Lab5——Hash 函数 MD5

学号：2111033
姓名：艾明旭
年级：2021 级
专业：信息安全

一、实验内容说明

1、实验目的

通过实际编程了解 MD5 算法的过程，加深对 Hash 函数的认识

2、实验要求

1. 自己编写完整的 MD5 实现代码，并提交程序和程序流程图
2. 对编好的 MD5 算法，测试其雪崩效应，要求给出文本改变前和改变后的 Hash 值，并计算出改变的位数。写出 8 次测试的结果，并计算出平均改变的位数

3、实验步骤

1. 算法分析:
   - 请参照教材内容，分析 MD5 算法实现的每一步原理。
2. 算法实现:
   - 利用 Visual C++语言，自己编写 MD5 的实现代码，并检验代码实现的正确性。
3. 雪崩效应检验：
   - 尝试对一个长字符串进行 Hash 运算，并获得其运算结果。对该字符串进行轻微的改动，比如增加一个空格或标点，比较 Hash 结果值的改变位数。进行 8 次这样的测试。

二、实验环境

- 操作系统：win11
- 软件系统：visual studio
- 编译工具：vs2022
- 编程语言：C++

三、实验过程

本次实验首先翻阅课本，对理论课上的知识进行回顾，然后设计整个实验的流程图以及各个结构体和函数的大致思路，然后进行具体代码的编写实现，以下为具体过程：

1、重要函数分析

(1)zip()函数

- 首先 zip 函数，如四个运算和循环左移等 F(),G(),H(),I(),leftshift()，四轮函数计算等
- 然后把这些函数整合起来放入
- 在能够压缩处理之前，还需要对数据进行分组，采用 for 循环，将数据分为 32 位一组
- 然后再执行一些列计算操作

(2)encode()函数

- 加密函数将 `int` 数组转换为 `char`，即将十进制转换为十六进制
- 在转换时，使用小端序存放结果

(3)init()函数

1. 首先设定了一个 count[2]数组，负责记录当前字符串位数
2. 获取当前已有的字节数（count[0]>>3 再模 64）
3. 接下来用 `count[0]+len<<3`，即已有位数加上新增加的位数
4. 然后判断是否有溢出
5. 若有，则将高位 `count[1]+1`
6. 让 count[1]获取高位的位数，即先让 len 右移 32 得到高位，再左移 3 位得到位数，即 count[1]+=len>>29
7. 将新加入字符长度与待填充长度进行比较，在第一次调用此函数时，等同于将输入的字符串字节数与 64 字节数即 512 比特进行比较
8. 若大于等于，则可以先将数据按 64 字节分组，先对这些 64 字节的组执行压缩函数，然后再对最后一组不足 64 字节的数据执行后续填充操作等
9. 最后函数执行 `memcpy(&buffer[nowlength], &input[i], len - i)`，进行拷贝。

(4)get_digest()函数

该函数负责对最后一组进行填充、压缩等操作：

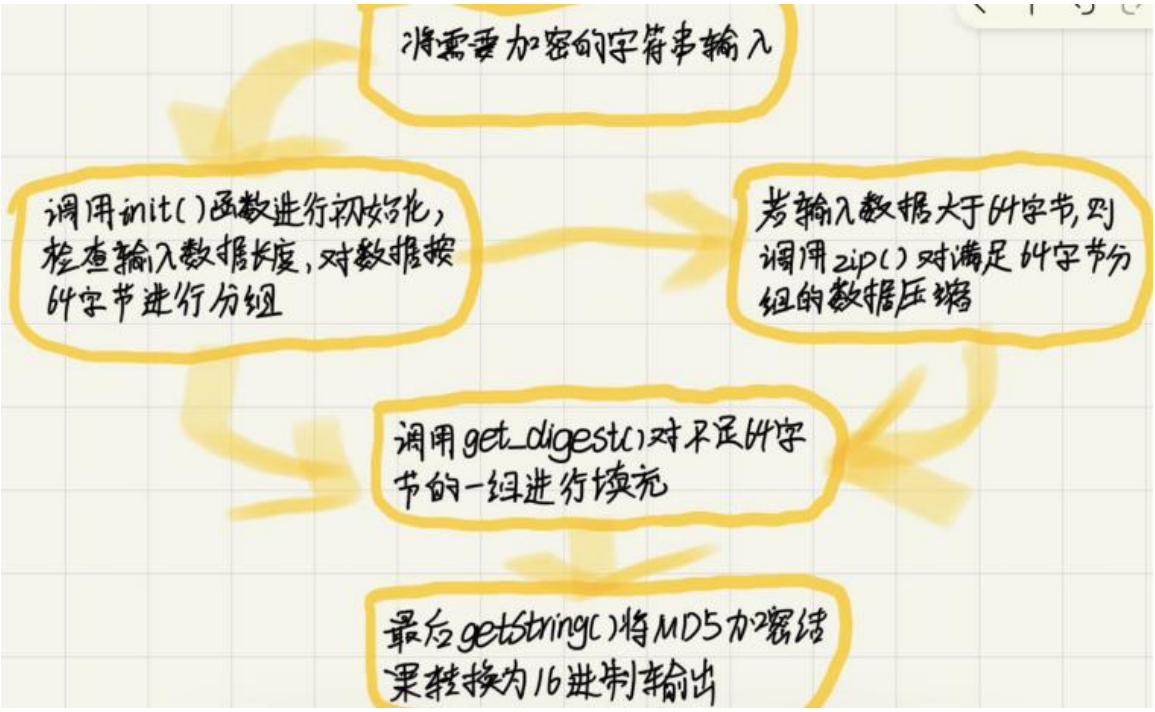
1. 首先将 count 中计算的数据长度用 encode 转换，转换完的数据可以直接填在最后 8 字节
2. 接着获取当前数据已有的字节，计算填充字节长度，如果长度小于 56 字节即 448 比特，则填充字节=56-数据字节，否则，填充字节=120-数据字
3. 填充字节是以 1 开头，后续全是 0，则构造一个数组 tinachong[64]，其开头为 0x80 即 10000000，用这个数组进行填充

4. 调用刚刚的 `init(tianchong, padLen)`将填充数组的内容以一定长度填入我们的最后一个分组
5. 将最后 8 字节填入之前算好数据长度，再调用压缩函数进行计算即可

(5)getstring()函数

- 获得加密后的字符串函数使用 for 循环，将数据转化为 16 进制数

总体流程图如下所示：



较为重要的函数到这里结束，接下来给出相应的代码

2、代码实现

- 首先还是为了方便后面的雪崩效应写的将 16 进制字符串转换为 2 进制 bit 流的函数

```
void getbit(string a, bitset<128>& temp)
{
    int num = 127;
    // 如果无前缀0x 则这里i 需要从2 开始
    for (int i = 0; i < a.length(); i++)
    {
        if (a[i] <= '9')
        {
            for (int j = 0; j < 4; j++)
            {
                temp[num--] = HexToBit[a[i] - 48][j];
            }
        }
```

```
        else
        {
            for (int j = 0; j < 4; j++)
            {
                temp[num--] = HexToBit[a[i] - 65 + 10][j];
            }
        }
    }
}
```

- 接下来是 4 个轮函数的定义，其中 F(),G(),H(),I()函数是书本上的基本逻辑函数

```
void round1(unsigned int& a, unsigned int& b, unsigned int& c, unsigned
 int& d, unsigned int& x, unsigned int s, unsigned int ac)
{
    a += F(b, c, d) + x + ac;
    a = leftshift(a, s);
    a += b;
}
void round2(unsigned int& a, unsigned int& b, unsigned int& c, unsigned
 int& d, unsigned int& x, unsigned int s, unsigned int ac)
{
    a += G(b, c, d) + x + ac;
    a = leftshift(a, s);
    a += b;
}
void round3(unsigned int& a, unsigned int& b, unsigned int& c, unsigned
 int& d, unsigned int& x, unsigned int s, unsigned int ac)
{
    a += H(b, c, d) + x + ac;
    a = leftshift(a, s);
    a += b;
}
void round4(unsigned int& a, unsigned int& b, unsigned int& c, unsigned
 int& d, unsigned int& x, unsigned int s, unsigned int ac)
{
    a += I(b, c, d) + x + ac;
    a = leftshift(a, s);
    a += b;
}
```

- 然后是填充和压缩函数

```
unsigned char* MD5::get_digest()
{
    if (!finished)
    {
        finished = true;

        unsigned char bits[8];
```

```
        unsigned int oldState[4];
        unsigned int oldCount[2];
        int nowlength, waitlength;

        memcpy(oldState, link, 16);
        memcpy(oldCount, count, 8);

        encode(count, bits, 8);

        nowlength = (unsigned int)((count[0] >> 3) & 0x3f);
        waitlength = (nowlength < 56) ? (56 - nowlength) : (120 - nowle
ngth);
        init(tianchong, waitlength);


        nowlength = (unsigned int)((count[0] >> 3) & 0x3f);

        memcpy(&buffer[nowlength], bits, 8);
        zip(buffer);

        encode(link, digest, 16);
        memcpy(link, oldState, 16);
        memcpy(count, oldCount, 8);
    }
    return digest;
}
```

- 接下来是初始化函数

```
void MD5::init(unsigned char* input, int len)
{

    unsigned int i, nowlength, waitlength;

    finished = false;

    nowlength = (unsigned int)((count[0] >> 3) & 0x3f);
    count[0] += (unsigned int)len << 3;
    if ((count[0]) < ((unsigned int)len << 3)) {
        count[1] += 1;
    }
    count[1] += ((unsigned int)len >> 29);

    waitlength = 64 - nowlength;
    if (len >= waitlength) {
        memcpy(&buffer[nowlength], input, waitlength);
        zip(buffer);
        for (i = waitlength; i + 63 < len; i += 64) {
            zip(&input[i]);
        }
```

```
        nowlength = 0;
    }
    else {
        i = 0;
    }
    memcpy(&buffer[nowlength], &input[i], len - i);
}
```

- 接下来是压缩函数，整合全局步骤

```
void MD5::zip(unsigned char block[64])
{

    unsigned int a = link[0], b = link[1], c = link[2], d = link[3], x
[16];
    for (int i = 0, j = 0; j < 64; ++i, j += 4)
    {
        x[i] = ((unsigned int)block[j]) | (((unsigned int)block[j + 1])
 << 8) | (((unsigned int)block[j + 2]) << 16) | (((unsigned int)block[j
 + 3]) << 24);
    }
    round1(a, b, c, d, x[0], s[0][0], 0xd76aa478);
    round1(d, a, b, c, x[1], s[0][1], 0xe8c7b756);
    round1(c, d, a, b, x[2], s[0][2], 0x242070db);
    round1(b, c, d, a, x[3], s[0][3], 0xc1bdceee);
    round1(a, b, c, d, x[4], s[0][0], 0xf57c0faf);
    round1(d, a, b, c, x[5], s[0][1], 0x4787c62a);
    round1(c, d, a, b, x[6], s[0][2], 0xa8304613);
    round1(b, c, d, a, x[7], s[0][3], 0xfd469501);
    round1(a, b, c, d, x[8], s[0][0], 0x698098d8);
    round1(d, a, b, c, x[9], s[0][1], 0x8b44f7af);
    round1(c, d, a, b, x[10], s[0][2], 0xffff5bb1);
    round1(b, c, d, a, x[11], s[0][3], 0x895cd7be);
    round1(a, b, c, d, x[12], s[0][0], 0x6b901122);
    round1(d, a, b, c, x[13], s[0][1], 0xfd987193);
    round1(c, d, a, b, x[14], s[0][2], 0xa679438e);
    round1(b, c, d, a, x[15], s[0][3], 0x49b40821);

    round2(a, b, c, d, x[1], s[1][0], 0xf61e2562);
    round2(d, a, b, c, x[6], s[1][1], 0xc040b340);
    round2(c, d, a, b, x[11], s[1][2], 0x265e5a51);
    round2(b, c, d, a, x[0], s[1][3], 0xe9b6c7aa);
    round2(a, b, c, d, x[5], s[1][0], 0xd62f105d);
    round2(d, a, b, c, x[10], s[1][1], 0x2441453);
    round2(c, d, a, b, x[15], s[1][2], 0xd8a1e681);
    round2(b, c, d, a, x[4], s[1][3], 0xe7d3fbc8);
    round2(a, b, c, d, x[9], s[1][0], 0x21e1cde6);
    round2(d, a, b, c, x[14], s[1][1], 0xc33707d6);
    round2(c, d, a, b, x[3], s[1][2], 0xf4d50d87);
    round2(b, c, d, a, x[8], s[1][3], 0x455a14ed);
    round2(a, b, c, d, x[13], s[1][0], 0xa9e3e905);
```

```
    round2(d, a, b, c, x[2], s[1][1], 0xfcefa3f8);
    round2(c, d, a, b, x[7], s[1][2], 0x676f02d9);
    round2(b, c, d, a, x[12], s[1][3], 0x8d2a4c8a);

    round3(a, b, c, d, x[5], s[2][0], 0xfffa3942);
    round3(d, a, b, c, x[8], s[2][1], 0x8771f681);
    round3(c, d, a, b, x[11], s[2][2], 0x6d9d6122);
    round3(b, c, d, a, x[14], s[2][3], 0xfde5380c);
    round3(a, b, c, d, x[1], s[2][0], 0xa4beea44);
    round3(d, a, b, c, x[4], s[2][1], 0x4bdecfa9);
    round3(c, d, a, b, x[7], s[2][2], 0xf6bb4b60);
    round3(b, c, d, a, x[10], s[2][3], 0xbebfbc70);
    round3(a, b, c, d, x[13], s[2][0], 0x289b7ec6);
    round3(d, a, b, c, x[0], s[2][1], 0xeaa127fa);
    round3(c, d, a, b, x[3], s[2][2], 0xd4ef3085);
    round3(b, c, d, a, x[6], s[2][3], 0x4881d05);
    round3(a, b, c, d, x[9], s[2][0], 0xd9d4d039);
    round3(d, a, b, c, x[12], s[2][1], 0xe6db99e5);
    round3(c, d, a, b, x[15], s[2][2], 0x1fa27cf8);
    round3(b, c, d, a, x[2], s[2][3], 0xc4ac5665);

    round4(a, b, c, d, x[0], s[3][0], 0xf4292244);
    round4(d, a, b, c, x[7], s[3][1], 0x432aff97);
    round4(c, d, a, b, x[14], s[3][2], 0xab9423a7);
    round4(b, c, d, a, x[5], s[3][3], 0xfc93a039);
    round4(a, b, c, d, x[12], s[3][0], 0x655b59c3);
    round4(d, a, b, c, x[3], s[3][1], 0x8f0ccc92);
    round4(c, d, a, b, x[10], s[3][2], 0xffeff47d);
    round4(b, c, d, a, x[1], s[3][3], 0x85845dd1);
    round4(a, b, c, d, x[8], s[3][0], 0x6fa87e4f);
    round4(d, a, b, c, x[15], s[3][1], 0xfe2ce6e0);
    round4(c, d, a, b, x[6], s[3][2], 0xa3014314);
    round4(b, c, d, a, x[13], s[3][3], 0x4e0811a1);
    round4(a, b, c, d, x[4], s[3][0], 0xf7537e82);
    round4(d, a, b, c, x[11], s[3][1], 0xbd3af235);
    round4(c, d, a, b, x[2], s[3][2], 0x2ad7d2bb);
    round4(b, c, d, a, x[9], s[3][3], 0xeb86d391);

    link[0] += a;
    link[1] += b;
    link[2] += c;
    link[3] += d;
}
```

- 最后输出函数和雪崩函数差不多，这里列出雪崩函数（主要是比较两个明文加密后不同的位数）

```
int avalanche(string& zyl, string& wxn)
{
    bitset<128> zyl_bit, wxn_bit;
```

```
    int num = 0;
    string zyl_MD5 = MD5(zyl).getstring();
    string wxn_MD5 = MD5(wxn).getstring();

    cout << "原始字符串：" << zyl << endl << "改后字符串：" << wxn << end
l;
    cout << "原始结果：0x" << zyl_MD5 << endl << "改后结果：0x" << wxn_MD
5 << endl;

    getbit(zyl_MD5, zyl_bit);
    getbit(wxn_MD5, wxn_bit);

    for (int i = 0; i < 128; i++)
    {
        if (zyl_bit[i] != wxn_bit[i])
        {
            num++;
        }
    }
//    cout << num;
    return num;
}
```

3、结果展示

- 首先是 MD5 的结果：

与测试数据给的结果相同，代表程序正确

- 然后是雪崩检测的结果：

```
================================================================================
这是我的`MD5`加密器，你可以输入以下数字进行相应操作：
0.`MD5`加密
1.检测雪崩
2.退出程序
================================================================================
1
--------------------------------------------------------------------------------
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0xfd75880b0f46060dd792896596bba4b8
不同的数字位数：55
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0x4808cf0f1bf532e58a8104c4f67c17ac
不同的数字位数：41
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0xb1f1b4a081b3bd85380be7aad6790db1
不同的数字位数：40
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0xa52ac5f1e946935ab88bbd1bc161cb9c
不同的数字位数：36
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0x589490233f6eb32f1b04d1bfbf5a20b4
不同的数字位数：45
```

```
不同的数字位数：45
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0x2879d9cfc10072c70bc00a33afbce181
不同的数字位数：48
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0x5c608fc72f3164ee610517fb9286ac34
不同的数字位数：43
原始字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
改后字符串：12345678901234567890123456789012345678901234567890123456789012345678901234567890
原始结果：0x57edf4a22be3c955ac49da2e2107b67a
改后结果：0xd1c1b8762e807bc90b1ed4aa0845ba0b
不同的数字位数：39
改变字符串获得不同的数字位数的平均数为：43.375
--------------------------------------------------------------------------------
================================================================================
这是我的`MD5`加密器，你可以输入以下数字进行相应操作：
0.`MD5`加密
1.检测雪崩
2.退出程序
================================================================================
```

每次仅更改逗号的位置，重复八次，平均每次有 43.375 位发生变化

综上可以看出即使小小的改动，HASH 加密结果也会发生很大的变化

四、总结与展望

1、总结

本次实验主要讲解了哈希函数的基本概念和常见算法，重点介绍了 MD5 哈希算法的原理、实现和应用。通过实验，我们学习到了如何使用 MD5 哈希算法对一段文本进行加密，并用 C++语言实现了该算法。

在实验中，我们也发现了哈希算法的一些优缺点，例如 MD5 算法在安全性和速度上相对较高，但其长度固定可能导致哈希冲突的发生。因此，在实际应用中，需要根据具体情况选择不同的哈希算法，并根据实际情况进行优化。

在未来的学习中，我们将继续深入了解密码学的相关知识，学习更多的哈希算法，并将其应用到实践中，更好地学习和掌握密码学的相关知识和技能。

2、展望

本次实验的结果表明，md5 哈希函数具有可靠的安全性和高效的计算速度，在密码学和网络安全中得到广泛应用。同时，本次实验也展示了如何编写和应用哈希函数的基本方法，增强了我们对哈希函数的理解和应用能力。

在未来，我们可以进一步加强对哈希函数的研究和应用，包括深入了解其算法实现和安全性问题，开发更加智能和高效的哈希函数工具，以提高密码学和网络安全的保障能力。