《应用密码学》实验报告

课程：__应用密码学__ 实验名称：__RSA 签名算法的实现__

姓名：__杨佳伲__ 实验日期：__2024.5.6__

学号：__2022132006__ 实验报告日期：__2024.5.6__

班级：__信安实验 221__

教师评语： 签名： 日期：	成绩：

一、实验名称

RSA 签名算法的实现

二、实验环境（详细说明运行的系统、平台及代码等）

1 平台：VC

2.操作系统：Windows

三、实验目的

（1）加深对数字签名算法的理解；

（2）加深消息摘要函数 SHA-1 的掌握；

（3）提高编程实践能力。

四、实验内容、步骤及结果

1．实验内容

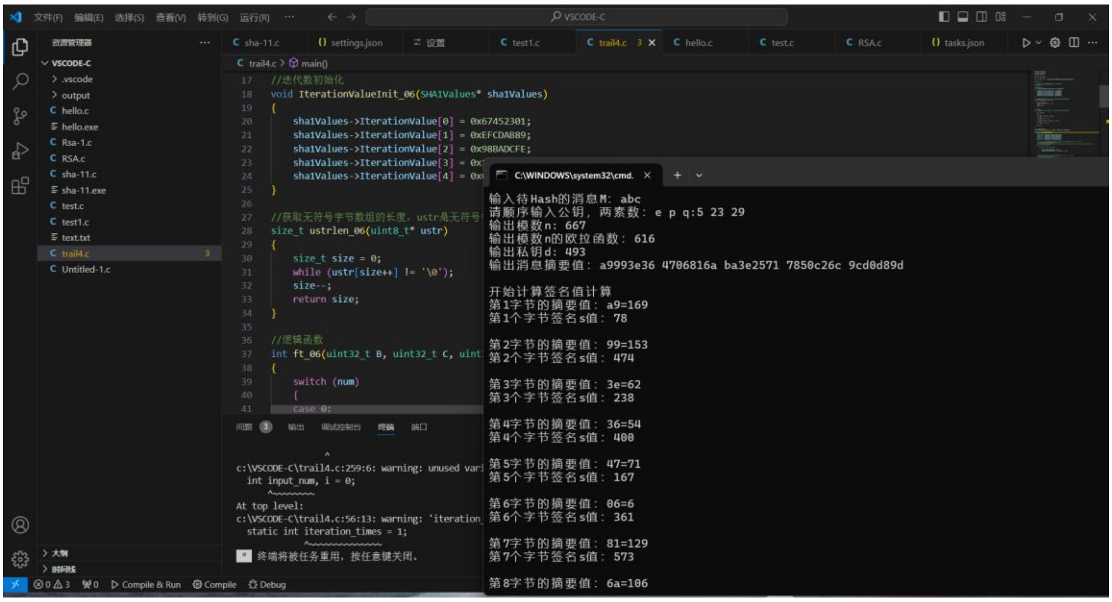
（1）输入字符作为原始消息，采用 SHA-1 算法，输出固定长度 160 比特的消息摘要；

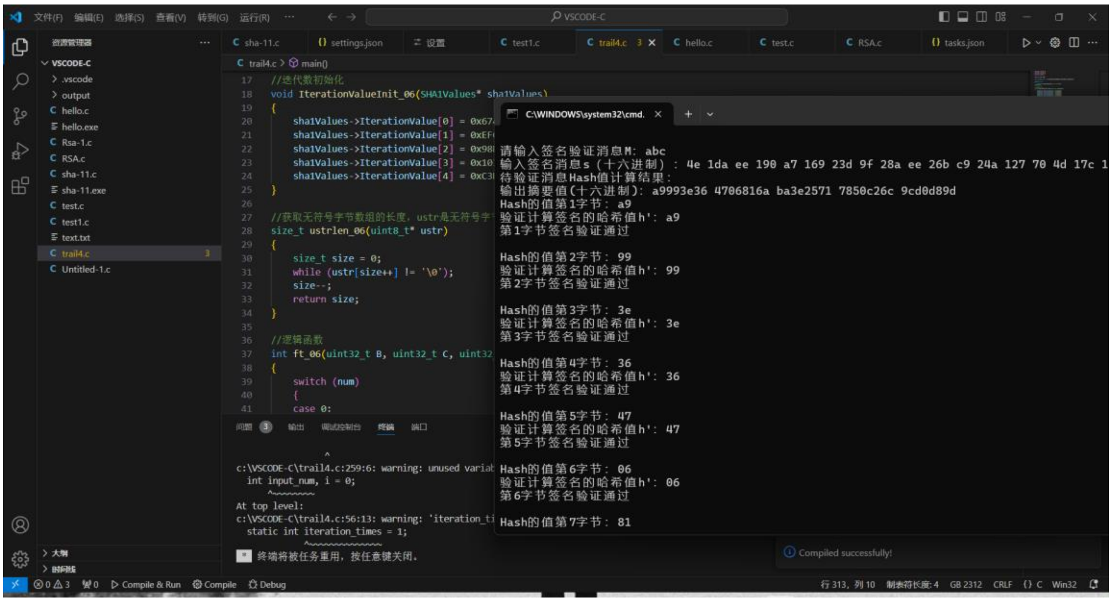
（2）将输出的消息摘要作为输入，采用 RSA 算法的私钥进行签名，得到签名消息；

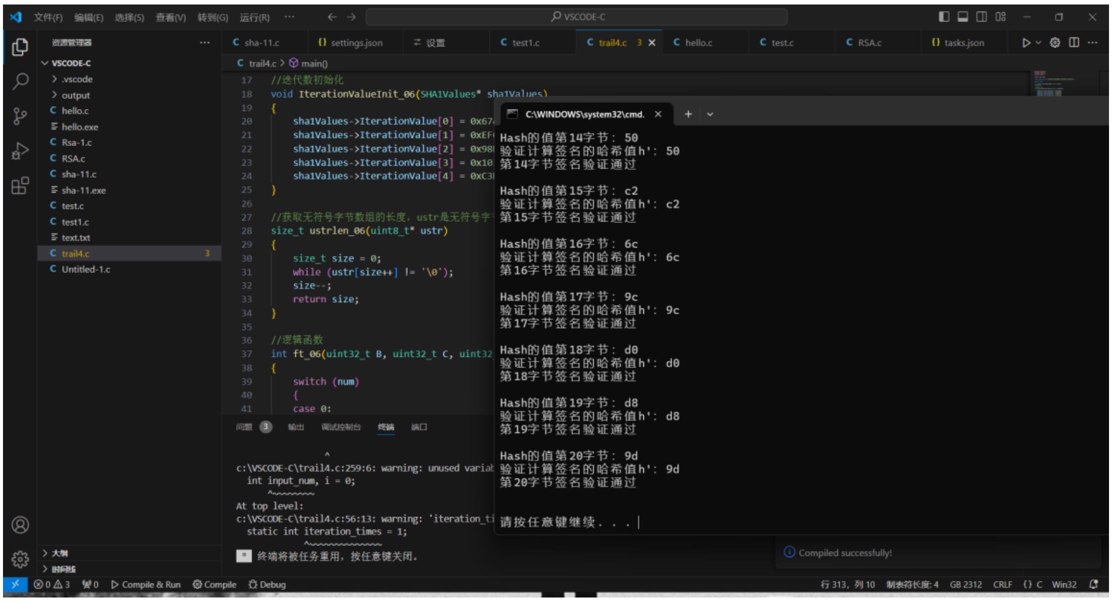
（3）将原始消息和签名消息作为输入，采用 RSA 算法的公钥进行签名验证。

2．实验结果

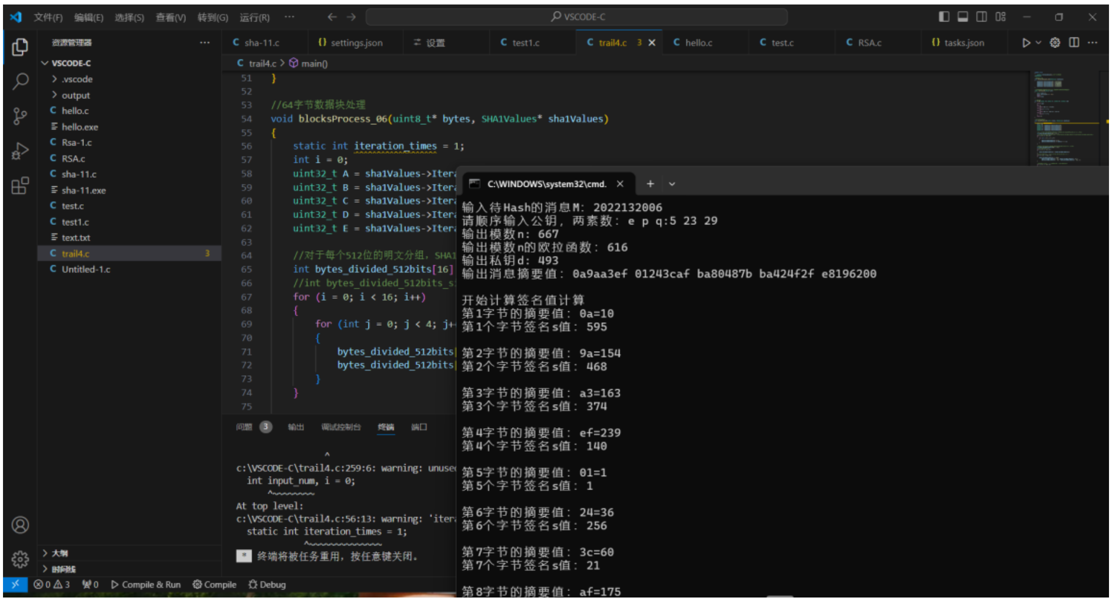
（1)、第 1 组测试要求：1 组固定的数值设置，RSA 算法测试，**e=5**，p=23，q=29；输入为 ASCII 码，程序的默认输入为 FIPS-180-2 中示例的“abc”

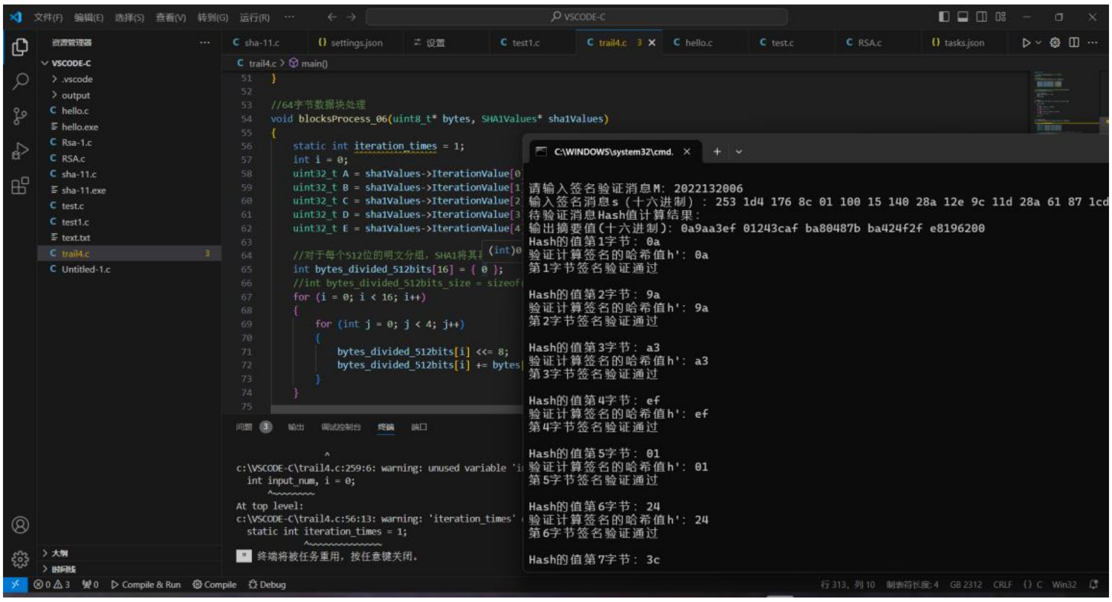


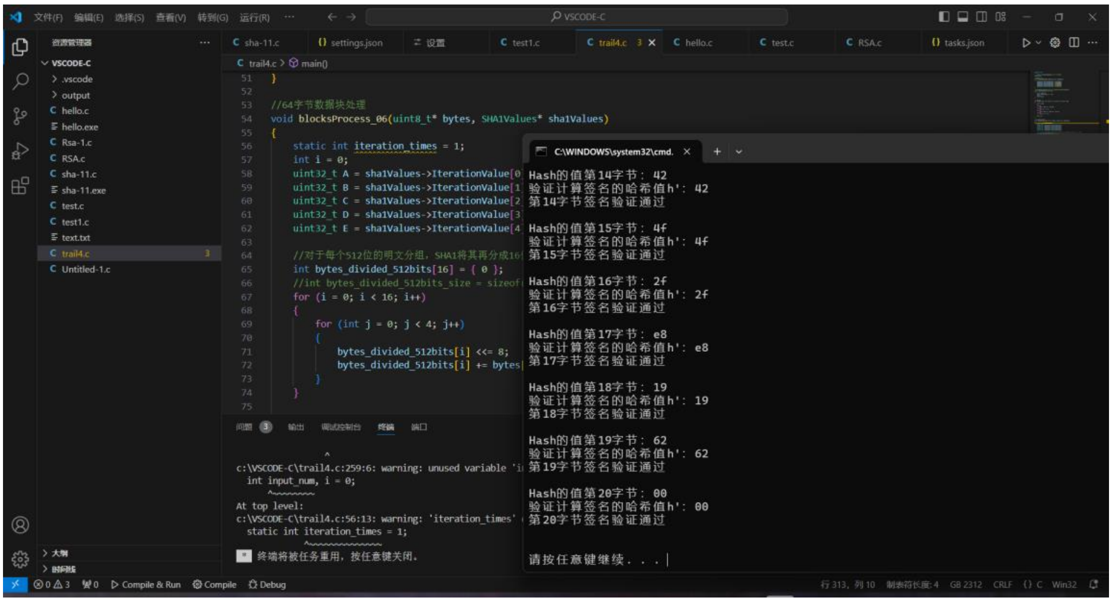



C:\WINDOWS\system32\cmd.
Hash的值第14字节： 50
验证计算签名的哈希值h'： 50
第14字节签名验证通过
Hash的值第15字节： c2
验证计算签名的哈希值h'： c2
第15字节签名验证通过
Hash的值第16字节： 6c
验证计算签名的哈希值h'： 6c
第16字节签名验证通过
Hash的值第17字节： 9c
验证计算签名的哈希值h'： 9c
第17字节签名验证通过
Hash的值第18字节： d0
验证计算签名的哈希值h'： d0
第18字节签名验证通过
Hash的值第19字节： d8
验证计算签名的哈希值h'： d8
第19字节签名验证通过
Hash的值第20字节： 9d
验证计算签名的哈希值h'： 9d
第20字节签名验证通过
请按任意键继续. . .
终端将被任务重用，按任意键关闭。
Compiled successfully!

（2）、


C:\WINDOWS\system32\cmd.
输入待Hash的消息M: 2022132006
请顺序输入公钥，两素数: e p q:5 23 29
输出模数n: 667
输出模数n的欧拉函数: 616
输出私钥d: 493
输出消息摘要值: 0a9aa3ef 01243caf ba80487b ba424f2f e8196200
开始计算签名值计算
第1字节的摘要值: 0a=10
第1个字节签名s值: 595
第2字节的摘要值: 9a=154
第2个字节签名s值: 468
第3字节的摘要值: a3=163
第3个字节签名s值: 374
第4字节的摘要值: ef=239
第4个字节签名s值: 140
第5字节的摘要值: 01=1
第5个字节签名s值: 1
第6字节的摘要值: 24=36
第6个字节签名s值: 256
第7字节的摘要值: 3c=60
第7个字节签名s值: 21
第8字节的摘要值: af=175
终端将被任务重用，按任意键关闭。





五、实验中的问题及心得

对于 RSA 签名算法的实现，在编写程序过程中，核心部分的程序是 RSA 签名算法的逻辑实现（综合了 sha-1 和 rsa 加解密算法），课后自己实现了一遍代码，对 RSA 签名算法的实现有了更深的了解和掌握，也掌握了 RSA 签名算法的实现的规则。

附件：程序代码

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <math.h>

#define CHUNK 1024
```

```
const uint32_t K[] = { 0x5A827999,0x6ED9EBA1,0x8F1BBCDC,0xCA62C1D6 };
```

```
typedef struct
{
    uint32_t IterationValue[5];//存放 5 个迭代数
}SHA1Values;
```

```
//迭代数初始化
void IterationValueInit_06(SHA1Values* sha1Values)
{
    sha1Values->IterationValue[0] = 0x67452301;
    sha1Values->IterationValue[1] = 0xEFCDAB89;
    sha1Values->IterationValue[2] = 0x98BADCFE;
    sha1Values->IterationValue[3] = 0x10325476;
    sha1Values->IterationValue[4] = 0xC3D2E1F0;
}
```

```
//获取无符号字节数组的长度，ustr 是无符号字节数组的首地址指针
size_t ustrlen_06(uint8_t* ustr)
{
    size_t size = 0;
    while (ustr[size++] != '\0');
    size--;
    return size;
}
```

```
//逻辑函数
int ft_06(uint32_t B, uint32_t C, uint32_t D, uint32_t num)
{
    switch (num)
    {
    case 0:
```

```
        return (B & C) | (~B & D);
    case 1:
        return B ^ C ^ D;
    case 2:
        return (B & C) | (B & D) | (C & D);
    case 3:
        return B ^ C ^ D;
    }
    return 0;
}
```

```
//64 字节数据块处理
void blocksProcess_06(uint8_t* bytes, SHA1Values* sha1Values)
{
    static int iteration_times = 1;
    int i = 0;
    uint32_t A = sha1Values->IterationValue[0];
    uint32_t B = sha1Values->IterationValue[1];
    uint32_t C = sha1Values->IterationValue[2];
    uint32_t D = sha1Values->IterationValue[3];
    uint32_t E = sha1Values->IterationValue[4];
```

```
    //对于每个 512 位的明文分组，SHA1 将其再分成 16 份更小的明文分组,M[t](t= 0, 1,……15)
    int bytes_divided_512bits[16] = { 0 };
    //int bytes_divided_512bits_size =
sizeof(bytes_divided_512bits)/sizeof(bytes_divided_512bits[0]);
    for (i = 0; i < 16; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            bytes_divided_512bits[i] <<= 8;
            bytes_divided_512bits[i] += bytes[j + 4 * i];
        }
    }
```

```
    //将这 16 个子明文分组扩充到 80 个子明文分组，记为 W[t](t= 0, 1,……79)
    int bytes_divided2_512bits[80] = { 0 };
    //当 0<t<15 时，Wt = Mt
    for (i = 0; i < 16; i++)
    {
        bytes_divided2_512bits[i] = bytes_divided_512bits[i];
    }
    //当 16<t<79 时，Wt = ( Wt-3 ⊕ Wt-8⊕ Wt-14⊕ Wt-16) <<< 1
    for (i = 16; i < 80; i++)
```

```
    {
        bytes_divided2_512bits[i] = _rotl(
            bytes_divided2_512bits[i - 3] ^ bytes_divided2_512bits[i - 8] ^
            bytes_divided2_512bits[i - 14] ^ bytes_divided2_512bits[i - 16], 1);
    }
```

```
    //80 次迭代，分 4 组进行，每组迭代常数和逻辑函数组合不同
    for (i = 0; i < 80; i++)
    {
        uint32_t temp = _rotl(A, 5) + E + bytes_divided2_512bits[i] + K[i / 20] + ft_06(B, C, D,
i / 20);
        E = D;
        D = C;
        C = _rotl(B, 30);
        B = A;
        A = temp;
    }
    sha1Values->IterationValue[0] += A;
    sha1Values->IterationValue[1] += B;
    sha1Values->IterationValue[2] += C;
    sha1Values->IterationValue[3] += D;
    sha1Values->IterationValue[4] += E;
}
```

```
//对 str 进行 SHA1 摘要计算，并把结果赋给 data
SHA1Values SHA1Encrypt_06(uint8_t* str, SHA1Values sha1Values)
{
    //SHA1Values sha1Values = { 0 };
    IterationValueInit_06(&sha1Values);//对 5 个迭代常数进行初始化
    long str_size = ustrlen_06(str);//获取无符号字符数组的长度
    uint8_t bytes_512bits[64] = { 0 };
    int packets = (int)ceil(str_size * 8.0 / 512);//以 512 比特为单位获取数据的分块个数
    int left_packets = packets;//剩余未处理的分块个数
    long str_index = 0;//数据索引位置
```

```
    while (left_packets > 0)
    {
        //当剩余未处理的分块个数为 1 是执行以下代码
        if (left_packets == 1)
        {
            int j = 0;
```

```
            //按 512 位的长度进行分组
            //要注意处理的特例是 str_size 等于 64，它为 64 时不进入下面的 for 循环
```

```
            if (str_size % 64 != 0)
                for (; j < str_size % 64; j++)
                {
                    bytes_512bits[j] = str[str_index];
                    str_index++;
                }
            else
            {
                for (; j < 64; j++)
                {
                    bytes_512bits[j] = str[str_index];
                    str_index++;
                }
                blocksProcess_06(bytes_512bits, &sha1Values);
                j = 0;
            }
            //将最后一组数据添入数组以后，判断这组数据的长度，如果小于 56，那么直接处理这些数据以后就结束了
            //如果大于等于 56，则说明还需要再多处理一组数据，因为根据算法流程，需要在最后一组数据后先添1，
            //然后补零直至数据长度对 512 求余后为 448，如果最后一组数据长度大于等于 56，添 1 以后补 0 会超过当前
            //64 字节的数据块，所以要多处理一组数据。
            if (j >= 56)
            {
                bytes_512bits[j++] = 0x80;
                for (; j < 64; j++)
                    bytes_512bits[j] = 0x00;
                blocksProcess_06(bytes_512bits, &sha1Values);
                for (j = 0; j < 56; j++)
                    bytes_512bits[j] = 0x00;
            }
            else
            {
                bytes_512bits[j++] = 0x80;
                for (; j < 56; j++)
                    bytes_512bits[j] = 0x00;
            }
            //将有效数据长度用 8 字节表示填充到数据末尾
            for (j = 63; j >= 56; j--)
            {
                long str_size1 = str_size * 8;
                for (int k = 0; k < 63 - j; k++)
                    str_size1 >>= 8;
```

```
                bytes_512bits[j] = str_size1 % 256;
            }
            blocksProcess_06(bytes_512bits, &sha1Values);
        }
        //当数据分块个数超过 1 个是执行以下代码
        else
        {
            //以 512 比特（64 字节）为单位对数组进行赋值
            for (int j = 0; j < 64; j++)
            {
                bytes_512bits[j] = str[str_index];
                str_index++;
            }
            //对 64 字节空间的数组进行处理，将迭代值赋给 sha1Values
            blocksProcess_06(bytes_512bits, &sha1Values);
        }
        left_packets--;
    }
    /*sprintf_s(data,45, "%08x %08x %08x %08x %08x",
        sha1Values.IterationValue[0],
        sha1Values.IterationValue[1],
        sha1Values.IterationValue[2],
        sha1Values.IterationValue[3],
        sha1Values.IterationValue[4]);
    printf("消息摘要为: %s\n", data);*/
    return sha1Values;
}
```

```
//接收用户输入，为防止溢出，给指针动态分配内存
char* readinput_06()
{
    char* old_input = NULL;
    char* new_input = NULL;
    char tempbuf[CHUNK];
    size_t inputlen = 0, templen = 0;
    rewind(stdin);
    do {
        fgets(tempbuf, CHUNK, stdin);
        templen = strlen(tempbuf);
        new_input = (char*)realloc(old_input, inputlen + templen + 1);
        if (new_input == NULL)
        {
            return NULL;
        }
```

```c
        old_input = new_input;
        memcpy(new_input + inputlen, tempbuf, templen + 1);
        inputlen += templen;
    } while (templen == CHUNK - 1 && tempbuf[CHUNK - 2] != '\n');
    new_input[strlen(new_input) - 1] = '\0';
    return new_input;
}
```

```c
// 模幂运算
uint32_t mod_exp_06(uint32_t base, uint32_t exponent, uint32_t mod) {
    uint32_t result = 1;
    base = base % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1)
            result = (result * base) % mod;
```

```c
        base = (base * base) % mod;
        exponent >>= 1;
    }
    return (uint32_t)result;
}
```

```c
// RSA 数字签名
void rsa_sign_06(unsigned int* c, uint32_t d, uint32_t n, uint32_t* signature, unsigned int* byte)
{
    for (int i = 0; i < 5; i++) {
        // 提取 c[i] 中的四个字节
        byte[0 + i * 4] = (c[i] >> 24) & 0xFF;  // 获取高 8 位
        byte[1 + i * 4] = (c[i] >> 16) & 0xFF;  // 获取次高 8 位
        byte[2 + i * 4] = (c[i] >> 8) & 0xFF;   // 获取次低 8 位
        byte[3 + i * 4] = c[i] & 0xFF;          // 获取低 8 位
```

```c
        for (int j = 0; j < 4; j++) {
            signature[j + i * 4] = mod_exp_06(byte[j + i * 4], d, n);
        }
    }
}
```

```c
void rsa_verify_06(uint32_t* signature, uint32_t e, uint32_t n, unsigned int* verify) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++) {
            verify[j + i * 4] = mod_exp_06(signature[j + i * 4], e, n);
        }
    }
```

```
}
```

```
int main() {
    int input_num, i = 0;
    uint8_t* str_1 = NULL;
    uint8_t* str_2 = NULL;
    SHA1Values sha1Values_1 = { 0 };
    SHA1Values sha1Values_2 = { 0 };
    printf("输入待 Hash 的消息 M: ");
    //读取输入的数据，并将最终结果的字符首地址赋给 str
    str_1 = (uint8_t*)readinput_06();
```

```
    // 输入公钥，两素数: e, p, q
    uint32_t e, p, q;
    printf("请顺序输入公钥，两素数: e p q:");
    scanf("%u %u %u", &e, &p, &q);
```

```
    // 计算模数 n 和私钥 d
    uint32_t n = p * q;
    uint32_t phi = (p - 1) * (q - 1);
    uint32_t d = 0;
    while ((d * e) % phi != 1) {
        d++;
    }
```

```
    // 输出模数 n, 欧拉函数, 私钥 d
    printf("输出模数 n: %u\n", n);
    printf("输出模数 n 的欧拉函数: %u\n", phi);
    printf("输出私钥 d: %u\n", d);
```

```
    // 计算消息摘要
    sha1Values_1 = SHA1Encrypt_06(str_1, sha1Values_1);
    printf("输出消息摘要值: %08x %08x %08x %08x %08x\n", sha1Values_1.IterationValue[0],
sha1Values_1.IterationValue[1], sha1Values_1.IterationValue[2], sha1Values_1.IterationValue[3],
sha1Values_1.IterationValue[4]);
    printf("\n 开始计算签名值计算\n");
```

```
    // RSA 数字签名
    uint32_t signature[20];
    unsigned int byte[20] = { 0 };
    rsa_sign_06(sha1Values_1.IterationValue, d, n, signature, byte);
```

```
    // 输出签名值
    for (int i = 0; i <= 4; i++) {
```

```
        for (int j = 0; j <= 3; j++) {
            printf("第%d 字节的摘要值：%02x=%d\n", j + 1 + i * 4, byte[j + i * 4], byte[j + i * 4]);
            printf("第%d 个字节签名 s 值：%u\n", j + 1 + i * 4, signature[j + i * 4]);
            printf("\n");
        }
    }
```

```
    // 验证签名
    printf("请输入签名验证消息 M：");
    str_2 = (uint8_t*)readinput_06();
```

```
    printf("输入签名消息 s（十六进制）：");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j <= 3; j++) {
            printf("%02x ", signature[j + i * 4]);
        }
    }
    printf("\n");
```

```
    // 哈希验证
    unsigned int verify[20] = { 0 };
    rsa_verify_06(signature, e, n, verify);
```

```
    printf("待验证消息 Hash 值计算结果：\n");
    // 计算消息摘要
    sha1Values_2 = SHA1Encrypt_06(str_2, sha1Values_2);
    printf("输出摘要值(十六进制)：%08x %08x %08x %08x %08x\n", sha1Values_2.IterationValue[0],
sha1Values_2.IterationValue[1], sha1Values_2.IterationValue[2], sha1Values_2.IterationValue[3],
sha1Values_2.IterationValue[4]);
```

```
    for (int i = 0; i <= 4; i++) {
        // 提取 sha1Values_2.IterationValue[i] 中的四个字节
        byte[0 + i * 4] = (sha1Values_2.IterationValue[i] >> 24) & 0xFF;  // 获取高 8 位
        byte[1 + i * 4] = (sha1Values_2.IterationValue[i] >> 16) & 0xFF;  // 获取次高 8 位
        byte[2 + i * 4] = (sha1Values_2.IterationValue[i] >> 8) & 0xFF;   // 获取次低 8 位
        byte[3 + i * 4] = sha1Values_2.IterationValue[i] & 0xFF;          // 获取低 8 位
        for (int j = 0; j <= 3; j++) {
            printf("Hash 的值第%d 字节：%02x\n", j + 1 + i * 4, byte[j + i * 4]);
            printf("验证计算签名的哈希值 h'：%02x\n", verify[j + i * 4]);
            if (verify[j + i * 4] != byte[j + i * 4]) {
                printf("签名验证失败\n");
                return 0;
            }
            else {
```

```
                printf("第%d 字节签名验证通过\n", j + 1 + i * 4);
                printf("\n");
            }
        }
    }
```

```
    free(str_1);
    free(str_2);
```

```
    return 0;
}
```