МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Университет ИТМО

Факультет систем управления и робототехники

ЛАБОТАТОРНАЯ РАБОТА № 1

по дисциплине «Практическая линейная алгебра»

Выполнила:

Студентка группы R3281

Троицкая Тамара Андреевна

Преподаватель:

Перегудин Алексей Алексеевич

Санкт-Петербург, 2023

# Содержание

1. [Intro](#_Intro)
2. [Task 1](#_Task_1)
3. [stucts.h](#_structs.h)
4. [structs.c](#_structs.c)
5. [Task1.c](#_Task1.c)
6. [Анализ результатов работы программы](#_Анализ_результатов_работы)
7. [Вывод](#_Вывод)
8. [Task 2](#_Task_2)
9. [Task 3](#_Task_3)
10. Task 4

# Intro

Приветствую тебя, уважаемый читатель моего отчёта. Он обещает быть обширным, так что перед началом я посоветовала бы тебе заварить кофе или чего покрепче.

Код для лабы я писала на Си, поэтому написала вручную все структуры и функции, потребовавшиеся для работы с матрицами и векторами. Я постаралась написать комментарии к коду максимально подробно, чтобы всё было понятно даже человеку, никогда не писавшему на Си. Можете просто читать комментарии перед функциями, чтобы понять, что они делают, но реализацию я тоже приложу.

Можете ориентироваться в отчёте, переходя по ссылкам в содержании. Под каждым из заданий перечислены файлы, по которым распределено решение. Если кликнуть на название файла, вы перейдёте к описанию реализованных там функций.

# Task 1

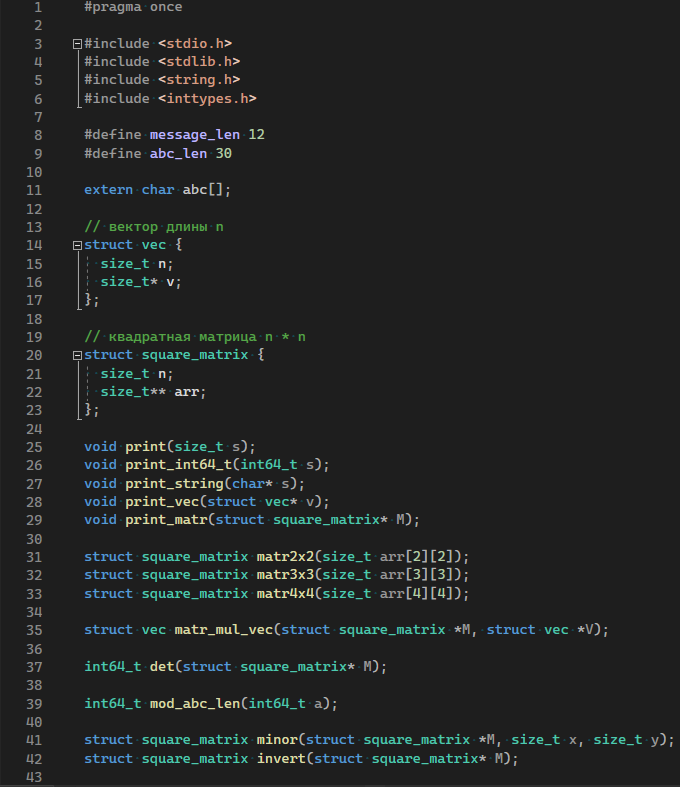
Начнём стандартно с задачи №1, в которой нужно было реализовать шифр Хилла. Говоря коротко, нужно было:

1. Придумать квадратную матрицу-ключ
2. Перевести сообщение из 12 букв в массив из 12 чисел -- номеров этих букв в заданном “алфавите”
3. Разбить массив этих чисел в массив векторов такого же размера, как матрица-ключ
4. Поочерёдно умножать матрицу-ключ на эти векторы, формируя массив векторов результата
5. Сконкатенировать, то есть склеить все векторы в один массив чисел
6. Перевести этот массив чисел обратно в буквы

## structs.h

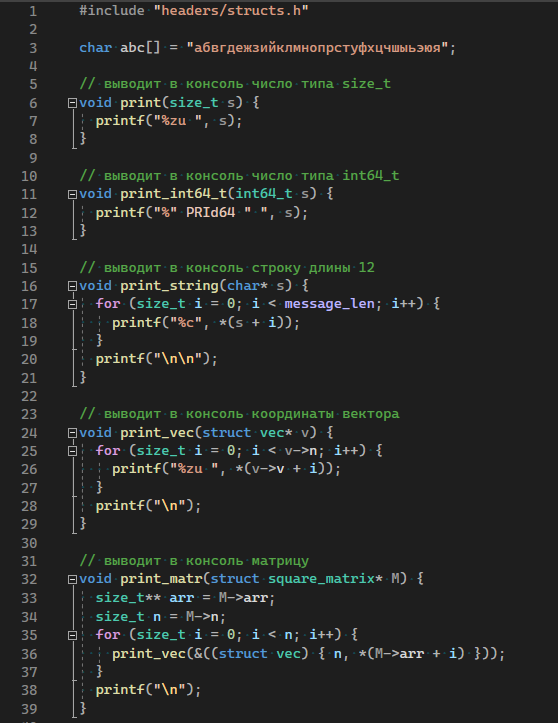
Теперь, когда идея понятна, перейдём к коду. Файл structs.h. В нём реализованы структуры и объявлены функции, реализованные в файле structs.c. Эти функции можно будет использовать в любом файле, в начале которого написано #include “structs.h”. Это именно те вспомогательные функции для работы с матрицами и векторами, о которых я говорила вначале. Скоро рассмотрим действие каждой из них более подробно.

Я написала структуры матрицы и вектора, чтобы можно было в явном виде задавать и использовать размеры матрицы и вектора. Это было бы труднее и не так безопасно, если бы мы использовали просто указатели. Также при моём подходе можно создавать указатель на матрицу и массив векторов, не путаясь в количестве звёздочек. Также я называла функции так, чтоб их название максимально понятно показывало, что делает данная функция.

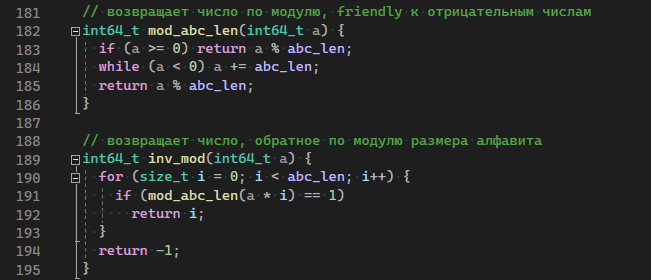


## structs.c

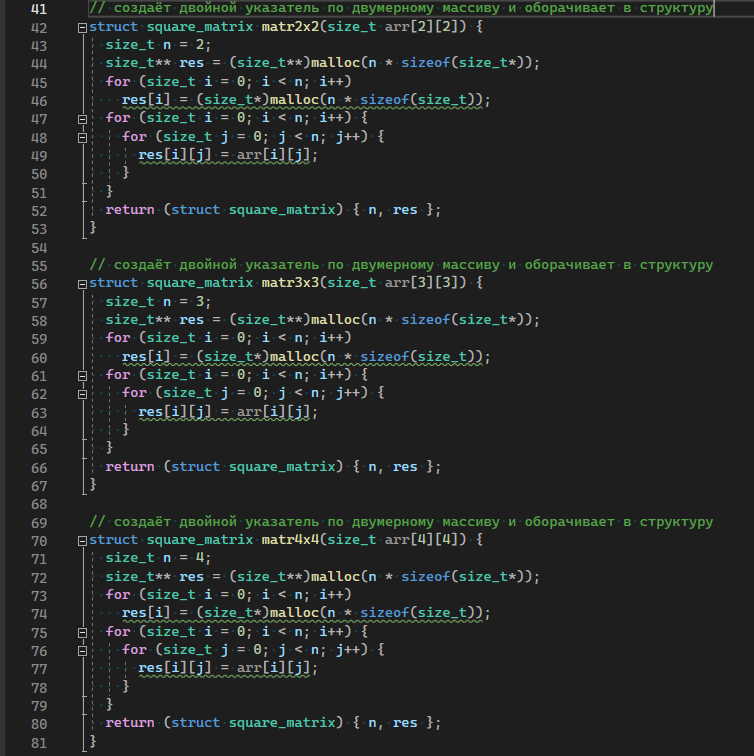
Перейдём к рассмотрению реализации функций. Файл structs.c. В начале файла -- массив используемых символов алфавита. Далее первые 5 функций нужны для того, чтобы выводить в консоль разные типы данных. Это пригодится для отладки и просмотра результата.

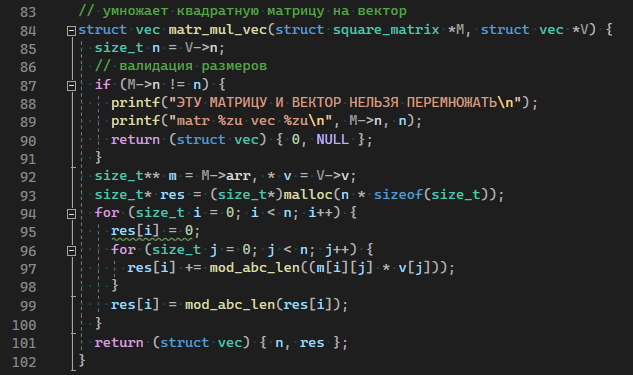


Две функции для работы в кольце (в общем, чтоб искать что-то по модулю)

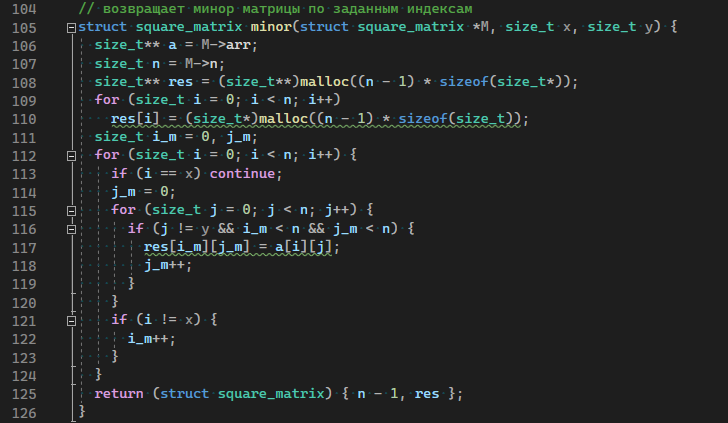


Функции для упрощённой инициализации матриц с помощью двумерного массива:

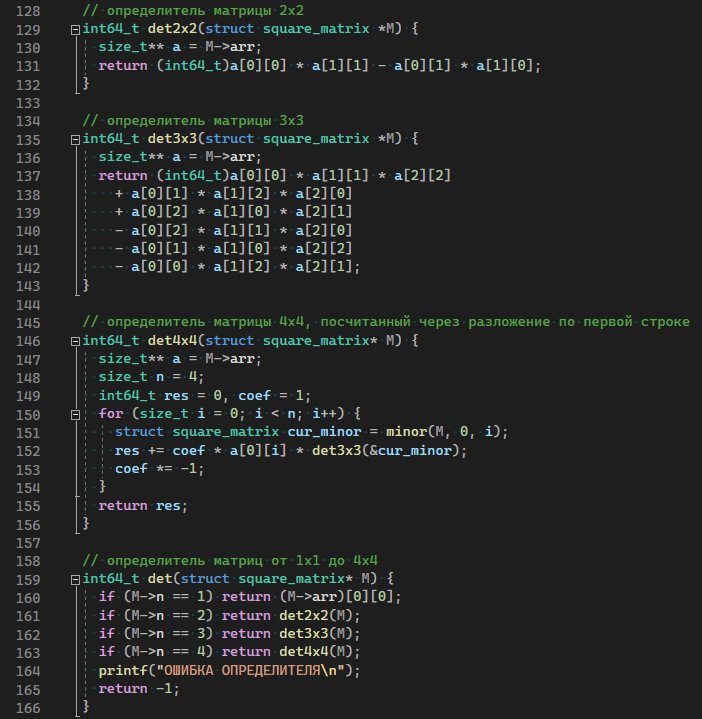




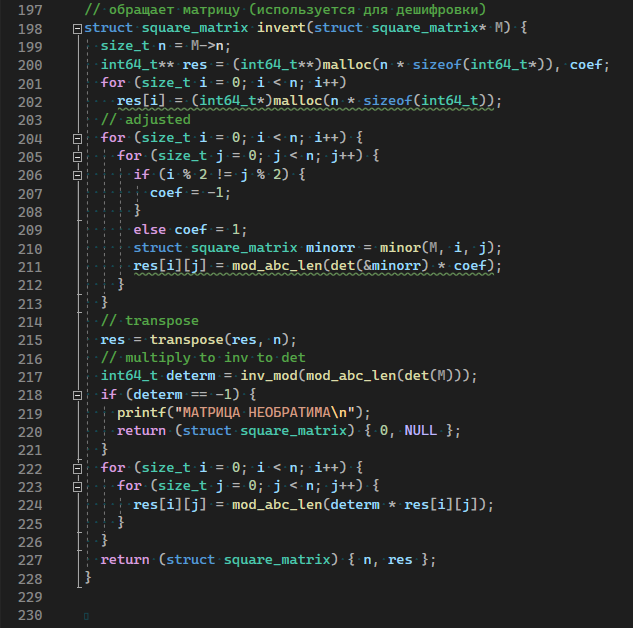
Возвращает минор



Функции для подсчёта определителя (последняя нужна для укороченного вызова первых трёх)



Напомню факт из матричной алгебры. Чтобы посчитать матрицу, обратную данной, надо сначала построить матрицу, состоящую из определителей её миноров, затем сделать из неё матрицу алгебраических дополнений, домножив нужные элементы на -1, после транспонировать её и полученную матрицу домножить на число, обратное дискриминанту, не забывая, что мы в кольце вычетов по модулю:

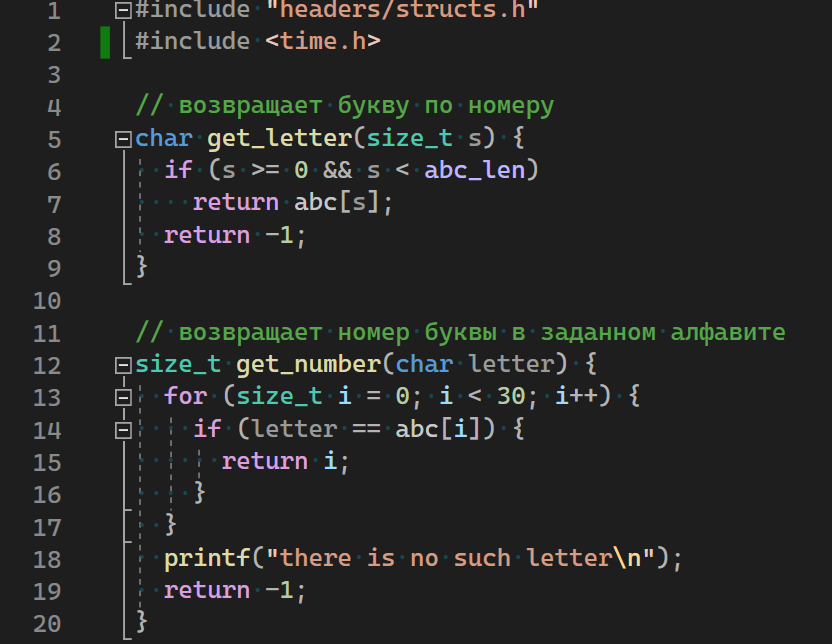


Итак, мы рассмотрели все вспомогательные функции работы с матрицами и векторами. Перейдём непосредственно к шифрованию.

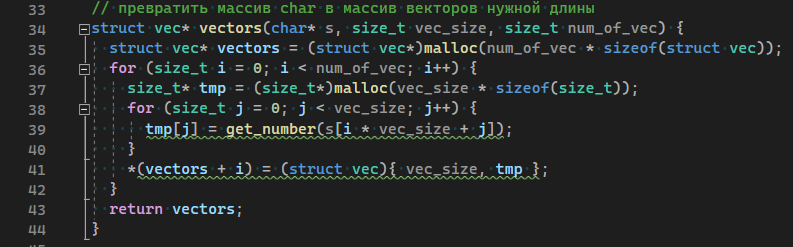
## Task1.c

В начале мы импортировали все написанные нами функции для работы с матрицами. Потом импортировали время, оно нам ещё понадобится.

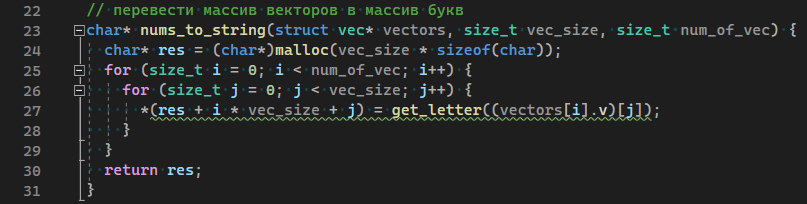
Для начала нам понадобятся функции, превращающие букву в её номер в алфавите и наоборот, номер в букву. Вот эти две функции:



Массив букв одновременно переводит в числа и разбивает на несколько векторов (пункты 2 и 3 исходного плана со страницы 2):



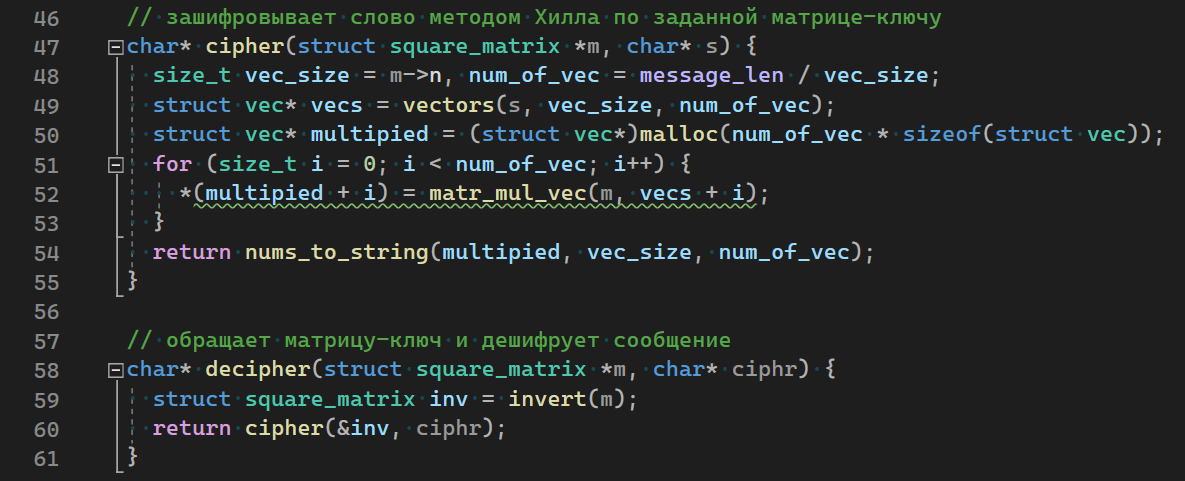
Выполняет действие, обратное к действию предыдущей функции (5 и 6 пункт):



cipher -- полностью выполняет шифрование по плану на стр 2 (строки 50-53 выполняют п.4)

Вызывает в нужном порядке описанные выше функции, объединяет их в единую логику.

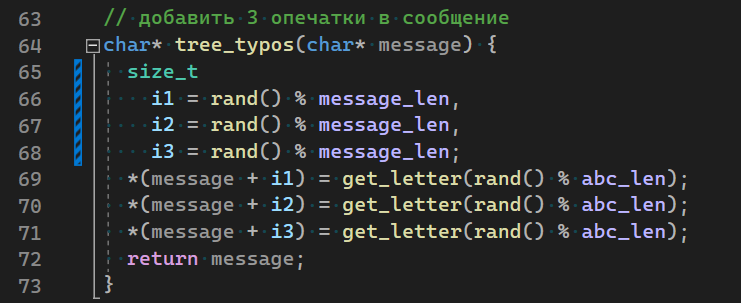
decipher выполняет дешифрование. Как вы можете заметить, в ней происходит обращение матрицы-ключа и вызов функции cipher с полученной матрицей.



Также по заданию нужно было добавить 3 опечатки в каждую из зашифрованных строк.

rand() % 30 даёт целое число в диапазоне [0; 29]

Эта функция изменяет исходное сообщение.



И вот мы наконец добрались до самой главной функции, которая запускает весь проект.

Чтобы рандом был ещё рандомнее, мы делаем srand(clock());

clock() (именно ради него мы подключали <time.h>) возвращает текущее время в некоторых особых единицах, а srand как-то на нём основывает все псевдорандомные числа, генерящиеся в ходе работы программы.

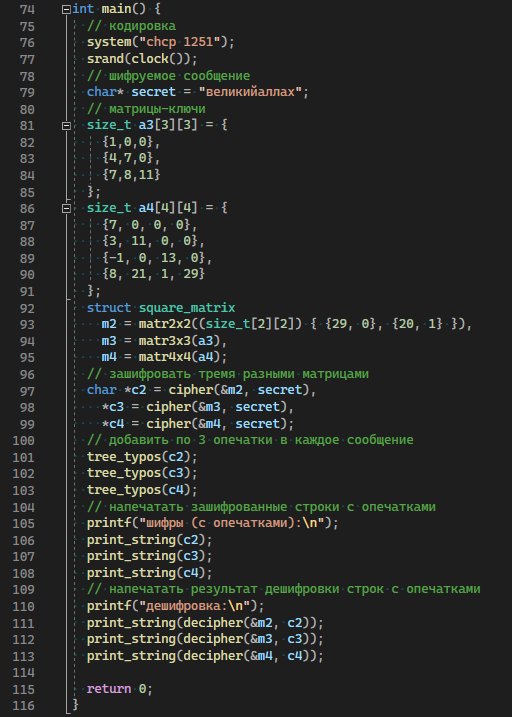
В строке 80 записана шифруемая строка из 12 символов.

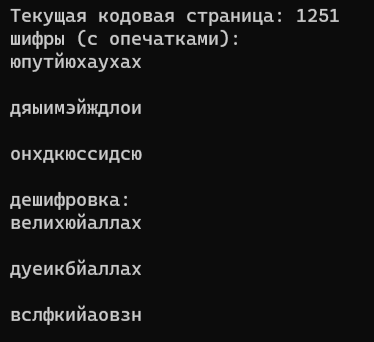
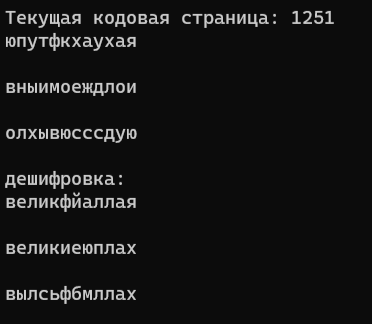
Строки 82-97 задают три матрицы-ключа, с помощью которых производится шифрование. Все их элементы взяты по модулю 30 (30 -- количество букв в заданном алфавите), поэтому они находятся в диапазоне от 0 до 29. Так как они треугольны, то их определитель -- произведение элементов на главной диагонали, очевидно, что 30 и эти определители являются взаимно простыми.

В строках 100-103 мы зашифровали сообщение тремя путями (матрицами m2, m3 и m4), получив массивы символов.

Далее в каждый из массивов этих символов мы добавили по 3 опечатки, то есть поменяли псевдорандомные 3 буквы в зашифрованных сообщениях. (105-108)

И наконец в строках 115-117 мы дешифруем эти сообщения и выводим результат на экран.





## Анализ результатов работы программы

Можно заметить, что 3 опечатки в зашифрованном сообщении могут повлечь за собой большее количество неправильных символов во втором. Почему это происходит? Ответ на вопрос становится очевидным если вспомнить, как вычисляется умножение матрицы на вектор. Строка матрицы “умножается” на столбец вектора. То есть если вектор неправильный, это порождает ошибку в худшем количестве векторов, равном количеству строк матрицы. В некоторых случаях буква может оказаться неошибочной, особенно при небольшом алфавите, так как все числа берутся по модулю числа символов алфавита.

## Вывод

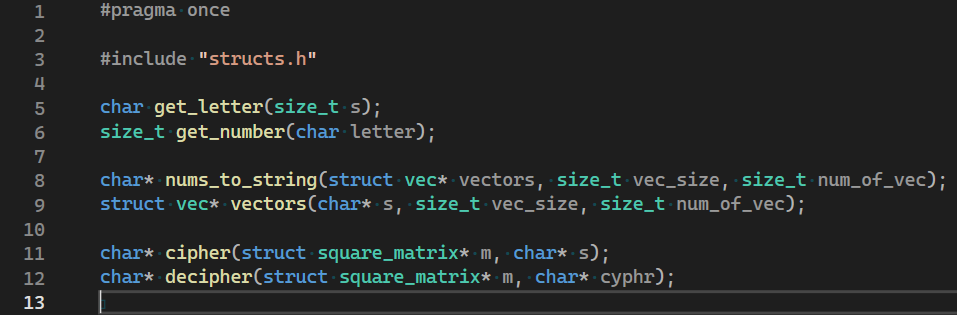
Шифр Хилла уязвим при изменениях зашифрованного сообщения. То есть если мы немного поменяем шифр, это довольно сильно скажется на расшифрованном сообщении.

Интересно, дочитал ли кто-нибудь до этой строчки...

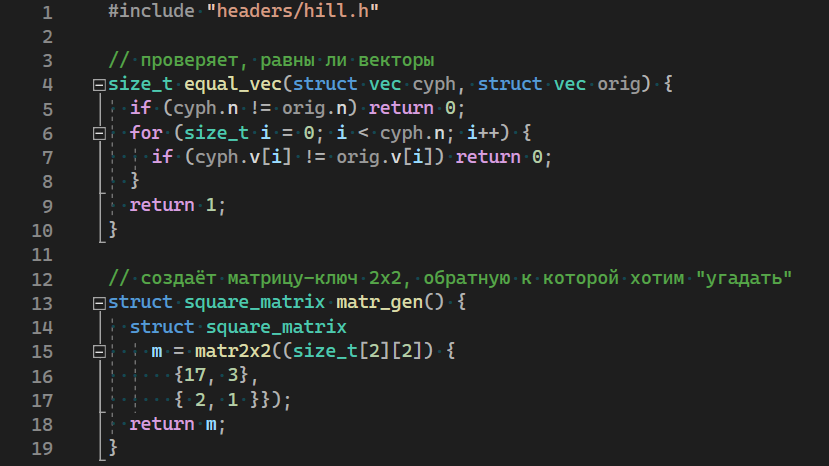
# Task 2

Второе задание. Мы хакеры. У какого-то пользователя было 2 сообщения. Он их зашифровал одной матрицей-ключом. Нам в руки попало 2 зашифрованных сообщения и расшифровка первого сообщения. Матрица-ключ нам неизвестна, и по сути её нам надо найти. Точнее сказать, нам нужно найти матрицу, обратную к матрице-ключу, чтобы с помощью её расшифровать потом второе сообщение пользователя. То есть наша задача -- расшифровать первое сообщение пользователя.

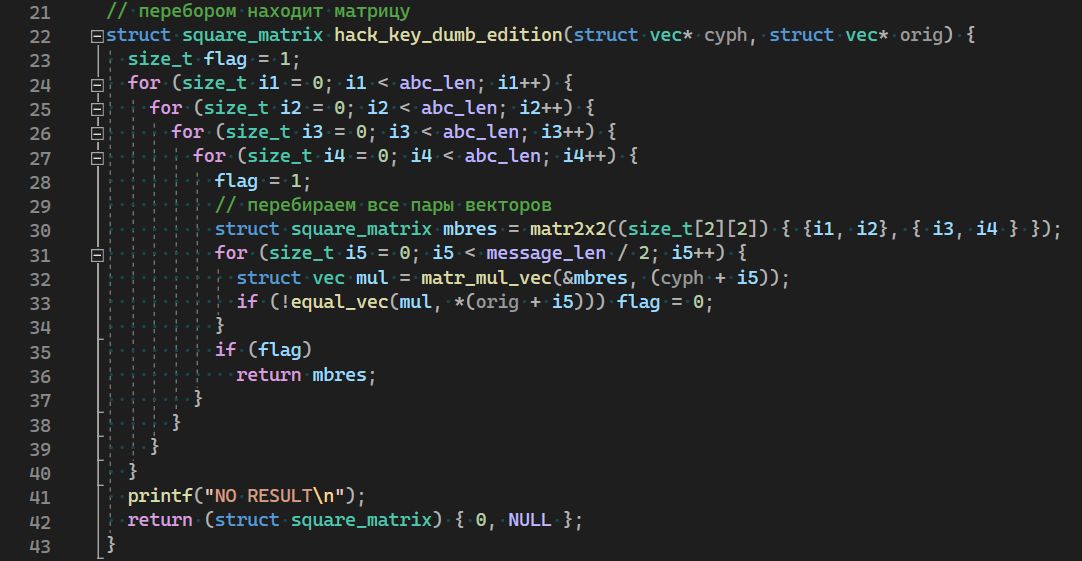
Для второй задачи я создала другой файл (Task2.c), и теперь чтобы использовать функции, реализованные в файле Task1.c, надо создать заголовочный файл и занести в него все функции, которые мы собираемся использовать. (hill.h)



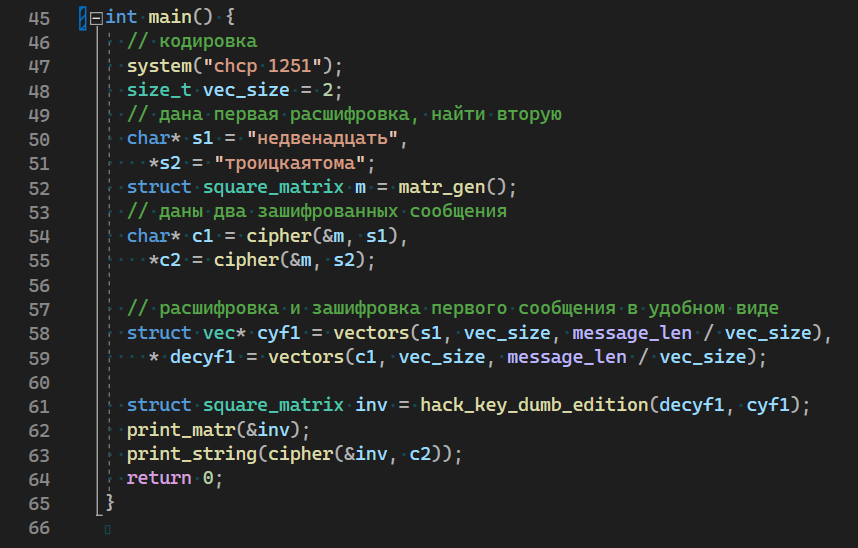
Вторая функция создаёт матрицу-ключ. Здесь это действие не автоматизировано, но так как мы вынесли это в отдельную функцию, в дальнейшем можно будет заменить алгоритм этого более низкого уровня, оставив без изменений высокий уровень.



Эта функция перебирает все 810 тысяч вариантов матрицы (по 30 вариантов на каждый элемент матрицы) и потом проверяет, верный ли результат она даёт при умножении её на каждый из векторов массива:



Функция main. Думаю, её действие довольно очевидно из комментариев.



# Task 3