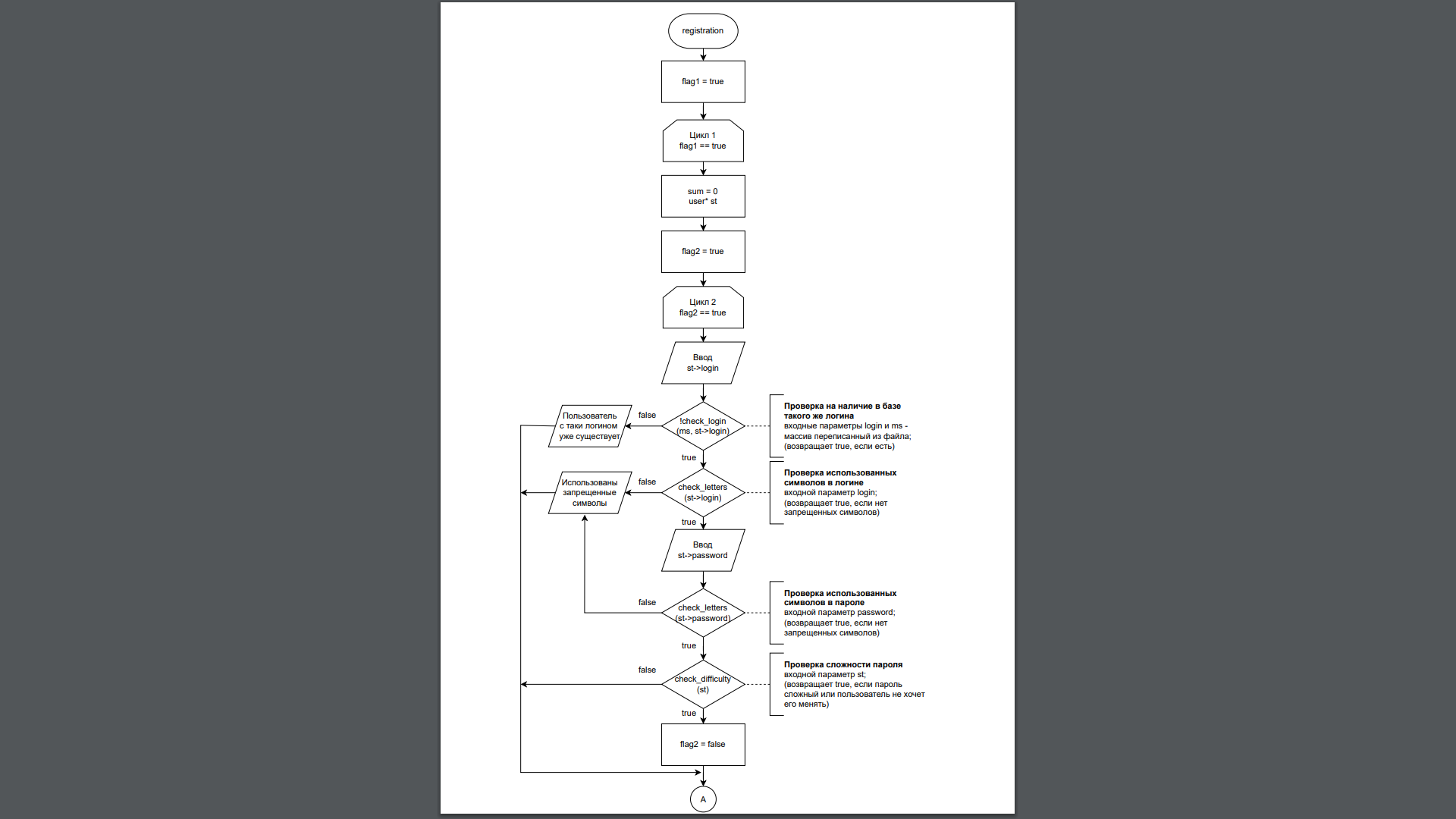


Рисунок 3.2 – Алгоритм функции «hashing»



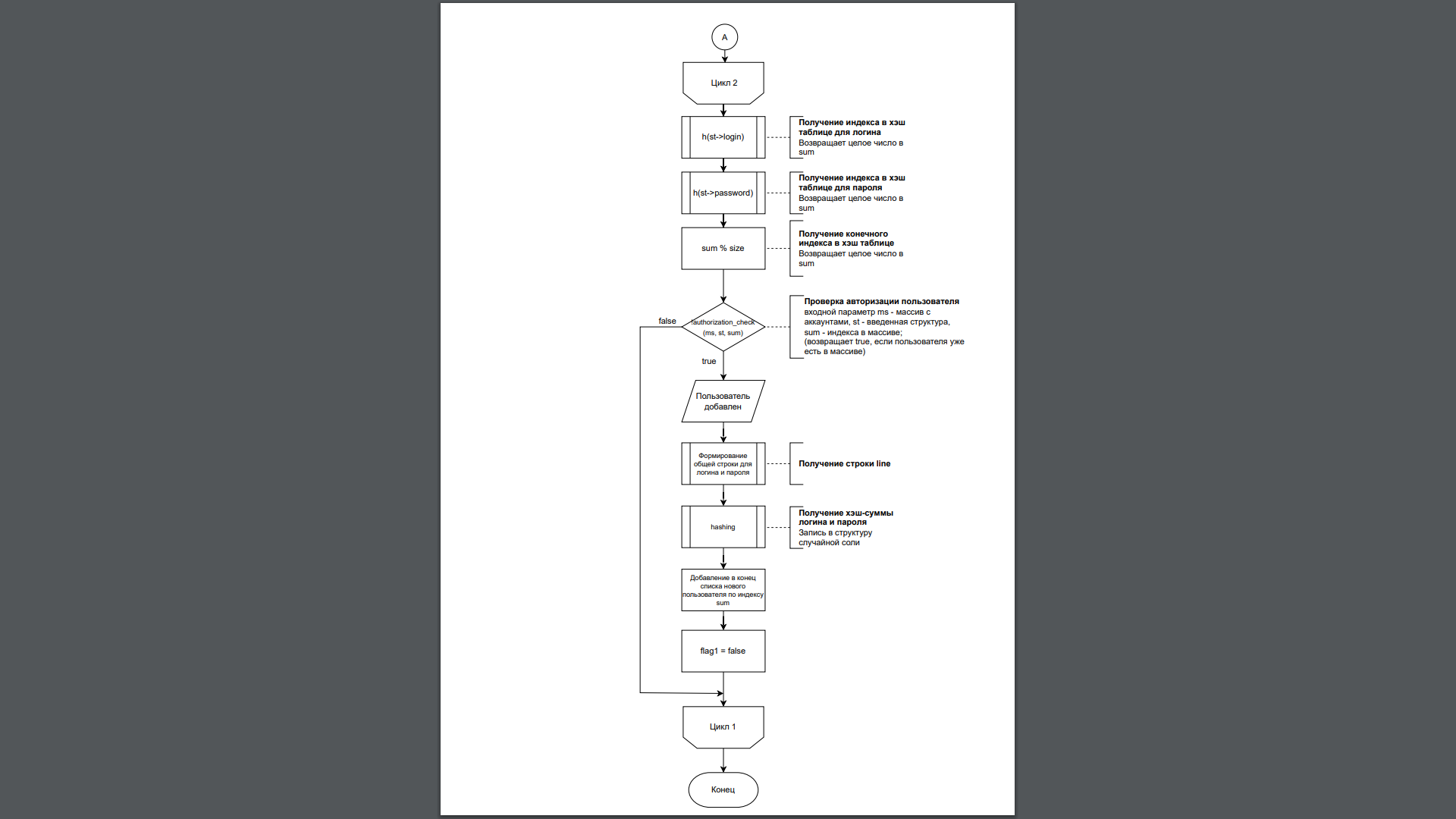


Рисунок 3.3 – Алгоритм функции «registration»

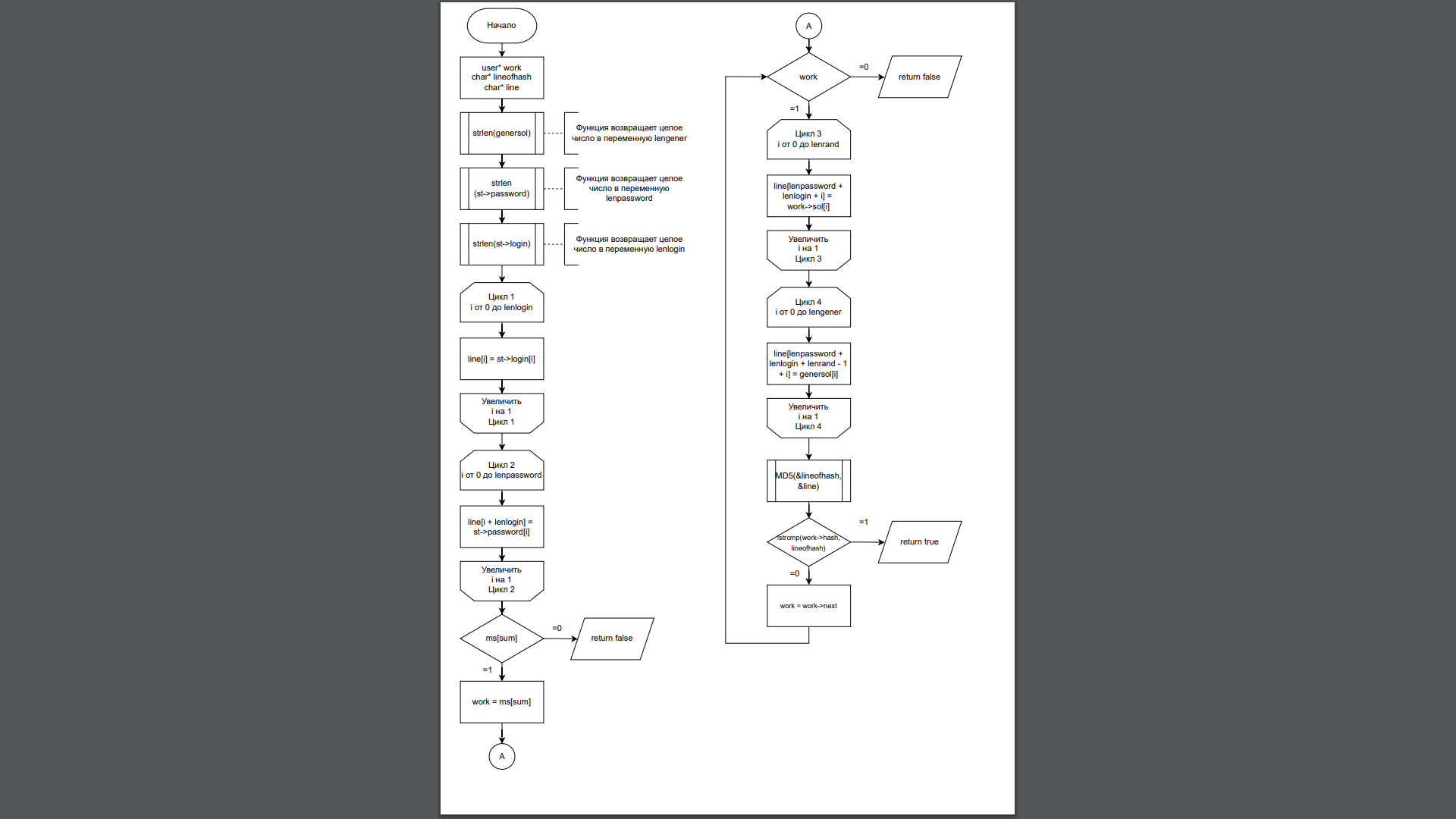


Рисунок 3.4 – Алгоритм функции «authorization\_check»

## **ТЕСТИРОВАНИЕ И ПРОВЕРКА РАБОТОСПОСОБНОСТИ ПРОГРАММЫ**

Исключительные ситуации – ситуации, возникающие во время работы системы, выходящие за пределы, предусмотренные спецификацией. Для того, чтобы предотвратить такие случаи необходимо в коде программы предусмотреть защиту от неправильных действий.

В первую очередь стоит уделить внимание на неправильный ввод отрицательных чисел, выбор не существующего пункта меню, ошибку открытия файла и/или чтения и записи данных "в никуда", очистка использованной памяти. Также стоит учесть такие ошибки, как некорректный ввод логина и пароля, ввод простого пароля, попытка ввода уже существующего логина. Такие моменты стоит обязательно обрабатывать для получения программы устойчивой к неправильным действиям.

На рисунке 3.1 представлена обработка ситуации ввода несуществующего пункта меню.

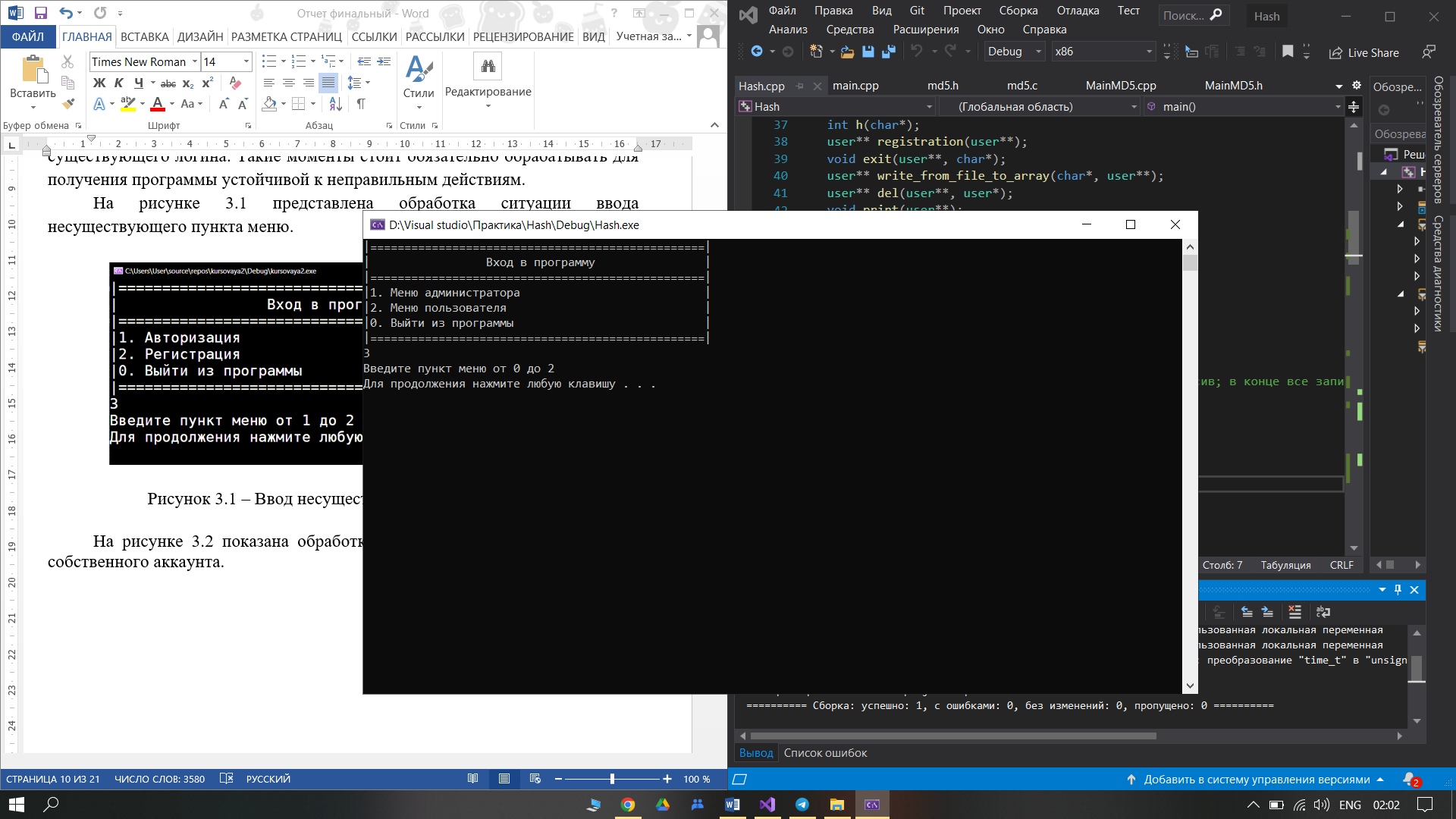


Рисунок 4.1 – Ввод несуществующего пункта меню

На рисунке 3.3 представлена обработка ситуации при добавлении нового аккаунта с логином, который уже занесен в базу данных.

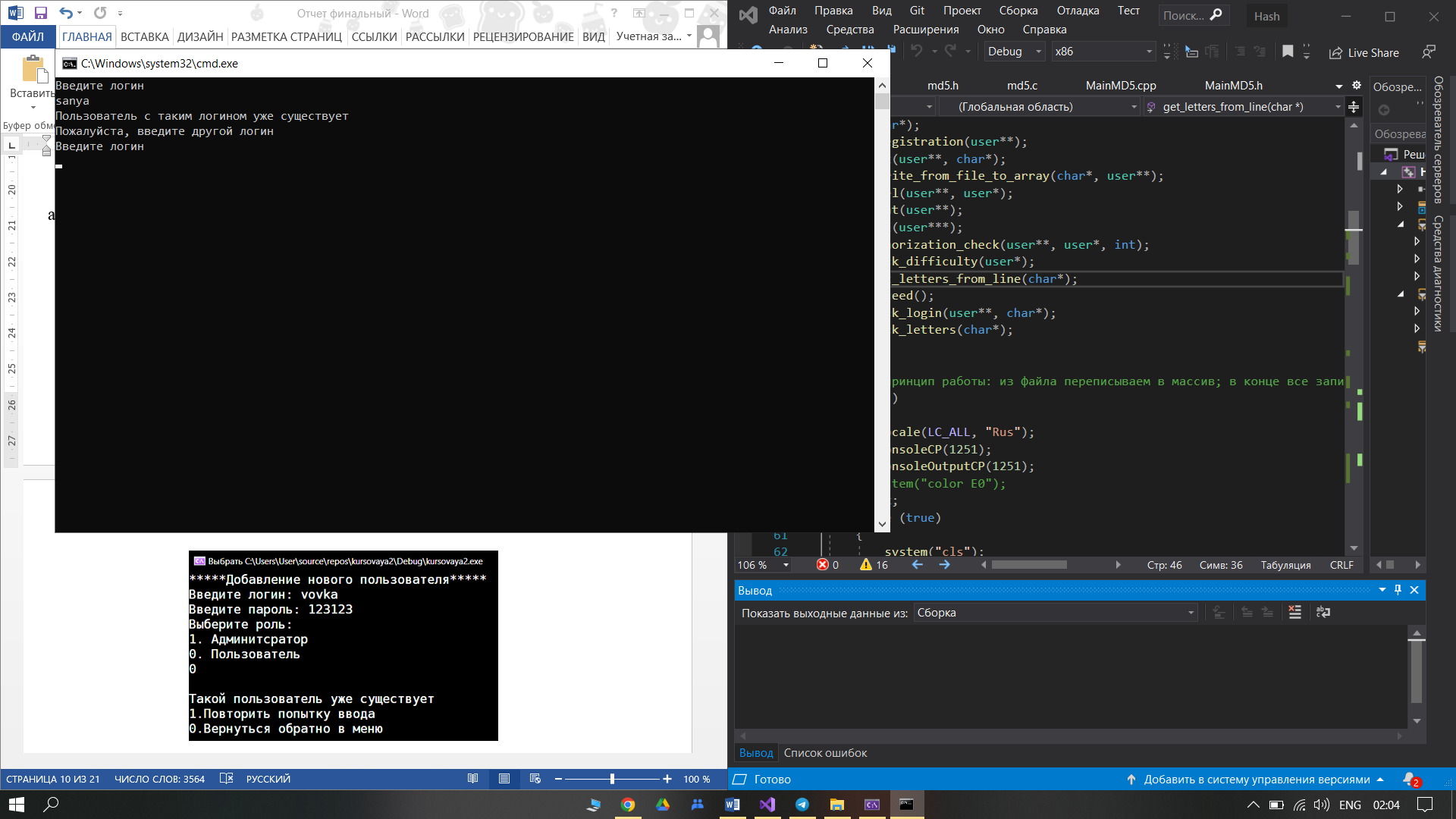


Рисунок 4.2 – Добавление нового аккаунта с логином, который уже занесен в базу данных

На рисунке 4.3 представлена обработка ситуации при некорректном вводе логина.

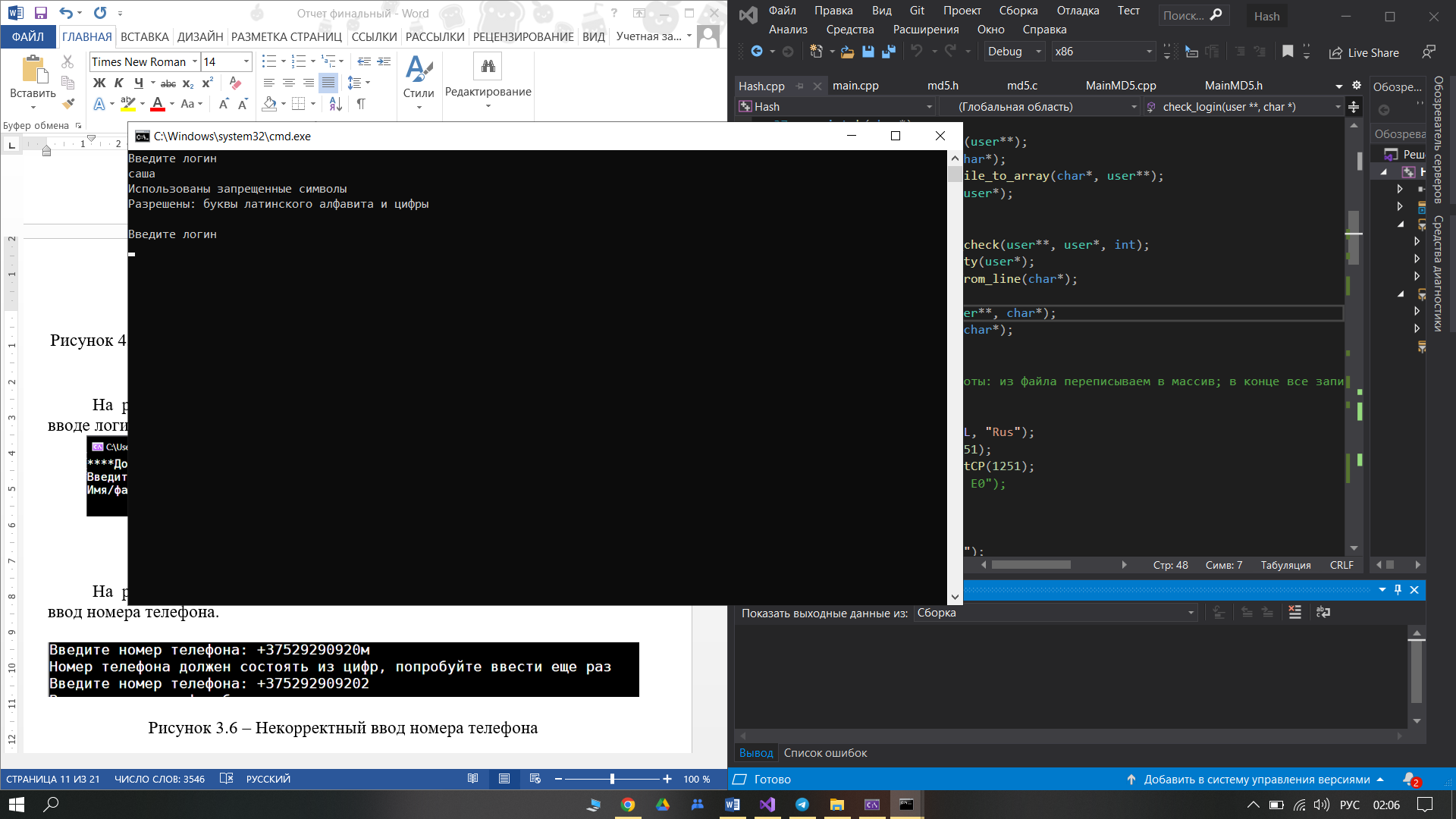


Рисунок 4.3 – Некорректный ввод логина

На рисунке 4.4 представлена обработка ситуации при вводе простого пароля.

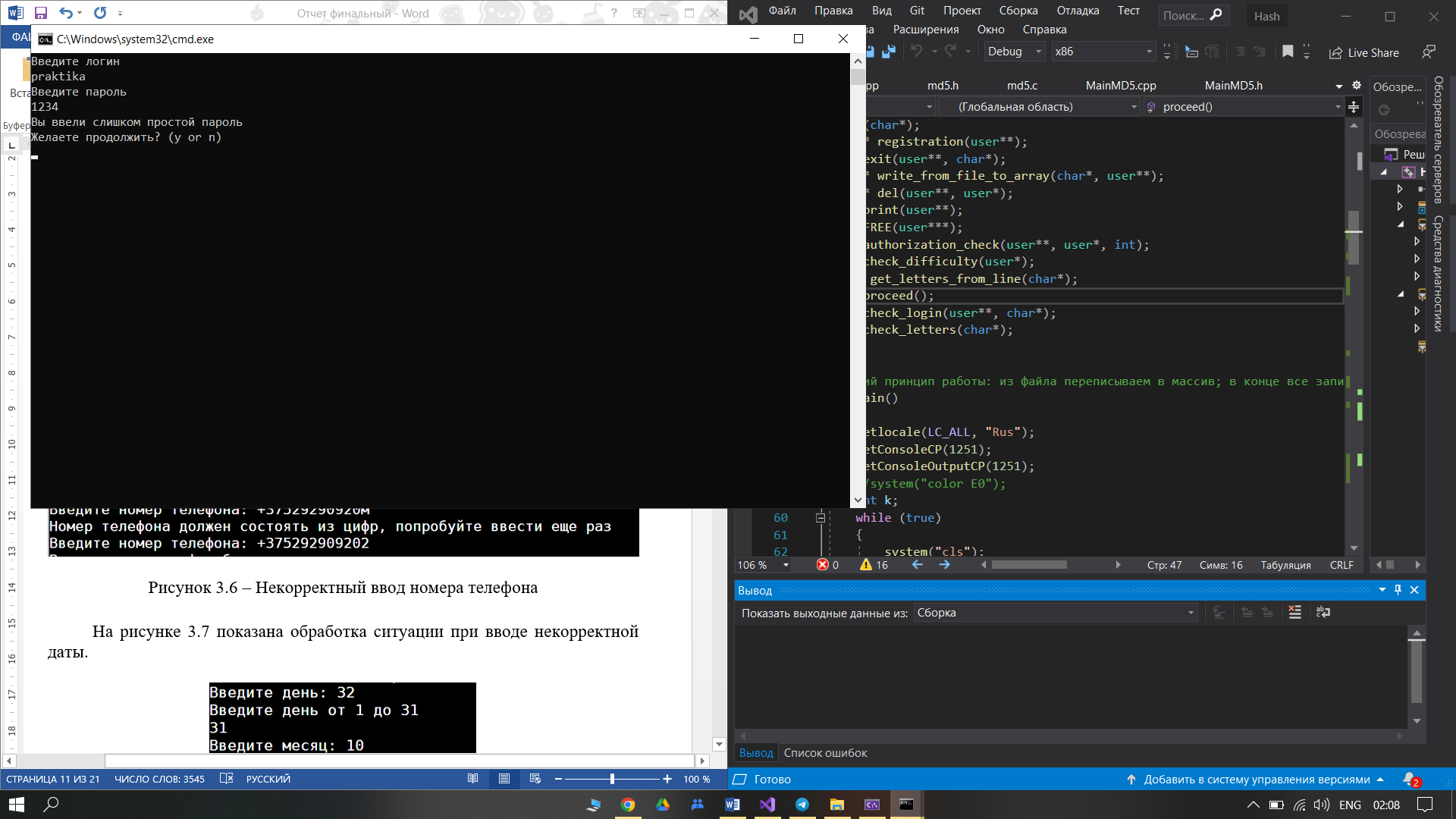


Рисунок 4.4 – Ввод простого пароля

На рисунке 4.5 показана обработка ситуации при вводе некорректного пароля.

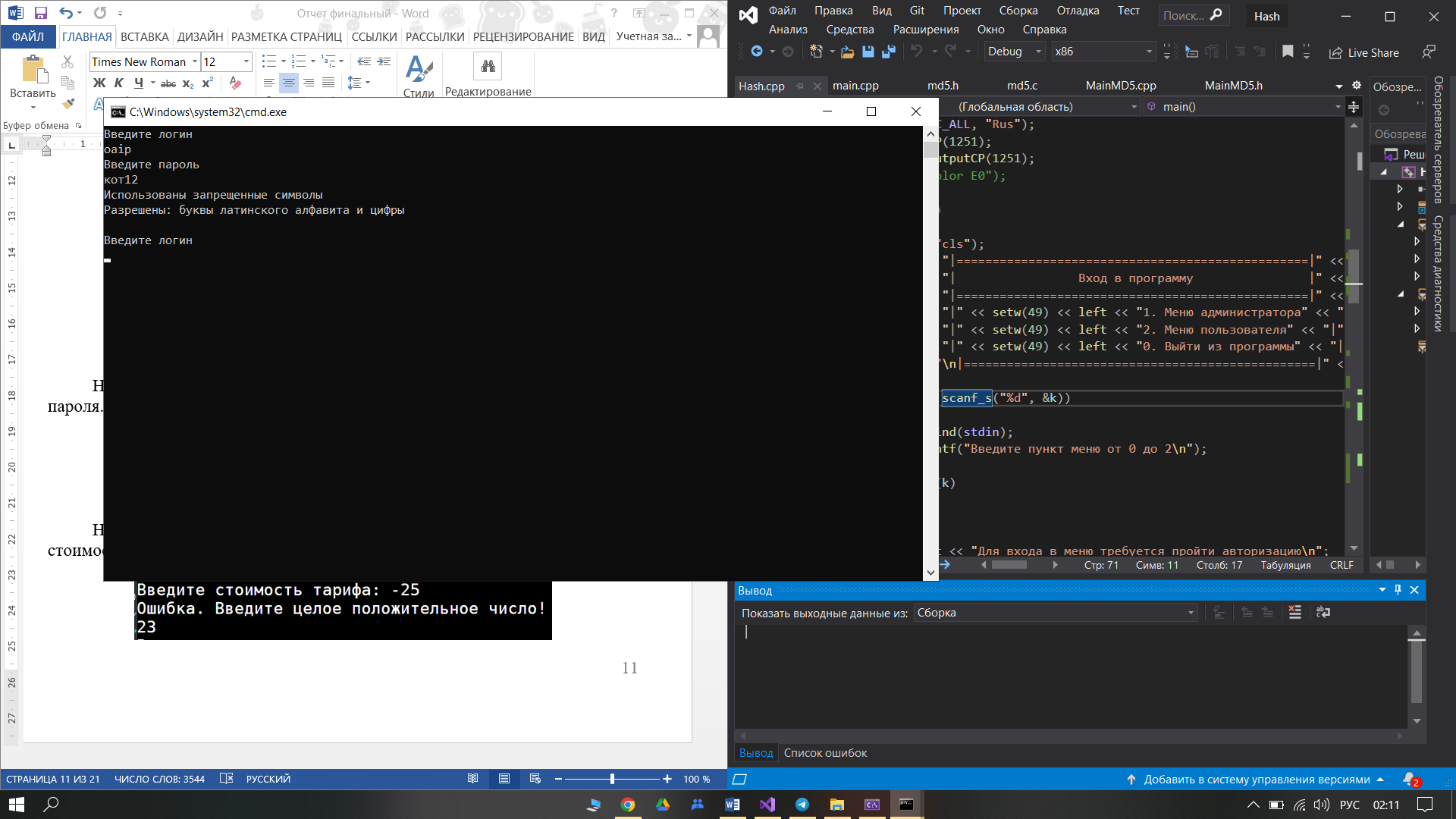


Рисунок 4.5 – Некорректный ввод пароля

На рисунке 4.6 показано удаление несуществующего пользователя.

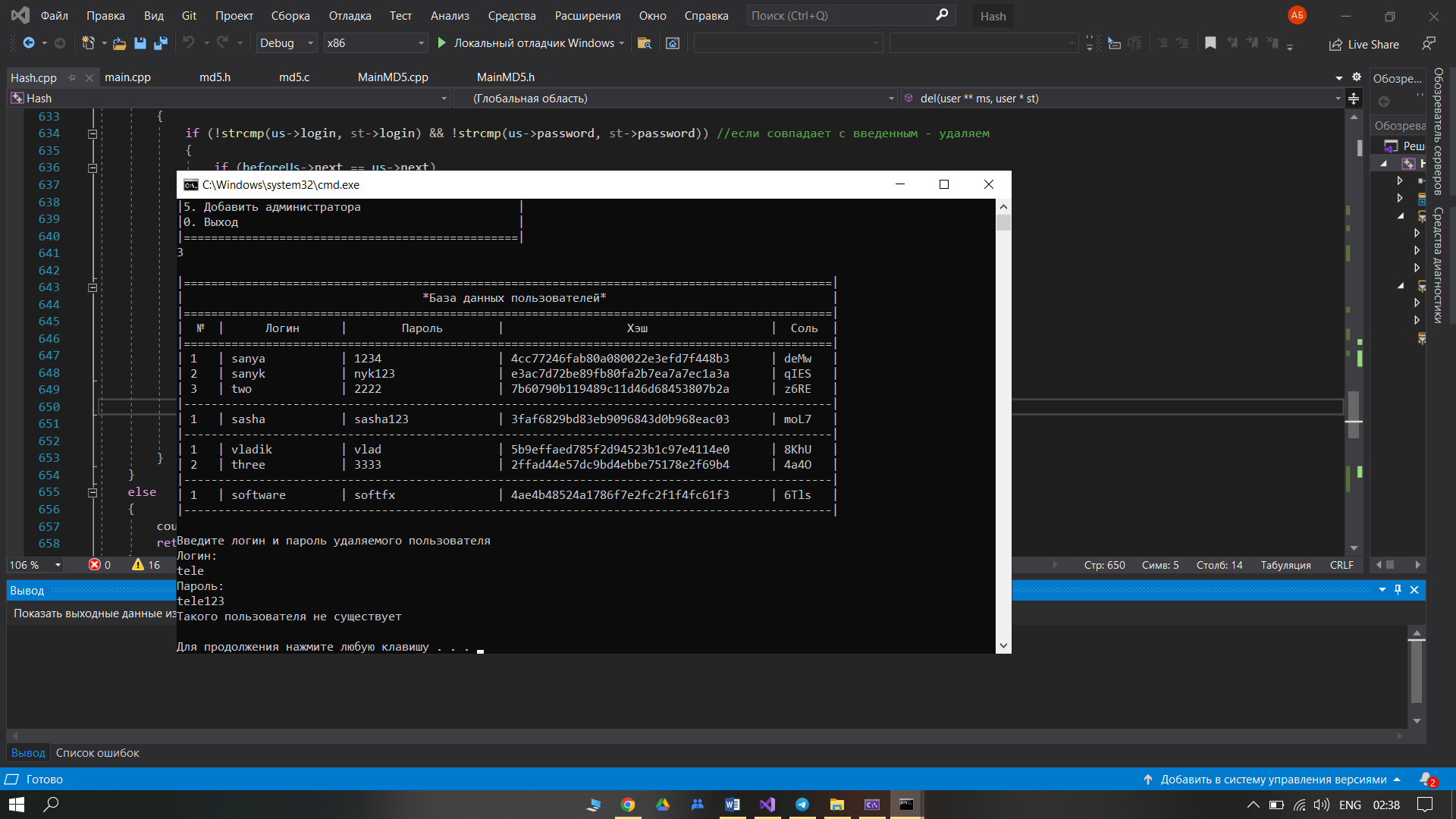


Рисунок 4.6 – Удаление несуществующего пользователя

Таким образом программа будет защищена от неправильных действий пользователем и от некорректного ввода данных, что обеспечит стабильную работу программы.

## **ИНСТРУКЦИЯ ПО РАЗВЕРТЫВАНИЮ ПРИЛОЖЕНИЯ И СКВОЗНОЙ ТЕСТОВЫЙ ПРИМЕР**

После входа в программу отображается меню (рис. 5.1), которое состоит из 3 пунктов: «Меню администратора», «Меню пользователя» и «Выйти из программы».

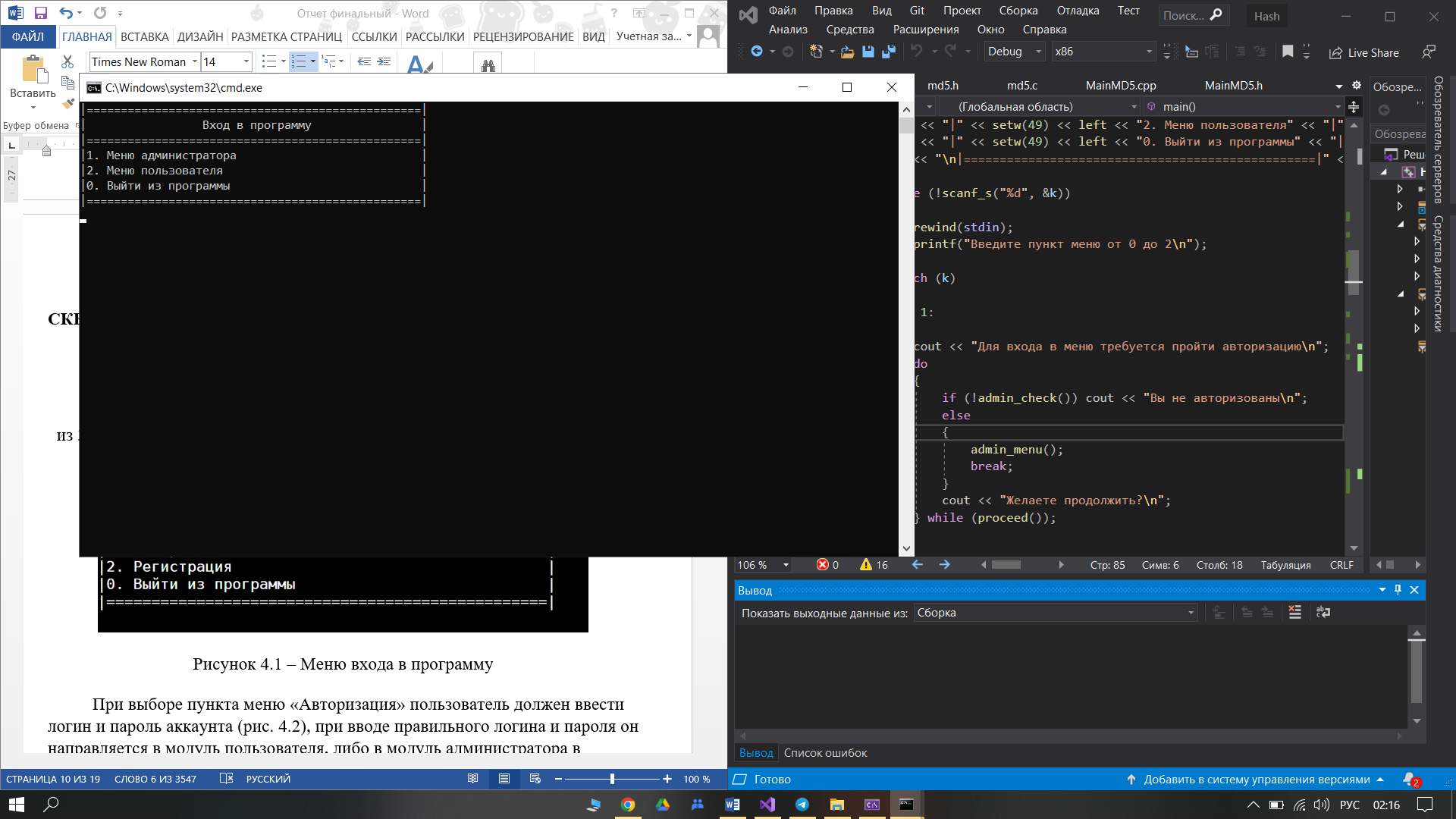


Рисунок 5.1 – Меню входа в программу

**5.1 Модуль администратора**

При выборе пункта «Меню администратора» пользователь должен ввести логин и пароль аккаунта (рис. 5.2), при вводе правильного логина и пароля он направляется в модуль администратора.

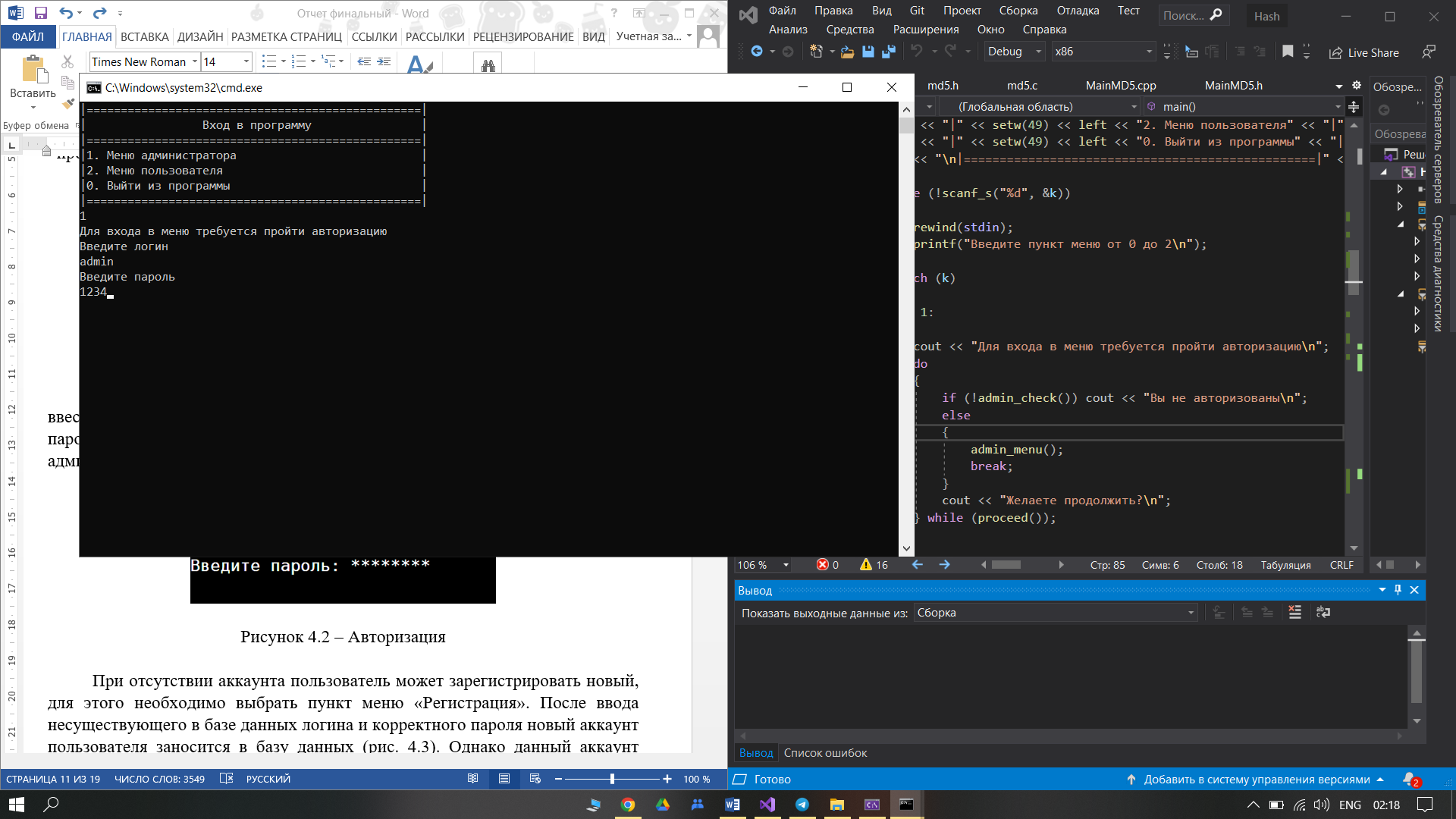


Рисунок 5.2 – Авторизация

Далее появляется окно с доступными функциями для администратора (рис 5.3).

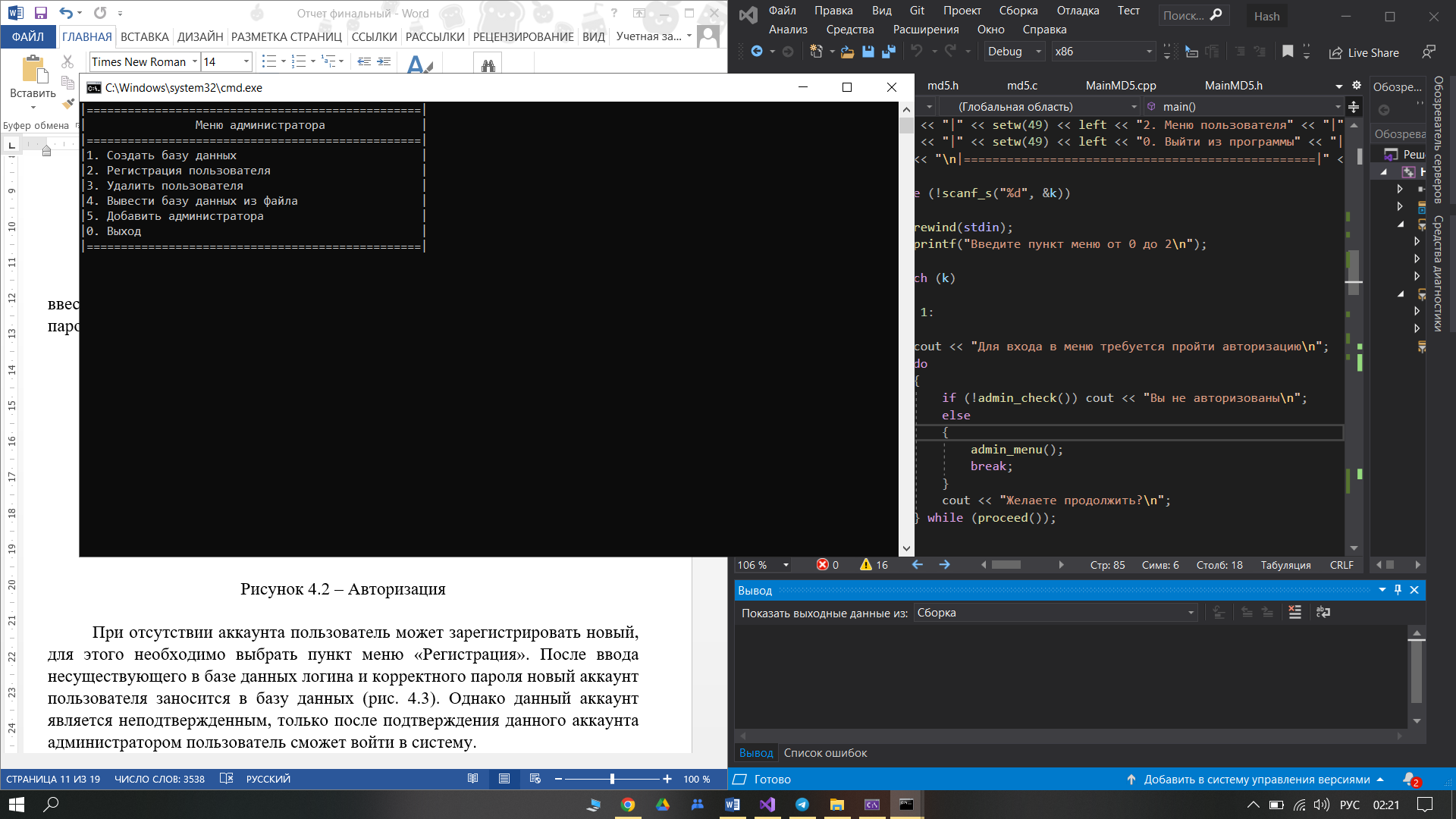


Рисунок 5.3 – Меню администартора

При выборе пункта «Вывести базу данных из файла» на экран выводится таблица со всеми зарегистрированными пользователями (рис. 5.4).

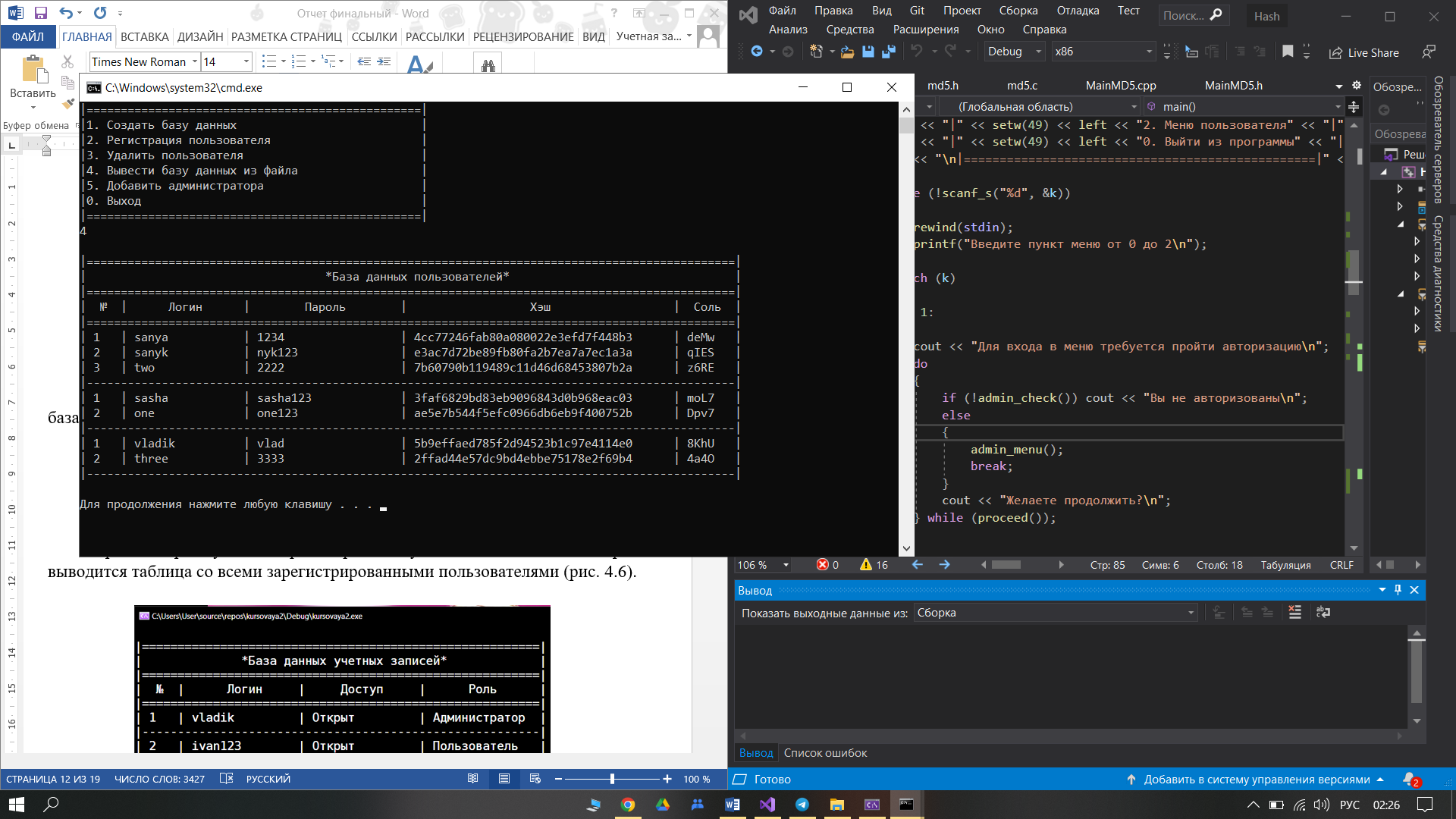


Рисунок 5.4 – База данных пользователей

На рисунке 5.5 показано добавление новой учетной записи.

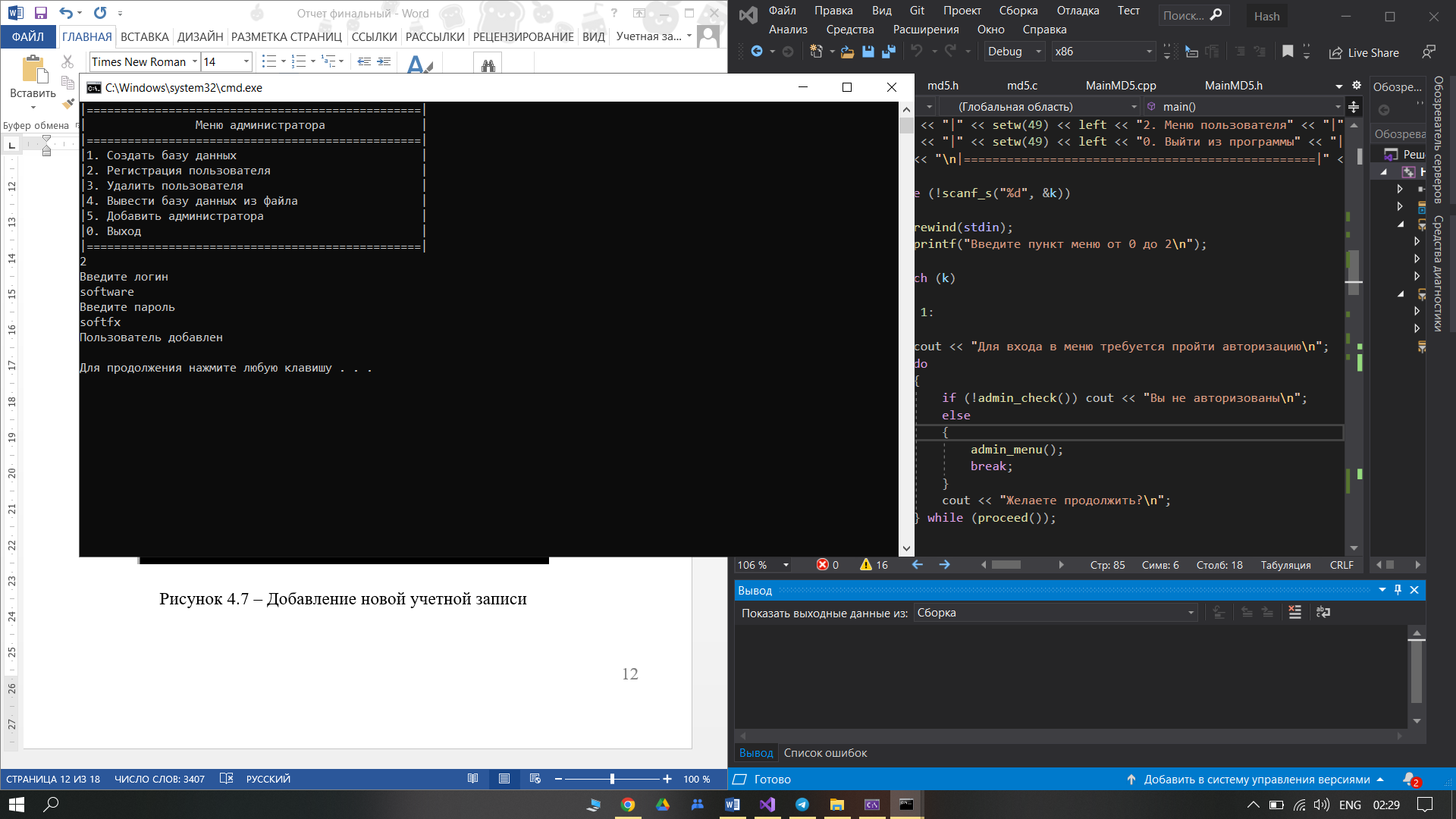


Рисунок 5.5 – Добавление новой учетной записи

При выборе пункта «Удаление учетных записей» на экран выводится таблица со всеми зарегистрированными пользователями, после чего администратор выбирает учетную запись, которую хочет удалить (рис. 5.6).

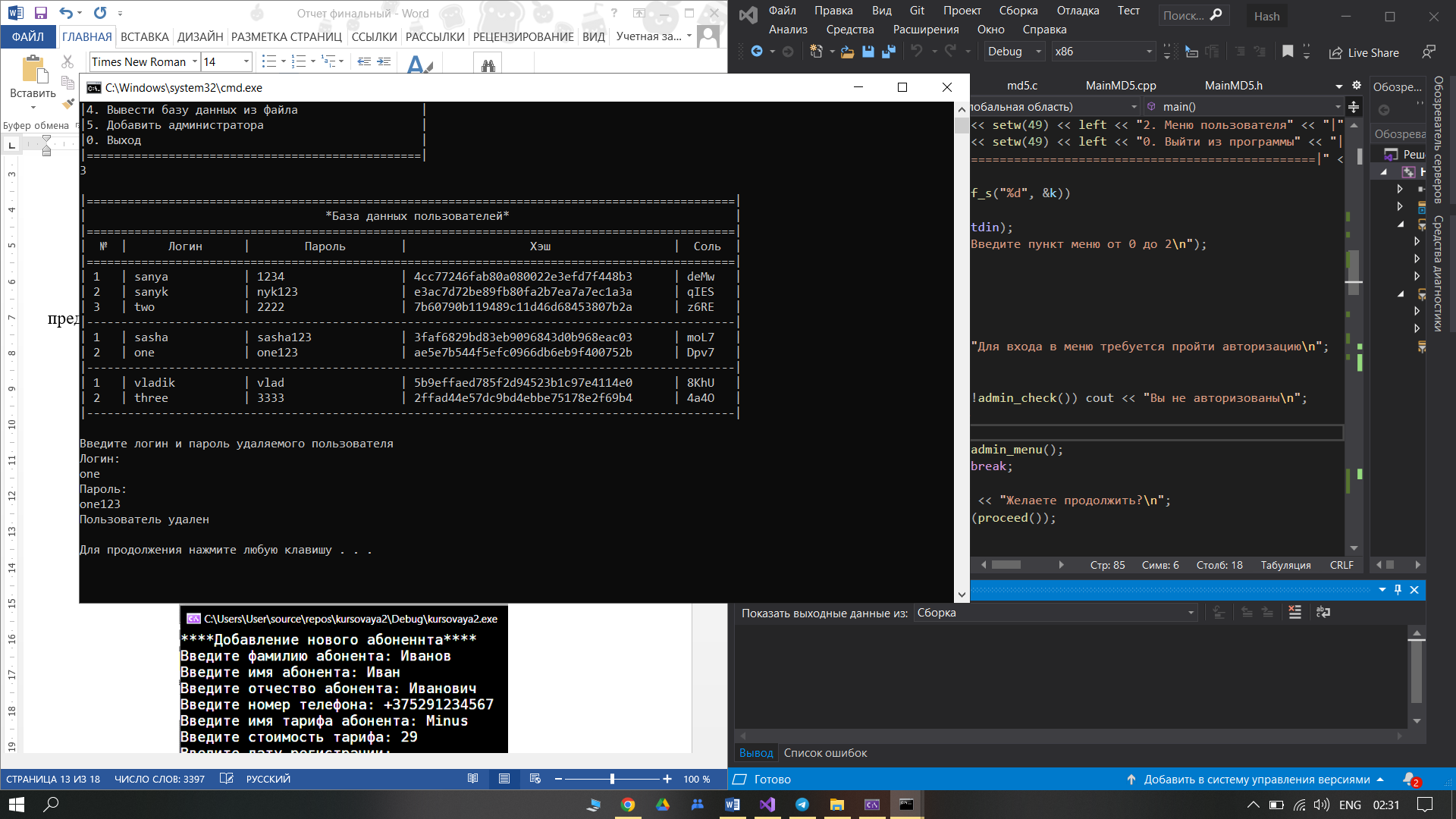


Рисунок 5.6 – Удаление учетной записи

При попытке удаления базы данных пользователей на экран выводится предупреждающее сообщение (рис. 5.7).



Рисунок 5.7 – Подтверждение удаления базы данных

На рисунке 5.8 показано добавление нового администратора.

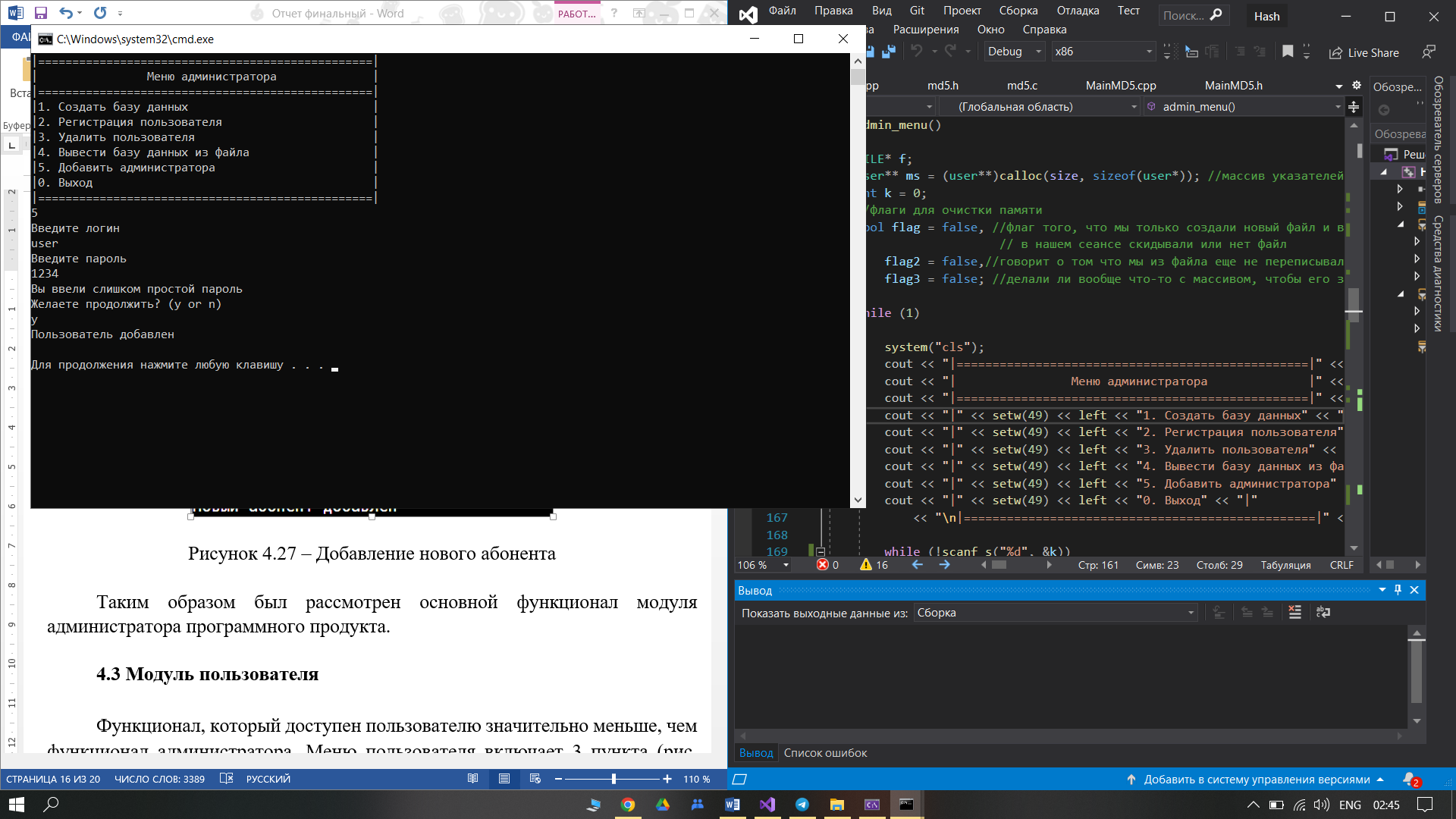


Рисунок 5.8 – Добавление нового администратора

Таким образом был рассмотрен основной функционал модуля администратора программного продукта.

### **5.2 Модуль пользователя**

Функционал, который доступен пользователю значительно меньше, чем функционал администратора. Меню пользователя включает 2 пункта (рис. 5.9): «Проверка авторизации», «Регистрация пользователя», «Выход».

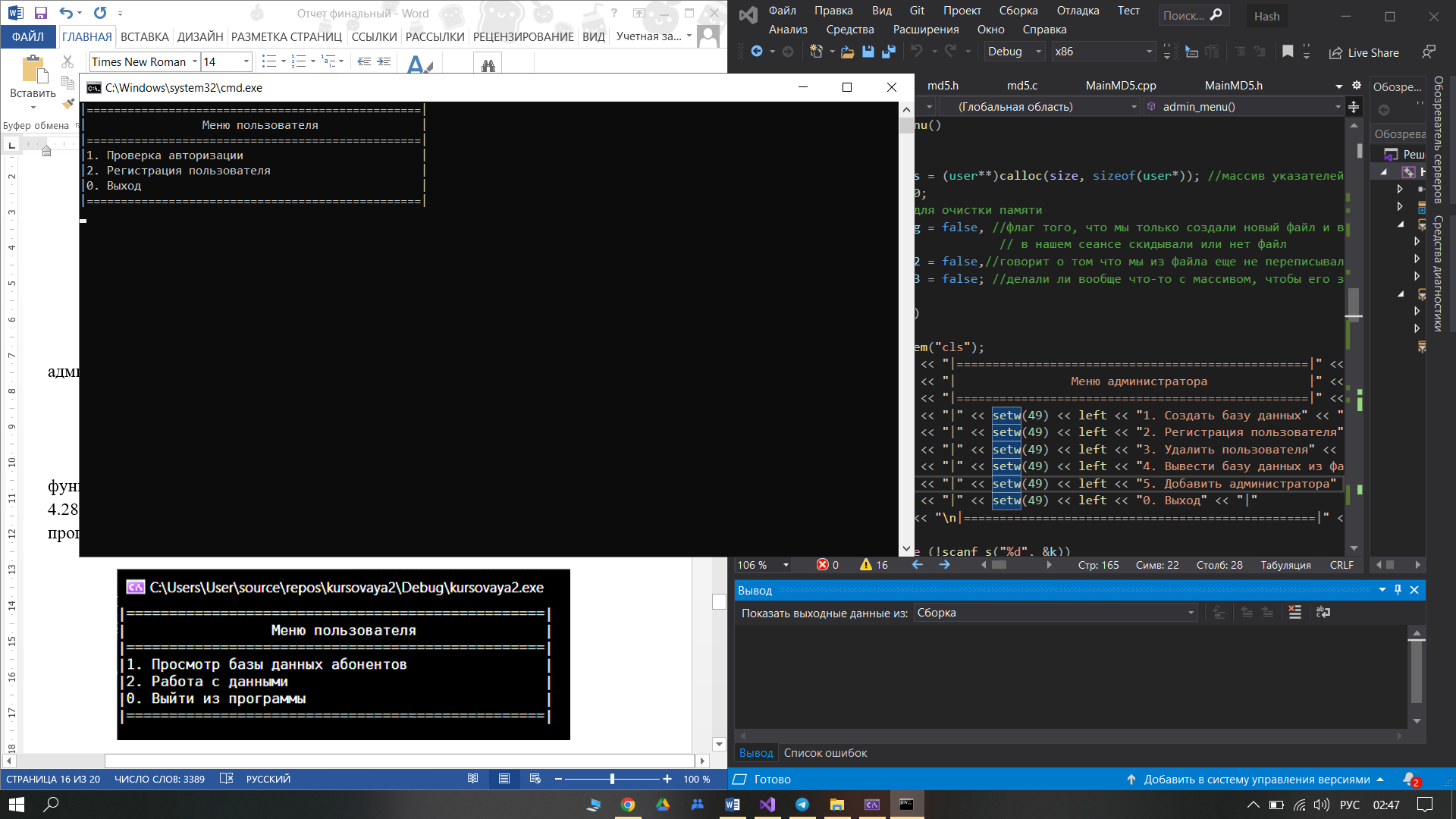


Рисунок 5.9 – Меню пользователя

На рисунке 5.10 показана проверка авторизации пользователя.

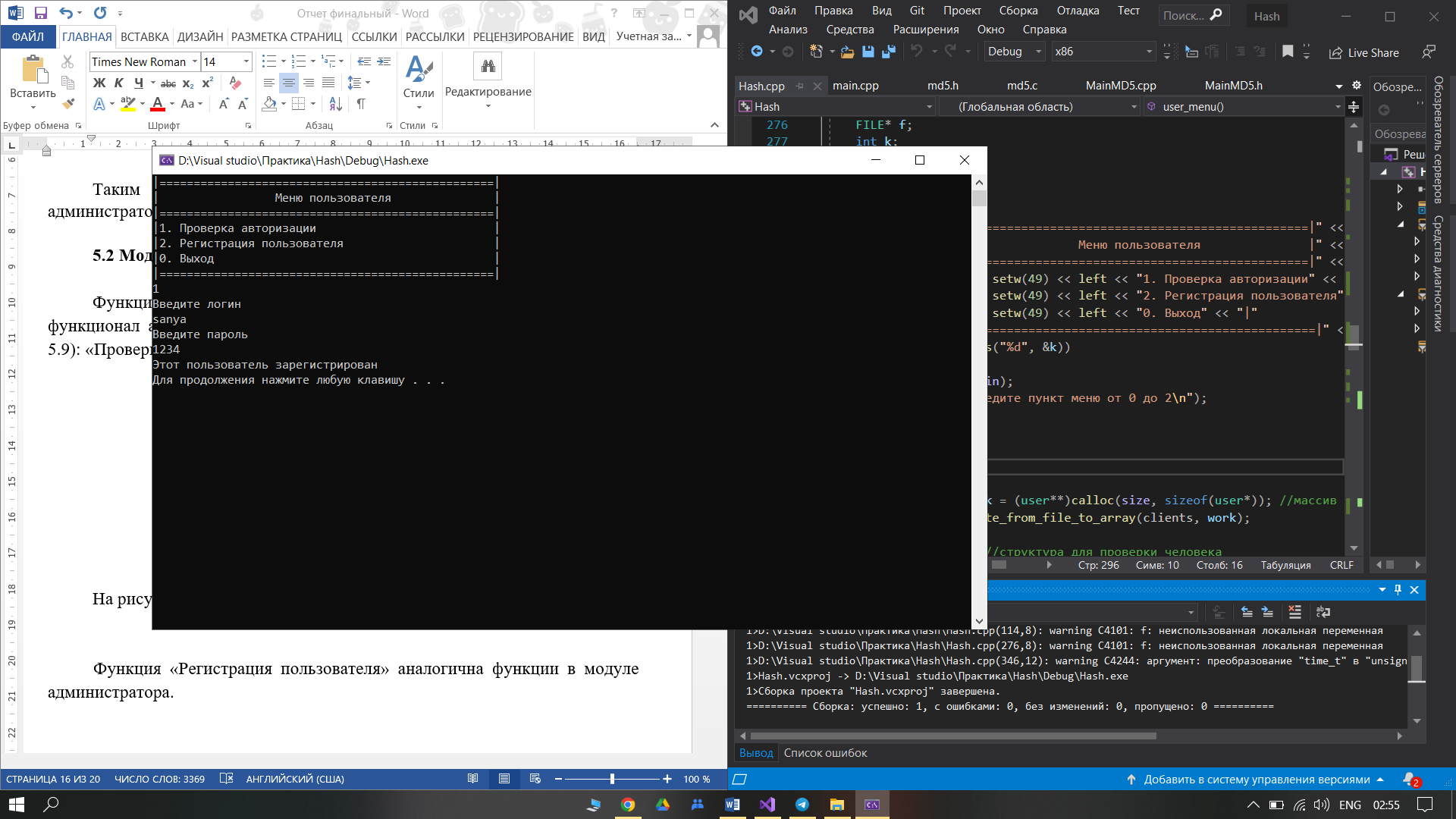


Рисунок 5.10 – Проверка авторизации пользователя

Функция «Регистрация пользователя» аналогична функции в модуле администратора.

## **ЗАКЛЮЧЕНИЕ**

В ходе учебной практики было выяснено, как работать с хэш-таблицами, списками, как правильно хранить данные о пользователя в базе данных, был рассмотрен алгоритм хэширования, а также была проделана работа с файлами и алгоритмом разрешения коллизий. В процессе разработки программного средства было проделано следующее:

* изучена литература по теме проекта;
* разработаны схемы алгоритмов работы программы;
* разработан интерфейс приложения;
* разработаны структуры и модули приложения;
* проведено тестирование программного продукта.

Данное программное средство обладает следующими достоинствами:

* использует мало ресурсов операционной системы;
* быстродействие программы;
* обеспечивает ввод корректных данных.

Для улучшения функционала программы можно реализовать следующее:

* более защищенный алгоритм хэширования;
* сортировки по различным критериям
* функции сохранения данных в случае непредвиденного завершения работы программного средства.

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Хранение паролей пользователей [Электронный ресурс]. – Режим доступа: <https://intsystem.org/security/users-passwords-store/> – Дата доступа: 21.06.2022.

[2] Понятие хеширования [Электронный ресурс]. – Режим доступа: https://studfile.net/preview/5514229/page:8/ – Дата доступа: 21.06.2022.

**ПРИЛОЖЕНИЕ А**

**Листинг кода алгоритмов, реализующих основную логику программы**

**Для функции «hashing»**

void hashing(char\* str, user\* st)

{

srand(time(NULL));

char\* randsol = (char\*)calloc(SALT\_SIZE, sizeof(char));

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

switch (rand() % 3){

case 0:

randsol[i] = rand() % 10 + '0';

break;

case 1:

randsol[i] = rand() % 26 + 'A';

break;

case 2:

randsol[i] = rand() % 26 + 'a';

}

}

randsol[SALT\_SIZE - 1] = '\0';

strcpy(st->sol, randsol);

int lenpassword = strlen(str);

int lenrand = strlen(randsol);

int lengener = strlen(genersol);

char\* line = (char\*)calloc(lenpassword + lenrand + lengener + 1, sizeof(char));

strcpy(line, str);

for (int i = 0; i < lenrand; i++) line[lenpassword +i] = randsol[i];

for (int i = 0; i < lengener; i++) line[lenpassword + lenrand + i] = genersol[i];

line[lenpassword + lenrand + lengener] = '\0';

char\* hash = (char\*)calloc(64, sizeof(char));

MD5(&hash, &line);

strcpy(st->hash, hash);

free(line);

free(randsol);

free(hash);

}

**Для функции «registration»**

user\*\* registration(user\*\* ms)

{

while (1)

{

int sum = 0;

user\* st;

while (1)

{

if (!(st = (user\*)calloc(1, sizeof(user))))

{

puts("ошибка при выделении памяти");

return nullptr;

}

rewind(stdin);

cout << "Введите логин\n";

cin >> st->login;

if (!check\_login(ms, st->login))

{

if (check\_letters(st->login))

{

rewind(stdin);

cout << "Введите пароль\n";

cin >> st->password;

if (check\_letters(st->password))

{

st->next = nullptr;

if (check\_difficulty(st)) break;

}

else

{

cout << "Использованы запрещенные символы\n";

cout << "Разрешены: буквы латинского алфавита и цифры\n\n";

}

}

else {

cout << "Использованы запрещенные символы\n";

cout << "Разрешены: буквы латинского алфавита и цифры\n\n";

}

}

else {

cout << "Пользователь с таким логином уже существует\n";

cout << "Пожалуйста, введите другой логин\n";

}

free(st);

}

sum += h(st->login);

sum += h(st->password);

sum = sum % size;

if (!authorization\_check(ms, st, sum))

{

cout << "Пользователь добавлен\n";

int lenlogin = strlen(st->login);

int lenpass = strlen(st->password);

char\* line = (char\*)calloc(lenlogin + lenpass + 1, sizeof(char));

for (int i = 0; i < lenlogin; i++) line[i] = st->login[i];

for (int i = 0; i < lenpass; i++) line[i + lenlogin] = st->password[i];

line[lenlogin + lenpass] = '\0';

hashing(line, st);

free(line);

user\* work;

if (ms[sum])

{

work = ms[sum];

while (work->next)

work = work->next;

work->next = st;

}

else ms[sum] = st;

return ms;

}

else cout << "Такой пользователь уже существует\nПоменяйте логин или пароль\n\n";

free(st);

}

}

**Для функции «authorization\_check»**

bool authorization\_check(user\*\* ms, user\* st, int sum)

{

user\* work;

char\* lineofhash = (char\*)calloc(64, sizeof(char));

int lengener = strlen(genersol);

int lenpassword = strlen(st->password);

int lenlogin = strlen(st->login);

int lenrand = SALT\_SIZE;

char\* line = (char\*)calloc(lenpassword + lenlogin + lenrand + lengener + 1, sizeof(char));

for (int i = 0; i < lenlogin; i++) line[i] = st->login[i];

for (int i = 0; i < lenpassword; i++) line[i + lenlogin] = st->password[i];

if (ms[sum])

{

work = ms[sum];

while (work)

{

for (int i = 0; i < lenrand; i++) line[lenpassword + lenlogin + i] = work->sol[i];

for (int i = 0; i < lengener; i++)

line[lenpassword + lenlogin + lenrand - 1 + i] = genersol[i];

line[lenpassword + lenlogin + lenrand + lengener] = '\0';

MD5(&lineofhash, &line

if (!strcmp(work->hash, lineofhash))

return true;

else work = work->next;

}

return false;

}

else return false;

}