Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 4

Выполнила студентка группы КС-30 (Ноль Эльвира Гарриевна)

Ссылка на репозиторий: (https://github.com/Elviranng/Nol-Elvira-KS-30/blob/master/lab4sem2)

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи: 05.06.21

Оглавление

[Описание задачи. 2](#_Toc71231795)

[Описание алгоритма. 2](#_Toc71231796)

[Выполнение задачи. 3](#_Toc71231797)

[Заключение. 14](#_Toc71231798)

# Описание задачи.

В рамках лабораторной работы необходимо реализовать 1 из ниже приведенных алгоритмов хеширования:

SHA2

После завершения реализации провести следующие тесты

* сравнить результат работы созданной функции с библиотечной реализацией на 10 произвольных строк произвольной длинны, сравнение можно провести по заранее заданным строкам и заранее вычислинным хешам.

В качестве задачи со звёздочкой сравнение можно производить библиотечной функцией из подключенных библиотек

* Провести проверку и построить зависимости скорости расчета хеша в зависимости от размера входных данных для строк длинной (32, 64, 128, 256, 512)

# Описание алгоритма.

SHA-2 (англ. Secure Hash Algorithm Version 2 — безопасный алгоритм хеширования, версия 2) — семейство криптографических алгоритмов — однонаправленных хеш-функций, включающее в себя алгоритмы SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/256 и SHA-512/224.

Хеш-функции семейства SHA-2 построены на основе структуры Меркла — Дамгора.

Исходное сообщение после дополнения разбивается на блоки, каждый блок — на 16 слов. Алгоритм пропускает каждый блок сообщения через цикл с 64 или 80 итерациями (раундами). На каждой итерации 2 слова преобразуются, функцию преобразования задают остальные слова. Результаты обработки каждого блока складываются, сумма является значением хеш-функции. Тем не менее, инициализация внутреннего состояния производится результатом обработки предыдущего блока. Поэтому независимо обрабатывать блоки и складывать результаты нельзя.

В алгоритме SHA-256 итоговая хеш-сумма равняется 256 битам.

# Выполнение задачи.

Реализация SHA2 в данном случае наиболее удобна в виде класса и дружественной функции. В классе – все необходимые начальные и промежуточные значения, методы обработки сообщения (в том числе раунды). Дружественная функция – создает объект класса, вызывает из него методы в соответствующем порядке, где подготавливает строку для передачи в методы, и результаты методов для передачи на выход из функции.

В лабораторной работе использован наиболее популярный вариант SHA2 – SHA256, в котором итоговый хеш получается равным 256 битам.

**Класс SHA256:**

Все методы и свойства класса – protected, чтобы нельзя было получить доступ извне дружественной функции.

Сам класс состоит из 4 методов, блока с макросами и свойств.

Свойства имеются следующие:

* uint8 - 8 бит

typedef unsigned char uint8;

* uint32 - 32 бита

typedef unsigned int uint32;

Пользовательские типы данных, которые используются в макросах – 8 и 32 бита соответственно.

Размер блока для разбиения (512 бит / 64 байта).

static const unsigned int SHA256\_BLOCK\_SIZE = (512 / 8);

Размер выходного хеша (256 бит / 32 байта).

static const unsigned int HASH\_SIZE = (256 / 8);

Статические свойства – свойства, к которым должна быть возможность получать к доступ до создания объекта класса. Статические свойства, которые хранят в себе размер блока, на которые делится начальное сообщение, и размер итогового хеша (в байтах).

Длина обработанного сообщения.

unsigned int crypted\_message\_len;

Длина ещё не обработанного сообщения.

unsigned int not\_crypted\_message\_len;

Здесь хранится сообщение – 1024 бит / 128 байт.

unsigned char m\_block[2 \* SHA256\_BLOCK\_SIZE] = {0};

Массив для значений хеша.

uint32 hash\_val[8];

Свойства под длину уже обработанной/не обработанной части сообщения, массив под сообщение и массив под значения хешей (тут будет собираться хеш-сумма).

const unsigned int SHA256\_KEYS[64] = {

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,

0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,

0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,

0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,

0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,

0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,

0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,

0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,

0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,

0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,

0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,

0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

};

В алгоритме задействована постоянная часть в виде значений первых 32-битных дробных частей кубических корней 64 простых целых чисел (хранятся в константном массиве).

Теперь рассмотрим все методы.

Инициализация значений хеша.

void initHashesAndLen(){

32-бита дробных частей квадратных корней простых чисел (2, 3, 5, 7, ...).

hash\_val[0] = 0x6a09e667;

hash\_val[1] = 0xbb67ae85;

hash\_val[2] = 0x3c6ef372;

hash\_val[3] = 0xa54ff53a;

hash\_val[4] = 0x510e527f;

hash\_val[5] = 0x9b05688c;

hash\_val[6] = 0x1f83d9ab;

hash\_val[7] = 0x5be0cd19;

Сбрасываем длины сообщения.

not\_crypted\_message\_len = 0;

crypted\_message\_len = 0;

}

Метод 1 – отвечает за начальную инициализацию данных для расчета хеша. Его вызов происходит в конструкторе. Он выставляет начальное хеш-значение, он же сбрасывает переменные длин сообщения.

Потом метод разбиения сообщения на блоки и его обработки. Он довольно объемный, разберем его в несколько этапов – функция разбиения и первичной обработки сообщения.

void separateMessage(const unsigned char\* message, unsigned int len){

Количество блоков разбиения по 512 байт / 64 байта.

unsigned int block\_amount;

Переменные для определения длин сообщения, остатка от сообщений и т.д.

unsigned int rest\_message\_len, not\_in\_blocks\_len;

Указатель на остаток сообщения (не влез в блоки разбиения).

const unsigned char\* leftover\_message;

Считаем остаток длины (если сообщение больше одного блока кодирования). Если длина сообщения не превышает размер блока, то выходим из функции.

not\_in\_blocks\_len = len < SHA256\_BLOCK\_SIZE ? len : SHA256\_BLOCK\_SIZE;

Копируем в m\_block байты из кодируемого сообщения.

memcpy(&m\_block[not\_crypted\_message\_len], message, not\_in\_blocks\_len);

if (not\_crypted\_message\_len + len < SHA256\_BLOCK\_SIZE) {

Приравниваем длину ещё не обработанного сообщения к общей длине сообщения (оно ещё не шифровалось).

not\_crypted\_message\_len += len;

return;

}

Первая часть метода считает – стоит ли разбить сообщение на блоки. Если влезло в один блок, то разбиения нет и происходит выход из метода после того, как сообщение было записано. Если же сообщение большое – после записи сообщения в блок, метод продолжает работать.

Оставшуюся длину сообщения записываем в rest\_message\_len.

rest\_message\_len = len - not\_in\_blocks\_len;

Записываем сколько блоков нужно на остаток сообщение в block\_amount.

block\_amount = rest\_message\_len / SHA256\_BLOCK\_SIZE;

Записываем в leftover\_message оставшееся сообщение, которое не влезло в предыдущий блок.

leftover\_message = message + not\_in\_blocks\_len;

Изменяем наши хеш-функции для первого блока.

computeHash(m\_block, 1);

Изменяем хеш-функции для последующих боков.

computeHash(leftover\_message, block\_amount);

Записываем в not\_in\_blocks\_len сколько символов останется вне блоков.

not\_in\_blocks\_len = rest\_message\_len % SHA256\_BLOCK\_SIZE;

Помещаем в начало m\_block оставшиеся – не влезшие в блоки – символы.

memcpy(m\_block, &leftover\_message[block\_amount << 6], not\_in\_blocks\_len);

Приравниваем длину ещё не обработанного сообщения к количеству не влезших в блоки символов.

not\_crypted\_message\_len = not\_in\_blocks\_len;

Приравниваем длину уже обработанного сообщения к количеству символов в блоках.

crypted\_message\_len += (block\_amount + 1) << 6;

}

В случае, когда сообщение не влезло в блок, мы получаем длину части, которая не поместилась, после чего рассчитываем сколько блоков под неё нужно выделить. Часть сообщения, не поместившаяся в первый блок, обрабатывается отдельно. Отдельно обрабатываем первый блок (вызывается метод вычисления хеша, который будет рассмотрен далее), отдельно – остальные блоки.

По итогу, остается последняя часть сообщения, которая не поместилась в блоки целиком. Мы записываем её длину, чтобы обработать отдельно в самом конце расчета хешей.

Метод, который завершает расчет хешей, и возвращает финальное значение (хеш-сумму) выглядит следующим образом – функция окончания хеширования.

void finalHash(unsigned char\* hash){

Количество блоков разбиения по 512 байт/64 байта.

unsigned int block\_amount;

Размер оставшихся блоков в байтах.

unsigned int rest\_block\_bytes;

Длина сообщения в битах.

unsigned int message\_len\_bits;

Считаем сколько блоков нам осталось обработать.

block\_amount = (1 + ((SHA256\_BLOCK\_SIZE - 9) < (not\_crypted\_message\_len % SHA256\_BLOCK\_SIZE)));

Умножаем общую длину на 8 сообщения (размер сообщения в битах).

message\_len\_bits = (crypted\_message\_len + not\_crypted\_message\_len) << 3;

Умножаем количество блоков на 64 (размер в байтах).

rest\_block\_bytes = block\_amount << 6;

Сбрасываем байты блоков, которые идут после необработанного сообщения (они уже были обработаны – больше не нужны).

memset(m\_block + not\_crypted\_message\_len, 0, rest\_block\_bytes - not\_crypted\_message\_len);

В следующем байте после сообщения выставляем старший бит.

m\_block[not\_crypted\_message\_len] = 0x80;

Распаковываем сообщение со сдвигом.

SHA256\_UNPACK32(message\_len\_bits, m\_block + rest\_block\_bytes - 4);

Вычисляем хеши оставшегося блока.

computeHash(m\_block, block\_amount);

Для значений хешей делаем распаковку со сдвигом (умножением на 4).

for (int i = 0 ; i < 8; i++) {

SHA256\_UNPACK32(hash\_val[i], &hash[i << 2]);

}

}

В методе происходит обработка последней части начального сообщения. От него берется последняя не обработанная часть, потом помещается в блок. Свободные биты этого блока обнуляются, чтобы в блоке было начальное сообщение. Блок обрабатывается, а затем происходит “распаковка” (специальная логическая операция) значений хешей, по алгоритму.

Функция изменений значений хешей.

void computeHash(const unsigned char\* message, unsigned int block\_amount){

Очередь сообщений (каждый элемент – двойное слово).

uint32 w[64];

Массив копий ключей.

uint32 hash\_val\_copy[8];

Временные переменные.

uint32 temp1, temp2;

Подблок – дополнительное разбиение сообщений.

const unsigned char\* sub\_block;

Основной цикл, в котором поочередно перебираются блоки.

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

Записываем в sub\_block куски сообщений с 64-байтным шагом.

sub\_block = message + (i << 6);

Перебираем 64 байта и упаковываем их в sub\_block.

for (int j = 0; j < 16; j++) {

SHA256\_PACK32(&sub\_block[j << 2], &w[j]);

}

Основным методом является метод, который рассчитывает хеш-значения для каждого блока сообщения. Он в цикле – по раундам – обрабатывает каждый блок сообщения, изменяя при этом значения хешей. Блоки разбиваются на маленькие куски данных, которые обрабатываются по алгоритму.

По алгоритму изменяются нулевые индексы.

for (int j = 16; j < 64; j++) {

w[j] = SHA256\_DELTA\_1(w[j - 2]) + w[j - 7] + SHA256\_DELTA\_0(w[j - 15]) + w[j - 16];

}

Копируются хеш-значений.

for (int j = 0; j < 8; j++) {

hash\_val\_copy[j] = hash\_val[j];

}

Цикл сжатия – изменяет копии хеш-значений.

for (int j = 0; j < 64; j++) {

temp1 = hash\_val\_copy[7] + SHA256\_SIGMA\_1(hash\_val\_copy[4]) + SHA256\_CH(hash\_val\_copy[4], hash\_val\_copy[5], hash\_val\_copy[6])

+ SHA256\_KEYS[j] + w[j];

temp2 = SHA256\_SIGMA\_0(hash\_val\_copy[0]) + SHA256\_MAJ(hash\_val\_copy[0], hash\_val\_copy[1], hash\_val\_copy[2]);

hash\_val\_copy[7] = hash\_val\_copy[6];

hash\_val\_copy[6] = hash\_val\_copy[5];

hash\_val\_copy[5] = hash\_val\_copy[4];

hash\_val\_copy[4] = hash\_val\_copy[3] + temp1;

hash\_val\_copy[3] = hash\_val\_copy[2];

hash\_val\_copy[2] = hash\_val\_copy[1];

hash\_val\_copy[1] = hash\_val\_copy[0];

hash\_val\_copy[0] = temp1 + temp2;

}

Возврат обратно хеш-значений.

for (int j = 0; j < 8; j++) {

hash\_val[j] += hash\_val\_copy[j];

}

}

}

Происходит “упаковка” данных, правые повороты, левые повороты, и ещё некоторые составные логические и математические операции, которые задействуются в алгоритме. В последней части метода создаются копии хешей, а затем происходит перестановка ними собой и изменения этих значений. В конце концов, уже их измененные значения могут записаться в изначальные хеш-значения. Потому что для преобразования этих значений задействуются высчитанные значения “очереди сообщений w”, то для каждого сообщения хеш будет получаться различный.

В классе также имеются макросы #define (используются для сокращения кода и более простого его понимания).

#define SHA256\_SHIFT\_R(x, n) (x >> n)

#define SHA256\_ROTATE\_R(x, n) ((x >> n) | (x << ((sizeof(x) << 3) - n)))

#define SHA256\_ROTATE\_L(x, n) ((x << n) | (x >> ((sizeof(x) << 3) - n)))

#define SHA256\_CH(x, y, z) ((x & y) ^ (~x & z))

#define SHA256\_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))

#define SHA256\_SIGMA\_0(x) (SHA256\_ROTATE\_R(x, 2) ^ SHA256\_ROTATE\_R(x, 13) ^ SHA256\_ROTATE\_R(x, 22))

#define SHA256\_SIGMA\_1(x) (SHA256\_ROTATE\_R(x, 6) ^ SHA256\_ROTATE\_R(x, 11) ^ SHA256\_ROTATE\_R(x, 25))

#define SHA256\_DELTA\_0(x) (SHA256\_ROTATE\_R(x, 7) ^ SHA256\_ROTATE\_R(x, 18) ^ SHA256\_SHIFT\_R(x, 3))

#define SHA256\_DELTA\_1(x) (SHA256\_ROTATE\_R(x, 17) ^ SHA256\_ROTATE\_R(x, 19) ^ SHA256\_SHIFT\_R(x, 10))

Макрос распаковки 4 байта/32 бита.

#define SHA256\_UNPACK32(x, str){ \

\*((str) + 3) = (uint8) ((x) ); \

\*((str) + 2) = (uint8) ((x) >> 8); \

\*((str) + 1) = (uint8) ((x) >> 16); \

\*((str) + 0) = (uint8) ((x) >> 24); \

}

Макрос упаковки 4 байта/32 бита.

#define SHA256\_PACK32(str, x){ \

\*(x) = ((uint32) \*((str) + 3) ) \

| ((uint32) \*((str) + 2) << 8) \

| ((uint32) \*((str) + 1) << 16) \

| ((uint32) \*((str) + 0) << 24); \

}

Данные операции – стандартные для алгоритма хеширования. Они состоят из примитивных логических и математических операций.

Последний член класса – дружественная функция sha256. Она принимает строку начального сообщения и возвращает строку с хешем.

Хеширование алгоритмом SHA256.

string sha256(string input){

Создаем массив unsigned char на 256 бит (нулевых).

unsigned char hash[SHA256::HASH\_SIZE] = {0};

Создаем объект SHA256 и инициализируем его.

SHA256 sha\_obj;

Метод разбиения и первичного шифрования.

sha\_obj.separateMessage( (unsigned char\*)input.c\_str(), input.length());

Метод финального шифрования.

sha\_obj.finalHash(hash);

Буффер для хеша.

char final\_hash[2 \* SHA256::HASH\_SIZE + 1] = {0};

final\_hash[2 \* SHA256::HASH\_SIZE] = 0;

По 2 байта записываем хеш в финальный массив.

for (int i = 0; i < SHA256::HASH\_SIZE; i++)

sprintf(final\_hash + i \* 2, "%02x", hash[i]);

Приводим символьный массив к строке и возвращаем готовый хеш.

return string(final\_hash);

}

В ней создается массив под хеш, создает объект SHA256, в котором будут расчеты. Из него поочередно вызываются методы separateMessage и finalHash. В результате нам возвращается символьный массив, в котором уже хранится хеш сумма. Записываются по 2 байта её в финальную строку, в которую и возвращаются из функции.

По результатам сравнения хеш-сумм данной реализации и хеш-сумм реализации sha256 в онлайн сервисе различий найдено не было (стандартной реализации sha256 в c++ не найдено). Все сообщения на выходе дали одинаковые хеш суммы. Ниже представлена таблица, подтверждающая это. Ссылка на веб-сервис - <http://crypt-online.ru/crypts/sha256/>).

|  |  |  |
| --- | --- | --- |
| Сообщение | Данная реализация | Веб-сервис |
| Hi! | ca51ce1fb15acc6d69b8a5700256172fcc507e02073e6f19592e341bd6508ab8 | ca51ce1fb15acc6d69b8a5700256172fcc507e02073e6f19592e341bd6508ab8 |
| Hello world! | c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a | c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a |
| sha256 online sha256 online | 744535f6152737a30c47b518efa053bfcf3771929eca713db0f1834ab3256a3c | 744535f6152737a30c47b518efa053bfcf3771929eca713db0f1834ab3256a3c |
| Lorem ipsum dolor sit amet, cons | 7a7d7888975ab24321cf4273c781313a6ff054fb667892dbee18370ad34ac24c | 7a7d7888975ab24321cf4273c781313a6ff054fb667892dbee18370ad34ac24c |
| SHA256 - хеш-функция из семейства алгоритмов SHA-2 предназначена для создания «отпечатков» или «дайджестов» для сообщений произвольной длины. Применяется в различных приложениях или компонентах, связанных с защитой информации. | e7793d18a715aa1ea8f8fabbb521327e1cd9353d1a581aa311a09cd23050566c | e7793d18a715aa1ea8f8fabbb521327e1cd9353d1a581aa311a09cd23050566c |
| . | cdb4ee2aea69cc6a83331bbe96dc2caa9a299d21329efb0336fc02a82e1839a8 | cdb4ee2aea69cc6a83331bbe96dc2caa9a299d21329efb0336fc02a82e1839a8 |
| Криптографические хеш-функции | 5c2c63e82f946540e10bbefc604443d6193b37ba20163bf6718ca2ac9fa9c17c | 5c2c63e82f946540e10bbefc604443d6193b37ba20163bf6718ca2ac9fa9c17c |
| Привет! | b2b11afc89e6a4635f13e1fe404909ee873b2aa9b4d62ce414d327d39fef258e | b2b11afc89e6a4635f13e1fe404909ee873b2aa9b4d62ce414d327d39fef258e |
| hash | d04b98f48e8f8bcc15c6ae5ac050801cd6dcfd428fb5f9e65c4e16e7807340fa | d04b98f48e8f8bcc15c6ae5ac050801cd6dcfd428fb5f9e65c4e16e7807340fa |
| d04b98f48e8f8bcc15c6ae5ac050801cd6dcfd428fb5f9e65c4e16e7807340fa | 707617679b50fe693cc7d098be136d50f43ec95f9e4b0903b517fdb122e365f3 | 707617679b50fe693cc7d098be136d50f43ec95f9e4b0903b517fdb122e365f3 |

Таблица 1. Результат работы хеш-функции.

В ходе проведения теста зависимости работы функции при обработке сообщений различной длины от времени выполнения, были получены следующие результаты:

График 1.

# Заключение.

Все необходимые функции и поставленные задачи данной лабораторной реализованы. Функция хеширования была реализована алгоритмом sha2 (sha256). Знания о работе алгоритма были получены, а также написана её реализация с использование класса и дружественной функции. Были проведены тесты корректности и производительности реализации. Стоит отметить, что в данной работе, ради интереса, в методах и макросах применялась операция сдвига влево, которая заменяет умножение на степень двойки. При засекании времени работы функции хеширования, выдавалось нулевое значение. Адекватные значения не получались на моем компьютере, поэтому было принято решение использовать онлайн-компилятор, который и выдал корректные значения. Данное затруднение может свидетельствовать о довольно высокой скорости работы алгоритма.