完成一个 DES 算法的详细设计，包括

算法原理概述；总体结构；模块分解；数据结构；类-C 语言算法过程。

DES 算法详细设计

1．算法原理概述

DES 算法将明文以 64 位为分组长度进行分组，将 64 位一组的明文作为算法的输入，经过一系列的过程后输出 64 位的密文。密钥也是 64 位，但密钥的第 8,16,24,...,64 作为奇偶校验位，所以实际起作用的只有 56 位。DES 算法的加密过程和解密过程唯一的不同是在 16 次迭代的时候子密钥的使用顺序相反。以下是 DES 加密算法的具体过程。

注：明文：P = $p_1p_2p_3...p_{64}$， p_i {0,1}

密文：C = $c_1c_2c_3...c_{64}$， c_i {0,1}

密钥：K = $k_1k_2k_3...k_{64}$， k_i {0,1}

（1）初始置换（64-64）

明文根据初始置换表（IP 置换表）进行置换。

IP 置换表							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

置换规则：明文中的第 58 位放到第一位，第 50 位放到第二位，依此类推，最终得到经过 IP 置换后的文本 A。

（2）子密钥生成

密钥的长度为 64 位二进制数。最终要生成 16 个 48 位的子密钥供迭代过程中 Feistel 轮函数每轮变换的调用。

① **PC-1 置换（64 -56）**

对 K 的 56 个非校验位实行置换 PC-1，置换规则与过程（1）IP 置换的规则相同，经置换后得到 $K_l K_r$，其中 K_l 和 K_r，分别由 PC-1 置换后的前 28 位和后 28 位组成。因为 8，16，24,32,40,48,56,64 这些位是奇偶校验位，所以在 PC-1 置换表中不会出现这些数字。

PC-1 置换表						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

② **每轮密钥左移（28-28）**

根据下面的表格决定左移的位数

迭代次数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
左移位数	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

例如：当迭代次数为 1 的时候，左右两半部分分别左移一位，得到 $K_L \quad k_2 k_3 \ldots k_{28} k_1$，$K_R \quad d_2 d_3 \ldots d_{28} d_1$，将 K_L 和 K_R 合并后进行 PC-2 置换即可得到 subkey[0]。将 K_L 和 K_R 再次进行左移并进行 PC-2 置换即可得到 subkey[1]，依次类推，直到得到 subkey[16]。

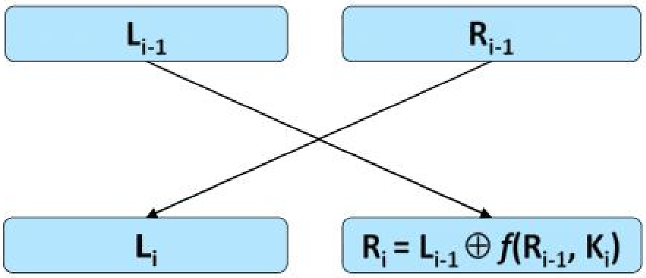
③ **PC-2 置换（56-48）**

因为 PC-2 置换将 56 位的密钥经置换后得到 48 位，所以 PC-2 置换也称为压缩置换。

PC-2 压缩置换表					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

（3）16 次迭代过程

将（1）得到的文本 A 进行 16 轮迭代得到文本 B,迭代过程为：



$L_i \quad R_{i\;1}$，$R_i \quad L_{i\;1} \quad f(R_{i\;1},k_i)$，其中 L_0 A[0...31], R_0 A[32...63].f 为 feistel 轮函数，ki 调用的是（2）中求得的 16 个长度为 48bit 的子密钥。

16 次迭代后得到的是 $L_{16}\ R_{16}$,进行交换得到 $R_{16}\ L_{16}$。

（4）feistel 轮函数（（32,48）-32）

1) 首先将 R_{i-1} 通过 E-扩展置换表进行扩展，将其从 32 位转换为 48 位的 R_e；

E-扩展规则 (比特-选择表)					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

2) R_e与k_i进行异或运算，得到 48 位的 T;

3) 将 T 经过 8 个 S-盒后得到 32 的 T1;

4) T1 与L_{i-1}进行异或运算得到R_i

（5）S-盒代替（48-32）

输入时 48bits,输出是 32bits

将(4)中输入的 48 位的 T,以 6 为分组长度分为 8 组，每一组 6 位经过一个 S 盒后得到输出 4 位，最后将 8 个 S 盒的输出合并得到 32 位的 T1.

S 盒的实现（6-4）：每一个 S 盒的输入是 6bit，输出是 4bit。S 盒中是一个 4*16 的矩阵（用二维数组实现），总共有 64 个数，这 64 个数用 16 个数字（0-15，4 个 bit 表示）填充。记输入为$a_1a_2a_3a_4a_5a_6$，则将$(a_1a_6)_{10}$作为矩阵的行坐标，将$(a_2a_3a_4a_5)_{10}$作为矩阵的列坐标，根据行列坐标在二维矩阵中找到对应的四位二进制数即为输出。

S_1-BOX															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	15	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S_2-BOX															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S_3-BOX															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S_4-BOX															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
12	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S_5-BOX															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S_6-BOX															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S_7-BOX															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S_8-BOX															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

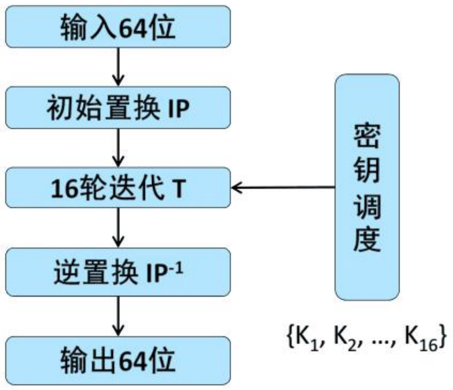
（6）逆置换 IP^{-1}（64-64）

根据 IP^{-1} 置换表将（3）中经过 16 次迭代后得到的 $R_{16}L_{16}$ 再次进行置换得到最终的密文。

IP^{-1} 置换表							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

2. 总体结构

（1）总体结构：



（2）迭代过程（Feistel 轮函数）

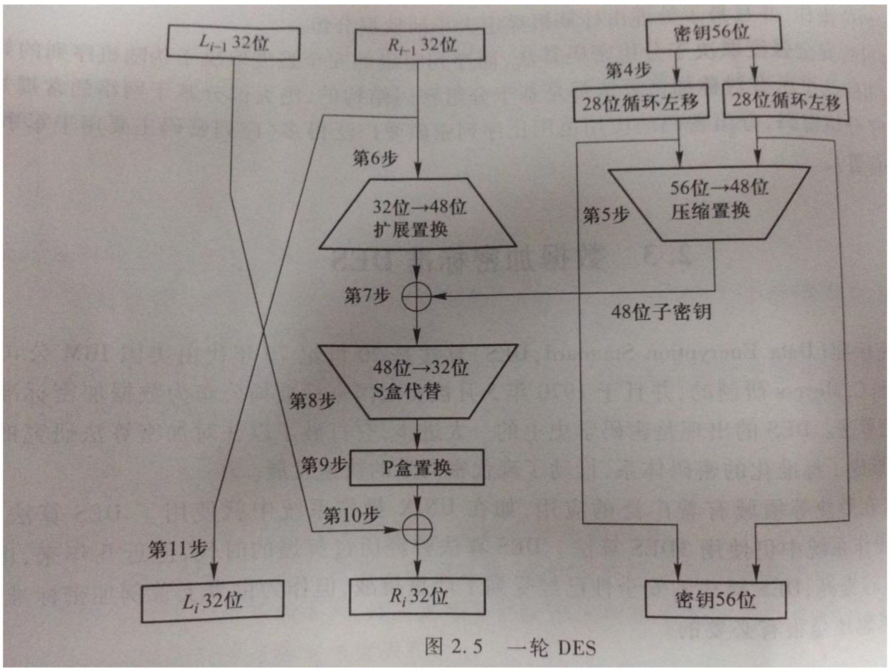


图 2.5　一轮 DES

（3）S 盒替换

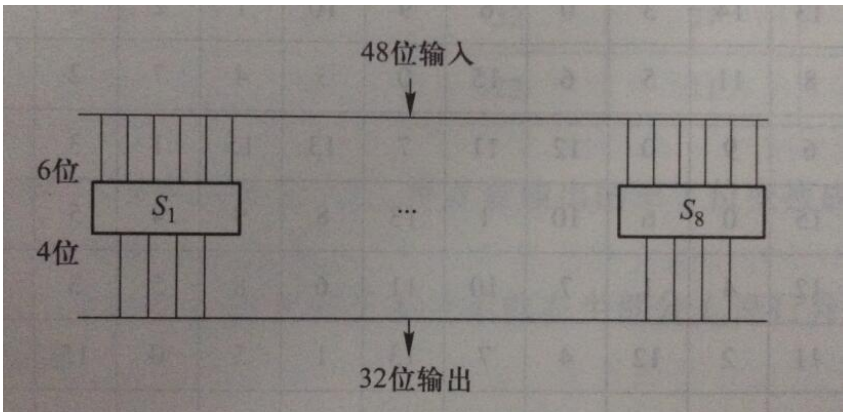


图 2.6 S 盒的输入输出

3．模块分解

```
//  functions
bitset<64> charToBit(const char plain[8]);
bitset<64> initialPermutation(bitset<64> plainBits);
void getSubKeys(bitset<64> key);
void getDecryptionSubKeys();
bitset<64> transform16(bitset<64> text);
bitset<32> feistel(bitset<32> lastRight, bitset<48> oneSubKey);
bitset<32> sBoxCompress(bitset<48> xorResult);
bitset<4> intToBit(int value);
bitset<56> PC1Permutation(bitset<64> key);
bitset<48> PC2Permutation(bitset<56> tempkey);
bitset<28> LSCirculation(bitset<28> halftemp, int shiftBit);
bitset<64> finalPermutation(bitset<64> afterTrans);
```

关系结构图：

Encryption |-initialPermutation

|-getSubKeys |-PC1Permutation

|-PC2Permutation

|-LSCirculation

|-transform16 |-feistel-sBoxCompress

|-sBoxCompress

|-finalPermutation

4．数据结构

bitset、数组、递归

5．实验小结

实验过程需要注意的问题：（1）置换表是从 1 开始，而数组是从 0 开始，一开始的时候误以为置换表的值是从 0 开始而导致溢出错误。（2）16 次迭代后要将 L_{16} R_{16} 进行交换得到 R_{16} L_{16}，保证解密过程能够使用加密的函数。（3）S 盒实现的时候涉及到十进制数和二进制数的转换，需要注意是 $(a_1a_6)_{10}$ 和 $(a_2a_3a_4a_5)_{10}$ 而不是 $(a_1a_2)_{10}$ 和 $(a_3a_4a_5a_6)_{10}$.

本次实验采用数组实现，基本算法思路非常清晰，实现没有太大难度，需要小心的是有很多表格，其输入需要注意。解密过程也不难，只需要将得到的子密钥倒序即可，算法与加密时完全一样。通过这次实验，加深了我对 DES 算法的理解，也掌握了 DES 算法的具体实现细节。

6.代码实现

测试：

明文：1234567

密钥：89674523

输出结果：

```
initial plain:
0011100000011011100110110001101010011010000110011001100100
0110001
```

cipher text:

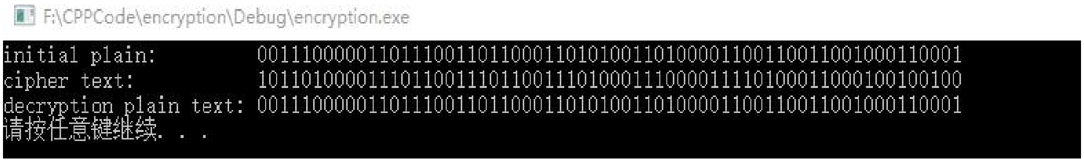
1011010000111011001110110011101000111000011110100011000100

0100100

decryption plain text:

001110000011011100110110001101010011010000110011001100100

0110001



------------------------ code --------------------------

```
#include <iostream>
#include <bitset>
#include <cstdlib>
using namespace std;

bitset<48> subkeys[16];
//  initial IP permutation table
//  notice: values in IP are from 1 to 64
int IP[] = {
        58, 50, 42, 34, 26, 18, 10, 2,
        60, 52, 44, 36, 28, 20, 12, 4,
        62, 54, 46, 38, 30, 22, 14, 6,
        64, 56, 48, 40, 32, 24, 16, 8,
        57, 49, 41, 33, 25, 17, 9, 1,
        59, 51, 43, 35, 27, 19, 11, 3,
        61, 53, 45, 37, 29, 21, 13, 5,
        63, 55, 47, 39, 31, 23, 15, 7
};
int IPInverse[] = {
        40, 8, 48, 16, 56, 24, 64, 32,
        39, 7, 47, 15, 55, 23, 63, 31,
        38, 6, 46, 14, 54, 22, 62, 30,
        37, 5, 45, 13, 53, 21, 61, 29,
        36, 4, 44, 12, 52, 20, 60, 28,
        35, 3, 43, 11, 51, 19, 59, 27,
        34, 2, 42, 10, 50, 18, 58, 26,
        33, 1, 41,   9, 49, 17, 57, 25
};
```

```
int P1[] = {
        57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10,    2, 59, 51, 43, 35, 27,
        19, 11,    3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14,    6, 61, 53, 45, 37, 29,
        21, 13,    5, 28, 20, 12,    4
};
//    56 bits to 48 bits
int P2[] = {
        14, 17, 11, 24,    1,    5,
        3, 28, 15,    6, 21, 10,
        23, 19, 12,    4, 26,    8,
        16,    7, 27, 20, 13,    2,
        41, 52, 31, 37, 47, 55,
        30, 40, 51, 45, 33, 48,
        44, 49, 39, 56, 34, 53,
        46, 42, 50, 36, 29, 32
};
//    32 bits to 48 bits
int expand[] = {
        32,    1,    2,    3,    4,    5,
        4,    5,    6,    7,    8,    9,
        8,    9, 10, 11, 12, 13,
        12, 13, 14, 15, 16, 17,
        16, 17, 18, 19, 20, 21,
        20, 21, 22, 23, 24, 25,
        24, 25, 26, 27, 28, 29,
        28, 29, 30, 31, 32,    1
};
// feistel final permutation
int fp[] = {
        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
};
//    48 bits to 32 bits
```

```
int boxes[8][4][16] = {
        {
                { 14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7 },
                { 0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8 },
                { 4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0 },
                { 15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13 }
        },
        {
                { 15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10 },
                { 3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5 },
                { 0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15 },
                { 13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9 }
        },
        {
                { 10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8 },
                { 13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1 },
                { 13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7 },
                { 1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12 }
        },
        {
                { 7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15 },
                { 13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9 },
                { 10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4 },
                { 3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14 }
        },
        {
                { 2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9 },
                { 14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6 },
                { 4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14 },
                { 11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3 }
        },
        {
                { 12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11 },
                { 10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8 },
                { 9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6 },
                { 4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13 }
        },
        {
                { 4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1 },
                { 13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6 },
                { 1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2 },
                { 6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12 }
        },
        {
```

```
        { 13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7 },
        { 1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2 },
        { 7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8 },
        { 2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11 }
    }
};
int shiftBits[] = { 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1 };

//  functions
bitset<64> charToBit(const char plain[8]);
bitset<64> initialPermutation(bitset<64> plainBits);
void getSubKeys(bitset<64> key);
void getDecryptionSubKeys();
bitset<64> transform16(bitset<64> text);
bitset<32> feistel(bitset<32> lastRight, bitset<48> oneSubKey);
bitset<32> sBoxCompress(bitset<48> xorResult);
bitset<4> intToBit(int value);
bitset<56> PC1Permutation(bitset<64> key);
bitset<48> PC2Permutation(bitset<56> tempkey);
bitset<28> LSCirculation(bitset<28> halftemp, int shiftBit);
bitset<64> finalPermutation(bitset<64> afterTrans);

//  change the char array to 64 bits
bitset<64> charToBit(const char plain[8]) {
    bitset<64> bits;
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; ++j) {
            //  here save the ascii binary bits of "12345678"
            bits[i * 8 + j] = ((plain[i] >> j) & 1);
        }
    }
    return bits;
}
//  initial permutation
bitset<64> initialPermutation(bitset<64> plainBits) {
    bitset<64> result;
    for (int i = 0; i < 64; ++i) {
        result[i] = plainBits[IP[i] - 1];
    }
    return result;
}
//  16 times transformation
bitset<64> transform16(bitset<64> text) {
    //  call feistel function
```

```
	bitset<32> lefts[17], rights[17];
	bitset<64> result;
	bitset<32> feistelRes;
	//	L0
	for (int i = 0; i < 32; ++i) {
		lefts[0][i] = text[i];
	}
	//	R0
	for (int i = 32; i < 64; ++i) {
		rights[0][i - 32] = text[i];
	}
	int count = 0;
	while (count < 16) {
		count++;
		//	Li = R	(i-1)
		lefts[count] = rights[count - 1];
		feistelRes = feistel(rights[count - 1], subkeys[count - 1]);
		for (int i = 0; i < 32; ++i) {
			rights[count] = lefts[count - 1] ^ feistelRes;
		}
	}
	//	result = R16L16
	for (int i = 0; i < 32; ++i)
		result[i] = rights[16][i];
	for (int i = 0; i < 32; ++i)
		result[i + 32] = lefts[16][i];
	return result;
}
//	feistel
bitset<32> feistel(bitset<32> lastRight, bitset<48> oneSubKey) {
	//	call sBoxCompress
	//	expand and make lastRight to 48 bits
	bitset<48> expandRight;
	for (int i = 0; i < 48; ++i) {
		expandRight[i] = lastRight[expand[i] - 1];
	}
	bitset<48> xorResult = expandRight ^ oneSubKey;
	bitset<32> fromSBox = sBoxCompress(xorResult);
	bitset<32> result;
	//	there is a P-permutation in the last
	for (int i = 0; i < 32; ++i) {
		result[i] = fromSBox[fp[i] - 1];
	}
	return result;
```

```
}
//    change 48 bits to 32 bits by using s-boxs
bitset<32> sBoxCompress(bitset<48> xorResult) {
        bitset<32> result;
        bitset<6> subGroups[8];
        for (int i = 0; i < 48; ++i) {
                subGroups[i / 6][i % 6] = xorResult[i];
        }
        int indexI, indexJ;
        int values[8];
        for (int i = 0; i < 8; ++i) {
                indexI = subGroups[i][0] * 2 + subGroups[i][5];
                indexJ = subGroups[i][1] * 8 + subGroups[i][2] * 4 + subGroups[i][3] * 2
                        + subGroups[i][4];
                values[i] = boxes[i][indexI][indexJ];
        }
        for (int i = 0; i < 8; ++i) {
                bitset<4> temp = intToBit(values[i]);
                for (int k = 0; k < 4; ++k)
                        result[i * 4 + k] = temp[k];
        }
        return result;
}
//    decimal to binary(4 bits)
bitset<4> intToBit(int value) {
        bitset<4> result;
        for (int i = 0; i < 4; ++i) {
                result[i] = (value >> 1) & 1;
        }
        return result;
}
//    inverse permutation in the last
bitset<64> finalPermutation(bitset<64> afterTrans) {
        bitset<64> result;
        for (int i = 0; i < 64; ++i) {
                result[i] = afterTrans[IPInverse[i] - 1];
        }
        return result;
}
//    get 16 sub keys and save them in subkeys
void getSubKeys(bitset<64> key) {
        //    PC-1 permutation
        bitset<56> keyAfterPC1 = PC1Permutation(key);
        bitset<28> tempLeft, tempRight;
```

```
	bitset<28> LSlefts[17], LSrights[17];
	bitset<56> keyAfterShift;
	for (int i = 0; i < 28; ++i)
		tempLeft[i] = keyAfterPC1[i];
	for (int i = 0; i < 28; ++i)
		tempRight[i] = keyAfterPC1[i + 28];
	LSlefts[0] = tempLeft;
	LSrights[0] = tempRight;
	int count = 0;
	while (count < 16) {
		count++;
		LSlefts[count] = LSCirculation(LSlefts[count - 1], shiftBits[count - 1]);
		LSrights[count] = LSCirculation(LSrights[count - 1], shiftBits[count - 1]);
		for (int i = 0; i < 28; ++i) keyAfterShift[i] = LSlefts[count][i];
		for (int i = 0; i < 28; ++i) keyAfterShift[i + 28] = LSrights[count][i];
		//	now get the 56 bits temp key
		subkeys[count - 1] = PC2Permutation(keyAfterShift);
	}
}
//	PC-1 permutation
bitset<56> PC1Permutation(bitset<64> key) {
	bitset<56> result;
	for (int i = 0; i < 56; ++i) {
		result[i] = key[P1[i] - 1];
	}
	return result;
}
//	PC-2 permutation
bitset<48> PC2Permutation(bitset<56> tempkey) {
	bitset<48> result;
	for (int i = 0; i < 48; ++i) {
		result[i] = tempkey[P2[i] - 1];
	}
	return result;
}
//	left shift
bitset<28> LSCirculation(bitset<28> halftemp, int shiftBit) {
	bitset<28> result;
	//	left shift 1 or 2 bits
	for (int i = 0; i < 28; ++i) {
		result[i] = halftemp[(i + shiftBit) % 28];
	}
	return result;
}
```

```cpp
//	decryption subkeys
void getDecryptionSubKeys() {
	//	PC-1 permutation
	bitset<48> temp[16];
	for (int i = 0; i < 16; ++i) {
		temp[i] = subkeys[15 - i];
	}
	for (int i = 0; i < 16; ++i) {
		subkeys[i] = temp[i];
	}
}
bitset<64> encryption(bitset<64> plainBits, bitset<64> keyBits) {
	getSubKeys(keyBits);
	bitset<64> afterInitialPer = initialPermutation(plainBits);
	bitset<64> result = transform16(afterInitialPer);
	bitset<64> finalResult = finalPermutation(result);
	return finalResult;
}
bitset<64> decryption(bitset<64> cipher, bitset<64> keyBits) {
	getDecryptionSubKeys();
	bitset<64> afterInitialPer = initialPermutation(cipher);
	bitset<64> result = transform16(afterInitialPer);
	bitset<64> finalResult = finalPermutation(result);
	return finalResult;
}
int main() {
	char plainText[] = "12345678";
	char key[] = "abcdefgh";
	bitset<64> plainBits = charToBit(plainText);
	bitset<64> keyBits = charToBit(key);

	bitset<64> cipher = encryption(plainBits, keyBits);
	bitset<64> plain = decryption(cipher, keyBits);

	cout << "initial plain:             " << plainBits << endl;
	cout << "cipher text:                " << cipher << endl;
	cout << "decryption plain text: " << plain << endl;
	system("pause");
	return 0;
}
```

------------------------ code --------------------------