

# 西安电子科技大学

## 物联网安全实验课程 实验报告

### 实验名称 DES 加密算法

物联网工程 1803041 班

姓名 魏红旭 学号 18030400014

同作者

实验日期 2021 年 5 月 25 日

成 绩

指导教师评语：

指导教师：

年月日

#### 实验报告内容基本要求及参考格式

- 一、实验目的
- 二、实验所用仪器（或实验环境）
- 三、实验基本原理及步骤（或方案设计及理论计算）
- 四、实验数据记录（或仿真及软件设计）
- 五、实验结果分析及回答问题（或测试环境及测试结果）

## 一、实验目的:

编程实现 DES 算法, 深入理解 DES 加密解密原理

## 二、实验所用仪器 (或实验环境)

计算机科学与技术学院实验中心, 可接入 Internet 网台式机 44 台。

## 三、实验基本原理及要求

### 实验原理:

第一阶段: 对输入的 64 位的明文分组进行固定的“初始置换” (Initial Permutation, IP), 即按固定的规则重新排列明文分组的 64 位二进制数据, 再重排后的 64 位数据前后 32 位分为独立的左右两个部分, 前 32 位记为  $L_0$ , 后 32 位记为  $R_0$ 。初始置换写为:

$$(L_0, R_0) \leftarrow IP(64 \text{ 位分组明文})$$

第二阶段: 16 轮相同函数迭代。将上一轮输出的  $R_{i-1}$  直接作为  $L_i$  输入, 同时将  $R_{i-1}$  与第  $i$  个 48 位的子密钥  $k_i$  经“轮函数  $f$ ”转换后, 得到一个 32 位的中间结果, 再将此中间结果与上一轮的  $L_{i-1}$  做异或运算, 并将得到的新的 32 位结果作为下一轮的  $R_i$ 。如此往复, 迭代处理 16 次。每次的子密钥不同, 16 个子密钥的生成与轮函数  $f$ , 后面单独阐述。可以将这一过程写为:

$$L_i \leftarrow R_{i-1}, R_i \leftarrow L_{i-1} \oplus f(R_{i-1}, k_i)$$

第三阶段: 将第 16 轮迭代结果左右两半组  $L_{16}, R_{16}$  直接合并为 64 位 ( $L_{16}, R_{16}$ ), 输入到初始逆置换来消除初始置换的影响。这一步的输出结果即为加密过程的密文。可将这一过程写为: 输出 64 位密文  $\leftarrow IP^{-1}(L_{16}, R_{16})$

### 实验要求:

以每个学生的学号为明文, 利用 DES 算法对其加密, 给加密结果, 其中密钥取: 23A4Z77995BC0FF。

## 四、实验步骤及实验数据记录: (要有文字描述和必要截图)

- 首先说明对于密钥的处理: 实验要求密钥取 23A4Z77995BC0FF, 但是由于密钥中存在大于 F 的字母, 所以程序中规定, 对于介于 (F, Z] 之间的字母, 均按照 F 处理 (不区分大小写); 学号为 18030400014, 因为明文需补齐 16 位, 所以输出的明文为 1803040001400000。

1. 首先将 16 位十六进制密钥转换成 64 位二进制密钥, 64 位的秘钥首先根据表格 PC-1 进行变换, 原秘钥中只有 56 位会进入新秘钥, PC-1 表也只有 56 个元素, 然后, 将这个秘钥拆分为左右两部分,  $C0$  和  $D0$ , 每半边都有 28 位。相关代码如下图所示:

- 1) Main 函数中整体代码:

```

string text="1803040001400000";
string key="23A4F77995BC0FF1";
string cipher_text[16];

int Result_Cipher_Text[16][64],middle_result[8][8],Combine_LR[64];

cout << "Text: " << text << endl;
cout << "Key: " << key << endl;

int** Target_text=new int*[8],**Target_key=new int*[8],**SubKey=new int*[8];
for (int i = 0; i < 8; ++i) {
    *(Target_text+i)=new int[8];
    *(Target_key+i)=new int[8];
    *(SubKey+i)=new int[7];
}

//将明文转换成二进制，并将64位二进制结果存成8*8的矩阵
Target_text=HexToBinary(text);

//将密钥转换成二进制，并将64位二进制结果存成8*8的矩阵
Target_key=HexToBinary(key);

//将64位密钥根据表格PC-1进行变换，得到56位新密钥K+，再分为C0,D0
SubKey=Create_SubKey(Target_key);

```

## 2) 十六进制转换成二进制代码：

```

//十六进制字符串，转换成二进制数组
int** HexToBinary(string original) {
    int x[original.size()],middle_data[4*original.size()],num=0;
    int** result=new int*[8];
    for (int i = 0; i < 8; ++i)
        *(result+i)=new int[8];
    for (int i = 0; i < original.size(); ++i) {
        if(original[i]>'F'&&original[i]<='Z') original[i]='F';
        else if(original[i]>'f'&&original[i]<='z') original[i]='f';
        if(original[i]>='0'&&original[i]<='9'){
            x[i]=original[i]-48;    //0的ASCII为48
        }else if(original[i]>='A'&&original[i]<='F'){
            x[i]=original[i]-55;    //A的ASCII为65, 减10
        }else if(original[i]>='a'&&original[i]<='f'){
            x[i]=original[i]-87;    //a的ASCII为97, 减10
        }
        middle_data[4*i] = x[i] / 8;
        middle_data[4*i+1] = (x[i] - middle_data[4*i] * 8) / 4;
        middle_data[4*i+2] = (x[i] - middle_data[4*i] * 8 - middle_data[4*i+1] * 4) / 2;
        middle_data[4*i+3] = x[i] - middle_data[4*i] * 8 - middle_data[4*i+1] * 4 - middle_data[4*i+2] * 2;
    }
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            result[i][j]= middle_data[num++];
    return result;
}

```

3) PC-1 变换产生第一轮 56 位密钥，并分成 C0, D0 代码:

```
//通过PC-1变换产生第一轮密钥 (64位密钥->56位), 同时生成C0/D0
int** Create_SubKey(int** key) {
    int** SubKey=new int*[8];
    for (int i = 0; i < 8; ++i)
        *(SubKey+i)=new int[7];
    int middle_data[64], num=0, C_num=0, D_num=0;
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            middle_data[num++]=key[i][j];
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 7; ++j) {
            SubKey[i][j]=middle_data[PC_1[i][j] - 1];
            if(C_num<28) C[C_num++]=SubKey[i][j]; //56位分为28+28, 分别存到C/D中
            else D[D_num++]=SubKey[i][j];
        }
    }
    return SubKey;
}
```

2. 现在已经得到 C0, D0, 我们现在创建 16 个块  $C_n$  和  $D_n$ ,  $1 \leq n \leq 16$ 。每一对  $C_n$  和  $D_n$  都是由前一对  $C_{n-1}$  和  $D_{n-1}$  移位而来。具体说来, 对于  $n = 1, 2, \dots, 16$ , 在前一轮移位的结果上, 使用左移表进行一些次数的左移操作, 就是将除第一位外的所有位往左移一位, 将第一位移动至最后一位。例如  $C_3$  and  $D_3$  是  $C_2$  and  $D_2$  移位而来,  $C_{16}$  and  $D_{16}$  则是由  $C_{15}$  and  $D_{15}$  通过左移得到。代码如下图所示:

```
for (int i = 1; i <= 16; ++i) {
    //根据C0, D0J进行16次循环移位, 依次得到C1-C16, D1-D16
    Cycle_SubKey( n: i-1);
}
```

```

//循环产生子密钥, C1-C16, D1-D16, 由C0、D0推导
void Cycle_SubKey(int n) {
    int times=Left_Shift[n], C0, D0;
    for (int i = 0; i < times; ++i) {
        C0=C[0];
        D0=D[0];
        for (int j = 0; j < 28; ++j) {
            C[j - 1] = C[j];
            D[j - 1] = D[j];
        }
        C[27] = C0;
        D[27] = D0;
    }
}

```

3. 由 C1-C16, D1-D16 可以得到所有新密钥  $K_n$  ( $1 \leq n \leq 16$ )，对每对拼合后的子密钥  $C_nD_n$ ，按表 PC-2 执行变换：每对子密钥有 56 位，但 PC-2 仅仅使用其中的 48 位，所以最终所得到新密钥  $K_n$  都将转换成 48 位，代码如下图所示：

```

for (int i = 1; i <= 16; ++i) {
    //根据C0, D0进行16次循环移位, 依次得到C1-C16, D1-D16
    Cycle_SubKey( n: i-1);
    //对子密钥进行拼合, 形成CnDn, 并按照PC-2执行变换, 得到48位密钥, 循环16次, 得到K1-K16
    TurnInto_NewKey( n: i-1);
} //子密钥处理告一段落

```

```

//通过PC-2, 使56位子密钥变为48位, 并记录
void TurnInto_NewKey(int n) {
    int** NewSubKey=new int*[8];
    for (int i = 0; i < 8; ++i)
        *(NewSubKey+i)=new int[6];
    int NowCD[56],NewKey[48],num=0;
    for (int i = 0; i < 56; ++i) {
        if(i<28) NowCD[i]=C[i];
        else NowCD[i]=D[i-28];
    }
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 6; ++j) {
            NewKey[num++]=NowCD[PC_2[i][j]-1];
            SubKey_ByPC2[15-n][num-1]=NewKey[num-1];
        }
    }
}

```

4. 下面我们开始对明文进行处理,对于明文数据,我们 IP 表进行初次变换,然后将变换结果分为 32 位的左半边 L0 和 32 位的右半边 R0,代码如下图所示:

```

} //子密钥处理告一段落
//对明文数据通过IP进行变换, 并将64位二进制数据分为L和R两部分, 各32
IP_Transform(Target_text);

```

```

//通过IP变换初始化L, R
void IP_Transform(int** original) {
    int ori_data[64],num=0,middle_data[64];
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 8; ++j)
            ori_data[num++]=original[i][j];
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            middle_data[8*i+j]=ori_data[IP[i][j]-1];
            if(8*i+j<32) L[8*i+j]=middle_data[8*i+j];
            else R[8*i+j-32]=middle_data[8*i+j];
        }
    }
}

```

5. 我们接着执行 16 个迭代，对  $1 \leq n \leq 16$ ，使用一个函数  $f$ ，函数  $f$  输入两个区块：一个 32 位的数据区块和一个 48 位的密钥区块  $K_n$ ；输出一个 32 位的区块。定义  $+$  表示异或 XOR，那么让  $n$  从 1 循环到 16，我们计算  $L_n = R_{n-1}$ ； $R_n = L_{n-1} + f(R_{n-1}, K_n)$ ；这样我们就得到最终区块，也就是  $n=16$  的  $L_{16}R_{16}$ 。

```
for (int i = 1; i <= 16; ++i) {  
    /*通次此函数进行16次迭代后得到L1-L16, R1-R16  
    *  $L_n = R_{n-1}$  ;  
    *  $R_n = L_{n-1} + F(R_{n-1}, K_n)$ ;  
    */  
    Update_LR(i, flag: 0);  
}  
  
/*通次此函数进行16次迭代后得到L1-L16, R1-R16  
*  $L_n = R_{n-1}$  ;  $R_n = L_{n-1} + F(R_{n-1}, K_n)$ ; */  
void Update_LR(int n, int flag) {  
    int middle_L[32], middle_R[32];  
    for (int i = 0; i < 32; ++i) {  
        middle_L[i] = L[i];  
        middle_R[i] = R[i];  
    }  
    int* result = new int[32];  
    //在F函数中实现F(Rn-1, Kn)  
    result = F(n, flag);  
    for (int i = 0; i < 32; ++i) {  
        L[i] = middle_R[i];  
        R[i] = (result[i] + middle_L[i]) % 2;  
    }  
    /*在第16个迭代之后，我们有了区块L16 and R16。  
    * 接着我们逆转两个区块的顺序得到一个64位的区块*/  
    if (n == 16) {  
        for (int i = 0; i < 32; i++) {  
            int temp = L[i];  
            L[i] = R[i];  
            R[i] = temp;  
        }  
    }  
}
```

6. 现在介绍 f 函数工作方式：

计算 f，我们首先拓展每个  $R_{n-1}$ ，将其从 32 位拓展到 48 位。这是通过使用 E 表来重复  $R_{n-1}$  中的一些位来实现的，也就是说函数  $E(R_{n-1})$  输入 32 位输出 48 位。接着在 f 函数中，我们对输出  $E(R_{n-1})$  和密钥  $K_n$  执行 XOR 运算： $K_n + E(R_{n-1})$ ，f 函数此部分代码如下图所示：

```
/*为了计算F，我们首先拓展每个Rn-1，将其从32位拓展到48位。这是通过使用表E
 * 来重复Rn-1中的一些位来实现。也就是说函数E(Rn-1)输入32位输出48位。*/
int* F(int n, int flag) {
    int *result=new int[32],** B=new int*[8];
    for (int i = 0; i < 8; ++i)
        *(B+i)=new int[6];
    int middle_ER[48],NowK[48],a[48],first,second,middle_all_s[8],result_s[32];
    for (int i = 0; i < 48; ++i) {
        middle_ER[i]=R[E_Table[i]-1]; //使用表E扩展位数
        NowK[i]=SubKey_ByPC2[16-n][i]; //取原先记录的子密钥
    }
    //逆变换，为了实现解密过程准备
    if(flag==1){
        for (int i = 0; i < 48; ++i) {
            NowK[i]=SubKey_ByPC2[n-1][i];
        }
    }
    for (int i = 0; i < 48; ++i)
        a[i]=(NowK[i]+middle_ER[i])%2; //进行XOR运算
    for (int i = 0; i < 8; ++i)
        for (int j = 0; j < 6; ++j) //将8组6比特数据记录在B中
            B[i][j]=a[6*i+j];
}
```

7. 通过 S 盒进行计算，8 组 6 比特的数据被转换为 8 组 4 比特（一共 32 位）的数据，得到结果之后进行 P 变换，代码如下图所示：

```
/*B的第一位和最后一位组合起来的二进制数决定一个介于0和3之间的十进制数（或者二进制00到11之间）。
 * 设这个数为i。B的中间4位二进制数代表一个介于0到15之间的十进制数（二进制0000到1111）。
 * 设这个数为j。查表找到第i行第j列的那个数*/
for (int i = 0; i < 8; ++i) {
    first=B[i][0]*2+B[i][5]; //将8组数据输入S函数
    second=B[i][1]*8+B[i][2]*4+B[i][3]*2+B[i][4];
    middle_all_s[i]=S_Box[i][first][second];
}
```



```

for (int i = 0; i < 8; ++i) {
    result_s[i*4]=middle_all_s[i]/8;
    result_s[i*4+1]=(middle_all_s[i]-result_s[i*4]*8)/4;
    result_s[i*4+2]=(middle_all_s[i]-result_s[i*4]*8-result_s[i*4+1]*4)/2;
    result_s[i*4+3]=middle_all_s[i]-result_s[i*4]*8-result_s[i*4+1]*4-result_s[i*4+2]*2;
}
//对S盒的输出转换成二进制之后，进行P变换，得到32位数据
for (int i = 0; i < 32; ++i)
    result[i]=result_s[P[i]-1];
return result;
}

```

8. 一直完成 16 个迭代。在第 16 个迭代之后，我们有了区块 L16 和 R16。接着我们逆转两个区块的顺序得到一个 64 位的区块，然后对其执行一个最终的变换 IP-1，然后写成 16 进制的结果即为明文对应的密文。

```

/*在第16个迭代之后，我们有了区块L16 and R16。
 * 接着我们逆转两个区块的顺序得到一个64位的区块*/
if(n==16){
    for (int i = 0; i < 32; i++) {
        int temp = L[i];
        L[i] = R[i];
        R[i] = temp;
    }
}
}

```

### ● 解密过程:

解密就是加密的反过程，执行上述步骤，只是中间存在一步子密钥顺序的调换，在 16 轮迭代中，调转左右子密钥的位置，代码如下图所示：

//实现DES的解密过程，只是中间存在一步子密钥顺序的调换，其他相同

```
void Decode(string original){  
    int** middle=new int*[8];  
    int result[64],Combine_LR[64];  
    for (int i = 0; i < 8; ++i)  
        middle[i]=new int[8];  
    middle=HexToBinary(original);  
    IP_Transform(middle);  
    for (int i = 1; i <= 16; ++i) {  
        Update_LR(i, flag: 1);  
    }  
    for (int j = 0; j < 64; ++j) {  
        if(j<32) Combine_LR[j]=L[j];  
        else Combine_LR[j]=R[j-32];  
    }  
    for (int j = 0; j < 8; ++j) {  
        for (int k = 0; k < 8; ++k) {  
            result[j*8+k]=Combine_LR[IP_Inverse[j][k]-1];  
        }  
    }  
    string Final_result=BinaryToHex(result);  
    cout<<"Decryption result: "<<Final_result<<endl;  
}
```

## 五、实验结果分析及实验总结与体会

### ● 实验结果截图：

```
F:\Clion file\IOTESecurity_first\cmake-build-debug\IOTESecurity_first.exe
Text: 1803040001400000
Key: 23A4F77995BC0FF1
----- 1 -----
Cipher Text: AC89A88A00828208
----- 2 -----
Cipher Text: D6E67445084BC1AC
----- 3 -----
Cipher Text: 615B9A8A8C2740DE
----- 4 -----
Cipher Text: 922FEF4DCC198A47
----- 5 -----
Cipher Text: 49BF7786E6A4EDA1
----- 6 -----
Cipher Text: 045F9341D378F478
----- 7 -----
Cipher Text: 28876B886B1CDA94
----- 8 -----
Cipher Text: 1CCB9D4C37A445E2
----- 9 -----
Cipher Text: 2ECFE48EB97A0879
----- 10 -----
Cipher Text: 15C5524F5E1D04BC
----- 11 -----
Cipher Text: 00E281A7A506205C
----- 12 -----
Cipher Text: 2873E859D2A3922C
----- 13 -----
Cipher Text: 36B9FC84C3F9C1B6
----- 14 -----
Cipher Text: 9BF4FC4A69F64259
----- 15 -----
Cipher Text: 4D7274A79EF1010E
----- 16 -----
Cipher Text: 1C7374F38BF4414A
Decryption result: 1803040001400000
-
```

### ● 结果分析：

实验要求输出 16 个子密钥，所以我们在每轮迭代过后我们需要对当前子密钥进行二进制→十六进制转换并输出密钥，在 16 轮迭代结束之后输出最终的密钥：即为 1C7374F38BF4414A，然后我们对此密钥进行解密，解密后的结果为：1803040001400000，和我们输入的明文一致，表示加解密过程正常实现。

## ● 实验总结:

本次实验通过对 DES 算法的研究,我初步了解了数据加密和解密的过程,虽然 DES 算法的过程和代码看着复杂,但是其原理并不难,实际上就是变化的过程较多、较为复杂。DES 使用一个 56 位的密钥以及附加的 8 位奇偶校验码,产生最大 64 位的分组,这是一个迭代的分组密码,使用了 Feistel 技术,将加密的文本分成两半,使用子密钥对其中一般应用循环功能,然后输出与另一半进行“异或”运算,接着交换这两半,此过程循环,最后一个循环不交换。

## 六、源代码

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4. /*-----变换表初始化-----*/
5. //---S 盒, 8 组 6bit
6. int S_Box[8][4][16]={
7.     {
8.         {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7},
9.         {0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8},
10.        {4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0},
11.        {15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}
12.    },
13.    {
14.        {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10},
15.        {3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5},
16.        {0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15},
17.        {13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9}
18.    },
19.    {
20.        {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8},
21.        {13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1},
22.        {13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7},
23.        {1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}
24.    },
25.    {
26.        {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15},
27.        {13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9},
28.        {10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4},
29.        {3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}
30.    },
31.    {
```

```

32.          {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9},
33.          {14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6},
34.          {4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14},
35.          {11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}
36.      },
37.      {
38.          {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11},
39.          {10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8},
40.          {9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6},
41.          {4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}
42.      },
43.      {
44.          {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1},
45.          {13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6},
46.          {1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2},
47.          {6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12}
48.      },
49.      {
50.          {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7},
51.          {1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2},
52.          {7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8},
53.          {2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}
54.      }
55. };

56. //E 扩展，拓展每个 Rn-1，将其从 32 位拓展到 48 位，通过此表来重复实现
57. int E_Table[48]={
58.     32,1,2,3,4,5,4,5,6,7,8,9,
59.     8,9, 10, 11,12,13,12,13,14,15,16,17,
60.     16,17,18,19,20,21,20,21,22,23,24,25,
61.     24,25,26,27,28,29,28,29,30,31,32,1
62. };

63. //对于明文数据 M，通过 IP 进行重新变换
64. int IP[8][8]={
65.     {58,50,42,34,26,18,10,2},
66.     {60,52,44,36,28,20,12,4},
67.     {62,54,46,38,30,22,14,6},
68.     {64,56,48,40,32,24,16,8},
69.     {57,49,41,33,25,17,9, 1},
70.     {59,51,43,35,27,19,11,3},
71.     {61,53,45,37,29,21,13,5},
72.     {63,55,47,39,31,23,15,7}
73. };

74. //密文逆置换
75. int IP_Inverse[8][8]={

```

```

76.         {40,8,48,16,56,24,64,32},
77.         {39,7,47,15,55,23,63,31},
78.         {38,6,46,14,54,22,62,30},
79.         {37,5,45,13,53,21,61,29},
80.         {36,4,44,12,52,20,60,28},
81.         {35,3,43,11,51,19,59,27},
82.         {34,2,42,10,50,18,58,26},
83.         {33,1,41, 9,49,17,57,25}
84. };
85. //64->56 选择, 得到 k+, 分割为 C\D
86. int PC_1[8][7]={
87.     {57,49,41,33,25,17, 9},
88.     {1,58,50, 42,34,26,18},
89.     {10, 2,59,51,43,35,27},
90.     {19,11, 3,60,52,44,36},
91.     {63,55,47,39,31,23,15},
92.     {7,62,54, 46,38,30,22},
93.     {14, 6,61,53,45,37,29},
94.     {21,13, 5,28,20,12, 4}
95. };
96. //密钥置换, 56->48, 使得 CD 组合变为新密钥 Kn
97. int PC_2[8][6]={
98.     {14,17,11,24,1,5},
99.     {3,28,15,6,21,10},
100.    {23,19,12,4,26,8},
101.    {16,7,27,20,13,2},
102.    {41,52,31,37,47,55},
103.    {30,40,51,45,33,48},
104.    {44,49,39,56,34,53},
105.    {46,42,50,36,29,32}
106. };
107. //P 置换及左移操作
108. int P[32] = {16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,2,8,24,14,32,27,3
    ,9,19,13,30,6,22,11,4,25};
109. int Left_Shift[16]={1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};
110.
111. /*-----全局参数声明-----
    ----*/
112. int L[32]={0},R[32]={0},C[28]={0},D[28]={0};
113. int SubKey_ByPC2[16][48]={0};
114.
115. /*-----函数部分-----
    ----*/
116. //十六进制字符串, 转换成二进制数组

```

```

117. int** HexToBinary(string original){
118.     int x[original.size()],middle_data[4*original.size()],num=0;
119.     int** result=new int*[8];
120.     for (int i = 0; i < 8; ++i)
121.         *(result+i)=new int[8];
122.     for (int i = 0; i < original.size(); ++i) {
123.         if(original[i]>'F'&&original[i]<='Z') original[i]='F';
124.         else if(original[i]>'f'&&original[i]<='z') original[i]='f';
125.         if(original[i]>='0'&&original[i]<='9'){
126.             x[i]=original[i]-48;    //0 的 ASCLL 为 48
127.         }else if(original[i]>='A'&&original[i]<='F'){
128.             x[i]=original[i]-55;    //A 的 ASCLL 为 65, 减 10
129.         }else if(original[i]>='a'&&original[i]<='f'){
130.             x[i]=original[i]-87;    //a 的 ASCLL 为 97, 减 10
131.         }
132.         middle_data[4*i] = x[i] / 8;
133.         middle_data[4*i+1] = (x[i] - middle_data[4*i] * 8) / 4;
134.         middle_data[4*i+2] = (x[i] - middle_data[4*i] * 8 - middle_data[4*i+
            +1] * 4) / 2;
135.         middle_data[4*i+3] = x[i] - middle_data[4*i] * 8 - middle_data[4*i+
            1] * 4 - middle_data[4*i+2] * 2;
136.     }
137.     for (int i = 0; i < 8; ++i)
138.         for (int j = 0; j < 8; ++j)
139.             result[i][j]= middle_data[num++];
140.     return result;
141. }
142. //-----二进制转化成十六进制
143. string BinaryToHex(int a[]){
144.     string result;
145.     result.resize(16);
146.     for (int i = 0,j=0; i < 64; i+=4,++j) {
147.         int sum=a[i] * 8 + a[i + 1] * 4 + a[i + 2] * 2 + a[i + 3];
148.         if(sum<10) result[j]=sum+'0';
149.         else if(sum==10) result[j]='A';
150.         else if(sum==11) result[j]='B';
151.         else if(sum==12) result[j]='C';
152.         else if(sum==13) result[j]='D';
153.         else if(sum==14) result[j]='E';
154.         else if(sum==15) result[j]='F';
155.     }
156.     return result;
157. }
158.

```

```

159. //通过 PC-1 变换产生第一轮秘钥 (64 位密钥->56 位), 同时生成 C0/D0
160. int** Create_SubKey(int** key){
161.     int** SubKey=new int*[8];
162.     for (int i = 0; i < 8; ++i)
163.         *(SubKey+i)=new int[7];
164.     int middle_data[64],num=0,C_num=0,D_num=0;
165.     for (int i = 0; i < 8; ++i)
166.         for (int j = 0; j < 8; ++j)
167.             middle_data[num++]=key[i][j];
168.     for (int i = 0; i < 8; ++i) {
169.         for (int j = 0; j < 7; ++j) {
170.             SubKey[i][j]=middle_data[PC_1[i][j] - 1];
171.             if(C_num<28) C[C_num++]=SubKey[i][j]; //56 位分为 28+28, 分别存到
                C/D 中
172.             else D[D_num++]=SubKey[i][j];
173.         }
174.     }
175.     return SubKey;
176. }
177.
178. //循环产生子密钥, C1-C16,D1-D16,由 C0、D0 推导
179. void Cycle_SubKey(int n){
180.     int times=Left_Shift[n],C0,D0;
181.     for (int i = 0; i < times; ++i) {
182.         C0=C[0];
183.         D0=D[0];
184.         for (int j = 0; j < 28; ++j) {
185.             C[j - 1] = C[j];
186.             D[j - 1] = D[j];
187.         }
188.         C[27] = C0;
189.         D[27] = D0;
190.     }
191. }
192. //通过 PC-2,使 56 位子密钥变为 48 位, 并记录
193. void TurnInto_NewKey(int n){
194.     int** NewSubKey=new int*[8];
195.     for (int i = 0; i < 8; ++i)
196.         *(NewSubKey+i)=new int[6];
197.     int NowCD[56],NewKey[48],num=0;
198.     for (int i = 0; i < 56; ++i) {
199.         if(i<28) NowCD[i]=C[i];
200.         else NowCD[i]=D[i-28];
201.     }

```



```

202.     for (int i = 0; i < 8; ++i) {
203.         for (int j = 0; j < 6; ++j) {
204.             NewKey[num++] = NowCD[PC_2[i][j]-1];
205.             SubKey_ByPC2[15-n][num-1] = NewKey[num-1];
206.         }
207.     }
208. }
209. //通过 IP 变换初始化 L,R
210. void IP_Transform(int** original){
211.     int ori_data[64], num=0, middle_data[64];
212.     for (int i = 0; i < 8; ++i)
213.         for (int j = 0; j < 8; ++j)
214.             ori_data[num++] = original[i][j];
215.     for (int i = 0; i < 8; ++i) {
216.         for (int j = 0; j < 8; ++j) {
217.             middle_data[8*i+j] = ori_data[IP[i][j]-1];
218.             if(8*i+j<32) L[8*i+j] = middle_data[8*i+j];
219.             else R[8*i+j-32] = middle_data[8*i+j];
220.         }
221.     }
222. }
223. /*为了计算 F，我们首先拓展每个 Rn-1，将其从 32 位拓展到 48 位。这是通过使用表 E
224.  * 来重复 Rn-1 中的一些位来实现。也就是说函数 E(Rn-1)输入 32 位输出 48 位。*/
225. int* F(int n, int flag){
226.     int *result = new int[32], ** B = new int*[8];
227.     for (int i = 0; i < 8; ++i)
228.         *(B+i) = new int[6];
229.     int middle_ER[48], NowK[48], a[48], first, second, middle_all_s[8], result_s[
32];
230.     for (int i = 0; i < 48; ++i) {
231.         middle_ER[i] = R[E_Table[i]-1]; //使用表 E 扩展位数
232.         NowK[i] = SubKey_ByPC2[16-n][i]; //取原先记录的子密钥
233.     }
234.     //逆变换，为了实现解密过程准备
235.     if(flag==1){
236.         for (int i = 0; i < 48; ++i) {
237.             NowK[i] = SubKey_ByPC2[n-1][i];
238.         }
239.     }
240.     for (int i = 0; i < 48; ++i)
241.         a[i] = (NowK[i] + middle_ER[i])%2; //进行 XOR 运算
242.     for (int i = 0; i < 8; ++i)
243.         for (int j = 0; j < 6; ++j) //将 8 组 6 比特数据记录在 B 中
244.             B[i][j] = a[6*i+j];

```

```

245.      /*B 的第一位和最后一位组合起来的二进制数决定一个介于 0 和 3 之间的十进制数（或者
          二进制 00 到 11 之间）。
246.      * 设这个数为 i。B 的中间 4 位二进制数代表一个介于 0 到 15 之间的十进制数（二进制
          0000 到 1111）。
247.      * 设这个数为 j。查表找到第 i 行第 j 列的那个数*/
248.      for (int i = 0; i < 8; ++i) {
249.          first=B[i][0]*2+B[i][5];          //将 8 组数据输入 S 函数
250.          second=B[i][1] * 8 + B[i][2] * 4 + B[i][3] * 2 + B[i][4];
251.          middle_all_s[i]=S_Box[i][first][second];
252.      }
253.      for (int i = 0; i < 8; ++i) {
254.          result_s[i*4]=middle_all_s[i]/8;
255.          result_s[i*4+1]=(middle_all_s[i]-result_s[i*4]*8)/4;
256.          result_s[i*4+2]=(middle_all_s[i]-result_s[i*4]*8-
              result_s[i*4+1]*4)/2;
257.          result_s[i*4+3]=middle_all_s[i]-result_s[i*4]*8-result_s[i*4+1]*4-
              result_s[i*4+2]*2;
258.      }
259.      //对 S 盒的输出转换成二进制之后，进行 P 变换，得到 32 位数据
260.      for (int i = 0; i < 32; ++i)
261.          result[i]=result_s[P[i]-1];
262.      return result;
263. }
264. /*通次此函数进行 16 次迭代后得到 L1-L16,R1-R16
265. * Ln = Rn-1 ; Rn = Ln-1 + F(Rn-1,Kn); */
266. void Update_LR(int n,int flag){
267.     int middle_L[32],middle_R[32];
268.     for (int i = 0; i < 32; ++i) {
269.         middle_L[i]=L[i];
270.         middle_R[i]=R[i];
271.     }
272.     int* result=new int[32];
273.     //在 F 函数中实现 F(Rn-1,Kn)
274.     result=F(n,flag);
275.     for (int i = 0; i < 32; ++i) {
276.         L[i]=middle_R[i];
277.         R[i]=(result[i]+middle_L[i])%2;
278.     }
279.     /*在第 16 个迭代之后，我们有了区块 L16 and R16。
280.     * 接着我们逆转两个区块的顺序得到一个 64 位的区块*/
281.     if(n==16){
282.         for (int i = 0; i < 32; i++) {
283.             int temp = L[i];
284.             L[i] = R[i];

```

```

285.         R[i] = temp;
286.     }
287. }
288. }
289.
290. //实现 DES 的解密过程，只是中间存在一步子密钥顺序的调换，其他相同
291. void Decode(string original){
292.     int** middle=new int*[8];
293.     int result[64],Combine_LR[64];
294.     for (int i = 0; i < 8; ++i)
295.         middle[i]=new int[8];
296.     middle=HexToBinary(original);
297.     IP_Transform(middle);
298.     for (int i = 1; i <= 16; ++i) {
299.         Update_LR(i,1);
300.     }
301.     for (int j = 0; j < 64; ++j) {
302.         if(j<32) Combine_LR[j]=L[j];
303.         else Combine_LR[j]=R[j-32];
304.     }
305.     for (int j = 0; j < 8; ++j) {
306.         for (int k = 0; k < 8; ++k) {
307.             result[j*8+k]=Combine_LR[IP_Inverse[j][k]-1];
308.         }
309.     }
310.     string Final_result=BinaryToHex(result);
311.     cout<<"Decryption result: "<<Final_result<<endl;
312. }
313. /*-----Main 函数，程序入口-----
    ----*/
314. int main() {
315.     string text="1803040001400000";
316.     string key="23A4Z77995BC0FF1";
317.     string cipher_text[16];
318.     int Result_Cipher_Text[16][64],middle_result[8][8],Combine_LR[64];
319.     cout << "Text: " << text << endl;
320.     cout << "Key: " << key << endl;
321.     int** Target_text=new int*[8],**Target_key=new int*[8],**SubKey=new int
        *[8];
322.     for (int i = 0; i < 8; ++i) {
323.         *(Target_text+i)=new int[8];
324.         *(Target_key+i)=new int[8];
325.         *(SubKey+i)=new int[7];
326.     }

```

```

327.    //将明文转换成二进制，并将 64 位二进制结果存成 8*8 的矩阵
328.    Target_text=HexToBinary(text);
329.    //将密钥转换成二进制，并将 64 位二进制结果存成 8*8 的矩阵
330.    Target_key=HexToBinary(key);
331.    //将 64 位密钥根据表格 PC-1 进行变换，得到 56 位新密钥 K+，再分为 C0,D0
332.    SubKey=Create_SubKey(Target_key);
333.    for (int i = 1; i <= 16; ++i) {
334.        //根据 C0,D0 进行 16 次循环移位，依次得到 C1-C16,D1-D16
335.        Cycle_SubKey(i-1);
336.        //对子密钥进行拼合，形成 CnDn，并按照 PC-2 执行变换，得到 48 位密钥，循环
        16 次，得到 K1-K16
337.        TurnInto_NewKey(i-1);
338.    } //子密钥处理告一段落
339.    //对明文数据通过 IP 进行变换，并将 64 位二进制数据分为 L 和 R 两部分，各 32
340.    IP_Transform(Target_text);
341.    //进行 16 次迭代
342.    for (int i = 1; i <= 16; ++i) {
343.        /*通次此函数进行 16 次迭代后得到 L1-L16,R1-R16
344.        * Ln = Rn-1 ;
345.        * Rn = Ln-1 + F(Rn-1,Kn);
346.        */
347.        Update_LR(i,0);
348.        for (int j = 0; j < 64; ++j) {
349.            if(j<32) Combine_LR[j]=L[j];
350.            else Combine_LR[j]=R[j-32];
351.        }
352.        //对翻转过后的 LR 执行最终一步 IP-1 变换
353.        for (int j = 0; j < 8; ++j) {
354.            for (int k = 0; k < 8; ++k) {
355.                Result_Cipher_Text[i-1][j*8+k]=Combine_LR[IP_Inverse[j]][k]-
                1];
356.            }
357.        }
358.        cipher_text[i-1]=BinaryToHex(Result_Cipher_Text[i-1]);
359.        cout<<"----- " <<i<<" -----"<<endl;
360.        cout<<"Cipher Text: " <<cipher_text[i-1]<<endl;
361.    }
362.    Decode(cipher_text[15]); //对最终第 16 次加密结果进行解密，查看是否对应初始
        明文。
363.    return 0;
364. }

```