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

Выполнил студент группы .....КС-30.....Сидоров Сергей Александрович
Ссылка на репозиторий: ..https://github.com/MUCTR-IKT-CPP/Sidorov.S.A_KS-30_2sem/tree/main/lab4

Приняли: ..... Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович

Дата сдачи: 30.05.2021

Оглавление

Описание задачи.

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

MD5

SHA1

SHA2

Стриборг

RIPEMD-160 Доп вариант для тех кто хочет посложнее:

Luffa

SHA3

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.

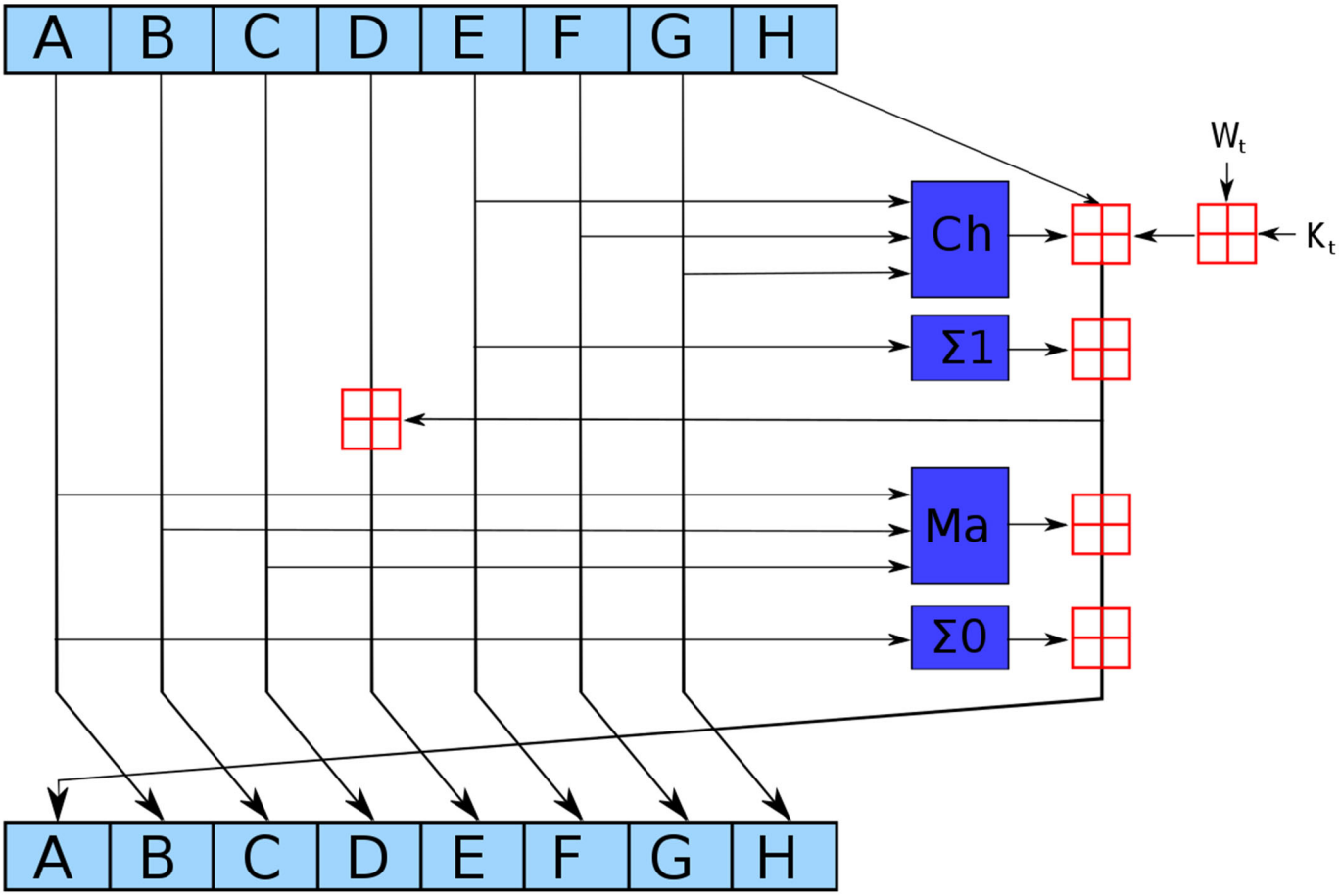


Рисунок 1 Схема раунда SHA2

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

Выполнение задачи.

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

В данной лабораторной работе использовался наиболее популярный вариант SHA2 – SHA256, в котором итоговый хэш 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:

```
typedef unsigned char uint8;
typedef unsigned int uint32;
```

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

```
static const unsigned int SHA2_BLOCK_SIZE = (512 / 8);
static const unsigned int HASH_SIZE = (256 / 8);
```

Статические свойства, которые хранят в себе размер одного блока (на блоки делится начальное сообщение) и размер итогового хэша (в байтах). Статические они, так как должна быть возможность получать к ним доступ до создания объекта класса.

```
unsigned int crypted_message_len;
unsigned int not_crypted_message_len;
unsigned char m_block[2 * SHA2_BLOCK_SIZE] = {0};
uint32 hash_val[8];
```

Свойства под длину уже обработанной части сообщения, длину ещё не обработанной части сообщения, массив под само сообщение и массив под значения хэшей (из них и будет собираться хэш-сумма).

```
const unsigned int SHA2_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 будет задействована постоянная часть в виде значений первых 32-битных дробных частей кубических корней 64 простых целых чисел. Они хранятся в константном массиве.

Объявление свойств класса на этом заканчивается. Теперь рассмотрим все методы.

```
void initHashesAndLen(){

        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;

        not_crypted_message_len = 0;
        crypted_message_len = 0;
    }
```

Первый метод отвечает за начальную инициализацию данных для расчета хэша. Он вызывается в конструкторе и выставляет начальные значений хэш-значений, а также сбрасывает переменные длин сообщения.

Далее идет метод разбиения сообщения на блоки и его обработки. Он довольно объемный, разберем его в несколько этапов:

```
    void separateMessage(const unsigned char* message, unsigned int len){
        unsigned int block_amount;
        unsigned int rest_message_len, not_in_blocks_len;
        const unsigned char* leftover_message;
        not_in_blocks_len = len < SHA2_BLOCK_SIZE ? len : SHA2_BLOCK_SIZE;

        memcpy(&m_block[not_crypted_message_len], message, not_in_blocks_len);

        if (not_crypted_message_len + len < SHA2_BLOCK_SIZE) {
            not_crypted_message_len += len;
            return;
        }
```

Первая половина метода считает – нужно ли разбивать наше сообщение на блоки. Если оно влезает в один блок, то разбиения не происходит, и происходит выход из метода сразу после того, как

сообщение было записано в блок. Если же сообщение больше, чем один блок, то после записи сообщения в блок, метод продолжает своё выполнение.

```
            rest_message_len = len - not_in_blocks_len;
            block_amount = rest_message_len / SHA2_BLOCK_SIZE;
            leftover_message = message + not_in_blocks_len;

            computeHash(m_block, 1);
            computeHash(leftover_message, block_amount);
            not_in_blocks_len = rest_message_len % SHA2_BLOCK_SIZE;
            memcpy(m_block, &leftover_message[block_amount << 6], not_in_blocks_len);
            not_crypted_message_len = not_in_blocks_len;
            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. Оставшуюся часть сообщения (не поместившуюся в первый блок) мы обработаем отдельно. Далее отдельно обрабатываем первый блок (вызывается метод вычисления хэша, который будет рассмотрен далее), и отдельно – остальные блоки.

По итогу, у нас останется последняя часть сообщения, которая не поместилась в блоки целиком. Мы записываем её длину, чтобы обработать отдельно в самом конце расчета хэшей.

Метод, который завершает расчет хэшей, и возвращает финальное значение (хэш-сумму) выглядит следующим образом:

```
        void finalHash(unsigned char* hash){
            unsigned int block_amount;
            unsigned int rest_block_bytes;
            unsigned int message_len_bits;

            block_amount = (1 + ((SHA2_BLOCK_SIZE - 9) < (not_crypted_message_len % SHA2_B
LOCK_SIZE)));
            message_len_bits = (crypted_message_len + not_crypted_message_len) << 3;
            rest_block_bytes = block_amount << 6;
            memset(m_block + not_crypted_message_len, 0, rest_block_bytes - not_crypted_me
ssage_len);
            m_block[not_crypted_message_len] = 0x80;

            SHA2_UNPACK32(message_len_bits, m_block + rest_block_bytes - 4);
            computeHash(m_block, block_amount);

            for (int i = 0 ; i < 8; i++) {
                SHA2_UNPACK32(hash_val[i], &hash[i << 2]);
            }
        }
```

В данном методе происходит обработка последней части начального сообщения. От него берется последняя не обработанная часть, после чего помещается в блок. Оставшиеся (свободные) биты этого блока обнуляются, чтобы в блоке было только начальное сообщение. Этот блок обрабатывается, а

затем происходит “распаковка” (специальная логическая операция) значений хэшей, согласно алгоритму.

```
void computeHash(const unsigned char* message, unsigned int block_amount){

            uint32 w[64];
            uint32 hash_val_copy[8];
            uint32 temp1, temp2;

            const unsigned char* sub_block;

            for (int i = 0; i < block_amount; i++) {
                sub_block = message + (i << 6);
                for (int j = 0; j < 16; j++) {
                    SHA2_PACK32(&sub_block[j << 2], &w[j]);
                }
```

Основным методом является метод, который рассчитывает хэш-значения для каждого блока сообщения. Он в цикле (по раундам) обрабатывает каждый блок сообщения, изменяя при этом значения хэшей. Сами блоки разбиваются на более мелки куски данных, которые обрабатываются по алгоритму.

```
for (int j = 16; j < 64; j++) {
                    w[j] =  SHA2_DELTA_1(w[j - 2]) + w[j -  7] + SHA2_DELTA_0(w[j - 15]) +
 w[j - 16];
                }

                for (int j = 0; j < 8; j++) {
                    hash_val_copy[j] = hash_val[j];
                }

                for (int j = 0; j < 64; j++) {
                    temp1 = hash_val_copy[7] + SHA2_SIGMA_1(hash_val_copy[4]) + SHA2_CH(ha
sh_val_copy[4], hash_val_copy[5], hash_val_copy[6])
                        + SHA2_KEYS[j] + w[j];
                    temp2 = SHA2_SIGMA_0(hash_val_copy[0]) + SHA2_MAJ(hash_val_copy[0], ha
sh_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;
                }

                for (int j = 0; j < 8; j++) {
                    hash_val[j] += hash_val_copy[j];
                }
            }
        }
```

Происходит так называемая “упаковка” данных, а также правые повороты, левые повороты, и ещё некоторые составные логические и математические операции, которые задействуются в данном алгоритме. О них будет сказано дальше.

В последней части этого метода создаются копии наших хэшей, после чего они переставляются между собой и изменяют свои значения, чтобы в итоге, уже их измененные значения мы могли записать в изначальные хэш-значения. Так как для преобразования этих значений задействуются высчитанные значения “очереди сообщений w”, то для каждого сообщения хэш будет получаться различный.

Так же в классе имеется дружественная функция, но её мы рассмотрим в самом конце.

Помимо методов и свойств в классе имеются макросы define. Они используются для сокращения кода и более простого его понимания.

```
#define SHA2_SHIFT_R(x, n)    (x >> n)
        #define SHA2_ROTATE_R(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
        #define SHA2_ROTATE_L(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
        #define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
        #define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
        #define SHA2_SIGMA_0(x) (SHA2_ROTATE_R(x,  2) ^ SHA2_ROTATE_R(x, 13) ^ SHA2_ROTATE
_R(x, 22))
        #define SHA2_SIGMA_1(x) (SHA2_ROTATE_R(x,  6) ^ SHA2_ROTATE_R(x, 11) ^ SHA2_ROTATE
_R(x, 25))
        #define SHA2_DELTA_0(x) (SHA2_ROTATE_R(x,  7) ^ SHA2_ROTATE_R(x, 18) ^ SHA2_SHIFT_
R(x,  3))
        #define SHA2_DELTA_1(x) (SHA2_ROTATE_R(x, 17) ^ SHA2_ROTATE_R(x, 19) ^ SHA2_SHIFT_
R(x, 10))

        #define SHA2_UNPACK32(x, str){              \
            *((str) + 3) = (uint8) ((x)      );       \
            *((str) + 2) = (uint8) ((x) >>  8);       \
            *((str) + 1) = (uint8) ((x) >> 16);       \
            *((str) + 0) = (uint8) ((x) >> 24);       \
        }

        #define SHA2_PACK32(str, x){                \
            *(x) =   ((uint32) *((str) + 3)   )       \
                | ((uint32) *((str) + 2) <<  8)       \
                | ((uint32) *((str) + 1) << 16)       \
                | ((uint32) *((str) + 0) << 24);      \
        }
```

Все эти операции являются стандартными для данного алгоритма хэширования, и состоят из примитивных логических и математических операций, поэтому не требуют дополнительного пояснения.

Последним членом класса выступает дружественная функция sha256, которая принимает на вход строку начального сообщения и возвращает строку с хэшем. Её реализация представлена таким образом:

```
string sha256(string input){
    unsigned char hash[SHA2::HASH_SIZE] = {0};
    SHA2 sha_obj;

    sha_obj.separateMessage( (unsigned char*)input.c_str(), input.length());
    sha_obj.finalHash(hash);

    char final_hash[2 * SHA2::HASH_SIZE + 1] = {0};
    final_hash[2 * SHA2::HASH_SIZE] = 0;

    for (int i = 0; i < SHA2::HASH_SIZE; i++)
        sprintf(final_hash + i * 2, "%02x", hash[i]);

    return string(final_hash);
}
```

В ней создается массив под сам хэш, создает объект SHA256, в котором и будут проводиться расчеты. Из него поочередно вызываются методы separateMessage и finalHash. В результате нам возвращается массив char, в котором уже хранится наша хэш сумма. Мы записываем по 2 байта её в финальную строку, которую и возвращаем из функции.

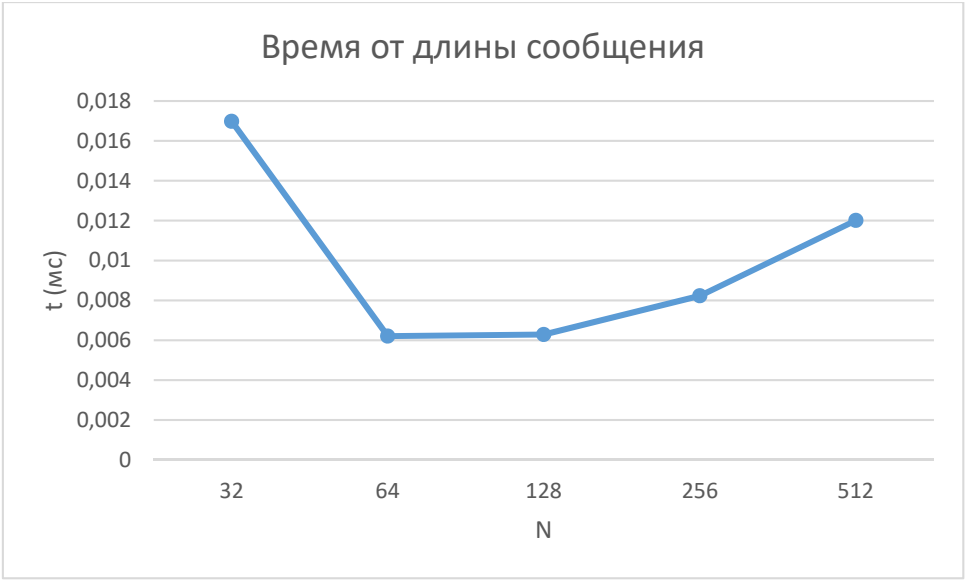
По результатам сравнения хэш-сумм данной реализации и хэш-сумм реализации sha256 в онлайн сервисе различий найдено не было (стандартной реализации sha256 в c++ с адекватной документацией мною не найдено). Все сообщения на выходе дали одинаковые хэш суммы. Ниже представлена таблица, подтверждающая это.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

Таблица 1 Результат работы хэш-функции

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 хэш-функции	5c2c63e82f946540e10bbefc604443d6193b37ba20163bf6718ca2ac9fa9c17c	5c2c63e82f946540e10bbefc604443d6193b37ba20163bf6718ca2ac9fa9c17c
Привет!	b2b11afc89e6a4635f13e1fe404909ee873b2aa9b4d62ce414d327d39fef258e	b2b11afc89e6a4635f13e1fe404909ee873b2aa9b4d62ce414d327d39fef258e
hash	d04b98f48e8f8bcc15c6ae5ac050801cd6dcfd428fb5f9e65c4e16e7807340fa	d04b98f48e8f8bcc15c6ae5ac050801cd6dcfd428fb5f9e65c4e16e7807340fa
d04b98f48e8f8bcc15c6ae5ac050801cd6dcfd428fb5f9e65c4e16e7807340fa	707617679b50fe693cc7d098be136d50f43ec95f9e4b0903b517fdb122e365f3	707617679b50fe693cc7d098be136d50f43ec95f9e4b0903b517fdb122e365f3

В ходе проведения теста зависимости работы функции при обработке сообщений различной длины от времени выполнения, были получены следующие результаты (сами полученные данные можно посмотреть в файле res.txt – здесь приведен только график):


Время от длины сообщения
0,018
0,016
0,014
0,012
0,01
0,008
0,006
0,004
0,002
0
t (мс)
32
64
128
256
512
N

Заключение.

В ходе данной лабораторной работы была реализована функция хэширования алгоритмом sha2 (sha256), которая является очень популярной. Была изучена работа самого алгоритма и написана её реализация с использование класса и дружественной функции, а также были проведены тесты корректности и производительности реализации.

Отдельно стоит заметить, что в методах и макросах применялась операция сдвига влево для замены умножению на степень двойки. Это делалось без исключительной необходимости – только ради интереса работы с битовыми операторами.