Часть 1:  
  
#include <bits.h>

#include <iostream>

#include <bitset>

#include <sstream>

#include <algorithm>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Russian");///<-- Включила русский язык

cout << "Enter the Name: ";/// <-- Прошу ввести имя

string Word;/// <--объявила строковую переменную Word, в ней будет храниться имя

getline(cin, Word); /// <-- Получила имя

int lenth\_Word = Word.length();///<-- Вычислила его длину

/\* Хочу перевести строку в символьный массив.

Это необходимо, тк мы будем работать в первым и последним символами имени \*/

char\* chars = new char[lenth\_Word];/// <-- Создала динамический символьный массив chars, длина которого равна длине имени

Word.copy(chars, lenth\_Word);/// <-- Скопировала строку имени в массив по-символьно

/\* Дальше нужно получить по-битовое представление двух символов \*/

bitset<8> char1; bitset<8> char2;

char1 = bitset<8>(chars[0]); char2 = bitset<8>(chars[lenth\_Word-1]);

/\* Тоесть я объявила, что в char1 и char2 буду помещать по-битовые представления,

а затем дала переменным char1 и char 2 значения этих самых по-битовых представлений \*/

string Char1 = char1.to\_string<char, char\_traits<char>, allocator<char> >();

string Char2 = char2.to\_string<char, char\_traits<char>, allocator<char> >();

/\* Объявила строки Char1 и Char2, поместила в них по-битовые представления символов \*/

/\* В итоге у нас две строки: Char1 и Char2, в которых хранятся биты первого и последнего символа имени

в строковом виде соответственно \*/

/\* Теперь я эти строки хочу перевести в символьные массивы

Возможно, можно было как-то обойтись вообще без строк Char1 и Char2, сразу отправив char1 и char2 в массивы,

но мои познания ограничены \*/

char\* Bit1 = new char[8]; char\* Bit2 = new char[8];/// <-- Объявила динамические символьные массивы размером 8 эл-в

Char1.copy(Bit1, 8); Char2.copy(Bit2, 8); /// <-- Поместила в них строки с битами

/\* Дальше разбираемся с паритетными битами.

Т.к. биты принимают значения 1 или 0 (true или false), взяла булевый тип.\*/

bool Flag1 = 0; bool Par\_Plus\_Bit1; bool Par\_Minus\_Bit1; /// <--Создаю булевую переменную для положительного парит.бита и для отрицательного

/\* Выше я так же создала булевую переменную Flag1, она нужна для выполнения условия о чётности/нечётности

Работает это так:изначально Flag1= 0, если в массиве из битов встретилась 1, то Flag1 =1 (команда Flag1++ увеличивает значение на один),

если 1 встретилась снова, то Flag= 0 (тк если к булевой переменной, равной 1 применить ++, то переменная снова примет значение 0)

Таким образом, когда кол-во 1 нечётно, Flag1= 1, а когда кол-во чётно, то Flag1= 0 \*/

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

{

if (Bit1[i] == 1) { Flag1++; }/// <-- Встретили 1, увеличили Flag1

}

/\*

Дальше нужно выполнить условие

Если Flag1 в итоге равен 1, значит единица встретила нечётное кол-во раз,

соответственно чётный паритетный бит станет true

Если Flag1 в итоге равен 0, значит единица встретила чётное кол-во раз,

соответственно чётный паритетный бит станет false

if (Flag1 == 1) { Par\_Plus\_Bit1 = 1; Par\_Minus\_Bit1 = 0; }

else { Par\_Plus\_Bit1 = 0; Par\_Minus\_Bit1 = 1; }

/\* Со вторым символом проделываем то же самое \*/

bool Flag2 = 0; bool Par\_Plus\_Bit2; bool Par\_Minus\_Bit2;

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

{

if (Bit2[i] == 1) { Flag2++; }

}

if (Flag2 == 1) { Par\_Plus\_Bit2 = 1; Par\_Minus\_Bit2 = 0; }

else { Par\_Plus\_Bit2 = 0; Par\_Minus\_Bit2 = 1; }

cout << endl; system("pause"); system("cls");/// <-- Очищаю экран (нужно нажать на любую кнопку)

/\* Осталось вывести на экран.\*/

cout << " Буква " << "Битовая строка " << "Чётный паритетный бит " << "нечётный паритетный бит"<<endl<<endl;

cout << " " << chars[0] << " " << Char1 << " " << Par\_Plus\_Bit1 << " " << Par\_Minus\_Bit1 << endl;

cout << " " << chars[lenth\_Word-1] << " " << Char2 << " " << Par\_Plus\_Bit2 << " " << Par\_Minus\_Bit2 << endl;

}

Часть 2.  
Её я делала не сама. Взяла из интернета и переделала, поэтому о ньюансах не расскажу  
Но суть такая:

*1. Инициализируются переменные*  
(первые 32 бита *дробных частей* квадратных корней первых восьми простых чисел [от 2 до 19]):  
h0 := 0x6A09E667  
h1 := 0xBB67AE85  
h2 := 0x3C6EF372  
h3 := 0xA54FF53A  
h4 := 0x510E527F  
h5 := 0x9B05688C  
h6 := 0x1F83D9AB  
h7 := 0x5BE0CD19

*Таблица констант*  
(первые 32 бита *дробных частей* кубических корней первых 64 простых чисел [от 2 до 311]):  
k[0..63] :=  
 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

2. *Предварительная обработка:*  
m := message ǁ [*единичный бит*]  
m := m ǁ [k *нулевых бит*], где **k** — наименьшее неотрицательное число, такое что   
 (L + 1 + K) mod 512 = 448, где L — число бит в сообщении (сравнима по модулю 512 c 448)  
m := m ǁ *Длина*(message) — длина исходного сообщения в битах в виде 64-битного числа с порядком байтов от старшего к младшему

3. *Далее сообщение обрабатывается последовательными порциями по 512 бит:*  
разбить сообщение на куски по 512 бит  
**для** каждого куска разбить кусок на 16 слов длиной 32 бита (с порядком байтов от старшего к младшему внутри слова): w[0..15]

4.*Сгенерировать дополнительные 48 слов:*  
**для** i **от** 16 **до** 63  
s0 := (w[i-15] **rotr** 7) **xor** (w[i-15] **rotr** 18) **xor** (w[i-15] **shr** 3)  
s1 := (w[i-2] **rotr** 17) **xor** (w[i-2] **rotr** 19) **xor** (w[i-2] **shr** 10)  
w[i] := w[i-16] **+** s0 **+** w[i-7] **+** s1

*5.Инициализация вспомогательных переменных:*  
a := h0  
b := h1  
c := h2  
d := h3  
e := h4  
f := h5  
g := h6  
h := h7

6.*Основной цикл:*  
**для** i **от** 0 **до** 63  
Σ0 := (a **rotr** 2) **xor** (a **rotr** 13) **xor** (a **rotr** 22)  
Ma := (a **and** b) **xor** (a **and** c) **xor** (b **and** c)  
t2 := Σ0 + Ma  
Σ1 := (e **rotr** 6) **xor** (e **rotr** 11) **xor** (e **rotr** 25)  
Ch := (e **and** f) **xor** ((**not** e) **and** g)  
t1 := h + Σ1 + Ch + k[i] + w[i]h := g  
g := f  
f := e  
e := d + t1  
d := c  
c := b  
b := a  
a := t1 + t2

7.*Добавить полученные значения к ранее вычисленному результату:*  
h0 := h0 + a  
h1 := h1 + b   
h2 := h2 + c  
h3 := h3 + d  
h4 := h4 + e  
h5 := h5 + f  
h6 := h6 + g   
h7 := h7 + h

*Получить итоговое значения хеша:*  
digest = hash = h0 ǁ h1 ǁ h2 ǁ h3 ǁ h4 ǁ h5 ǁ h6 ǁ h7

1.

constexpr array<uint32\_t, 64> K = {

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

};

uint8\_t m\_data[64]; /// Тут мы будем храник хэш. данные

uint64\_t m\_bitlen(0);/// Длина битов

uint32\_t m\_blocklen(0);/// Кол-во блоков

/\* Массив из квадратов первых 8 простых чисел \*/

uint32\_t m\_state[8] = {

0x6a09e667,

0xbb67ae85,

0x3c6ef372,

0xa54ff53a,

0x510e527f,

0x9b05688c,

0x1f83d9ab,

0x5be0cd19

};

2.  
Предварительная обработка данных идёт в функциях main (с неё ВСЁ ВСЕГДА начинается), update (их там две: в одной строка просто преобразуется в uint8\_t, в следующей работа с блоками), затем функция pad (основная первичная обработка: добавление единицы и нулей, сдвиги в конец )

У нас получилось, что шаг 3 и 2 по сути- в связаных функциях: update и в pad, поэтому пропускаю его.  
Шаг 4 там же.

Основной замес происходит в transform (шаги 5,6,7)

Ваша задача- рассказать поверхностно:

**В main взяли в строку**, преобразовали в биты и блоки битов ( в update), первичная обработка данных типа добавления нулей, единицы и длины в pad, само шифрование- в transform.  
Будет сильно докапываться- скажите, что часть кода брали в интернете и детали плохо поняли, но главное- что в код встало и заработало.  
Некоторые строчки прокомментировала.  
Весь листинг в другом документе.