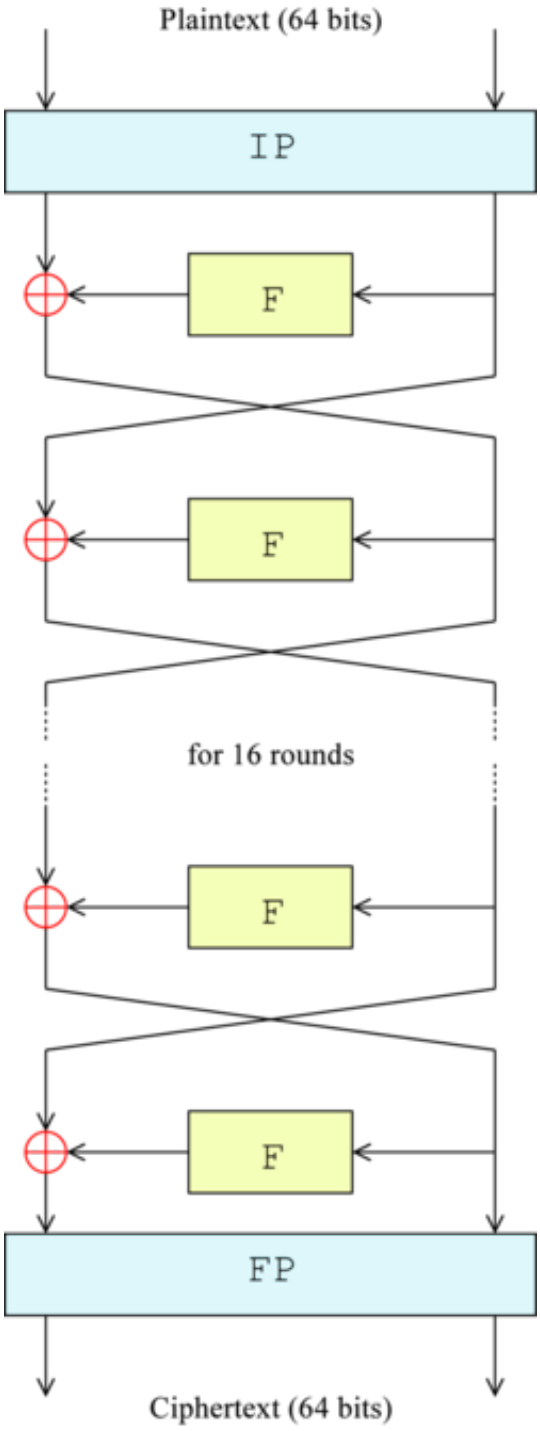
《密码学》实验二报告

DES分组密码实现

DES算法将明文分成64位大小的众多数据块，即分组长度为64位。同时用56位密钥对64位明文信息加密，最终形成64位的密文。如果明文长度不足64位，即将其扩展为64位（如补零等方法）。具体加密过程首先是将输入的数据进行初始置换（IP），即将明文M中数据的排列顺序按一定的规则重新排列，生成新的数据序列，以打乱原来的次序。然后将变换后的数据平分成左右两部分，左边记为L_0，右边记为R_0，然后对R_0实行在子密钥（由加密密钥产生）控制下的变换f（即Feistel网络），结果记为$f(R_0，K_1)$，再与L_0做逐位异或运算，其结果记为R_1，R_0则作为下一轮的L_1。如此循环16轮，最后得到L_{16}和R_{16}，再对L_{16}、R_{16}实行逆初始置换IP^{-1}，即可得到加密数据。解密过程与此类似，不同之处仅在于子密钥的使用顺序正好相反。



Part 1. 初始置换（Initial Permutation）

它的作用是把输入的64位数据块的排列顺序打乱，每位数据按照下面的置换规则重新排列，即将第58位换到第一位，第50位换打第2位，...，依次类推。置换后的)64位输出分为L_0 、R_0（左、右)两部分，每部分分别为32位。

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

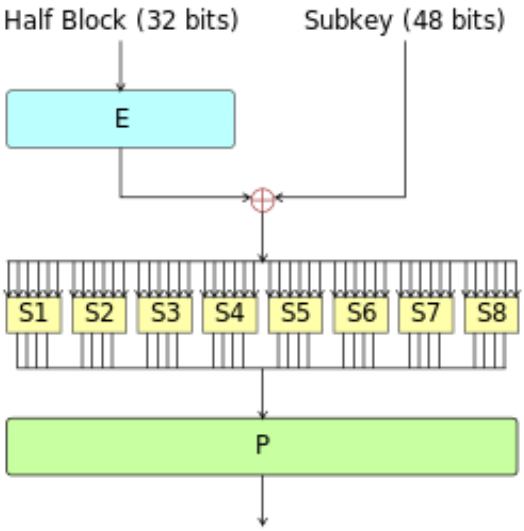
此时，我们可以轻松地使用C++的数组来实现这个变换。设置换表为IP，则$\mathrm{IP}[i] = j$表示在置换后的字符串中下标为i的比特应该是原来字符串的第j个元素（注意下标从0开始）。

```
// 左半部分的初始置换表
static const unsigned char initial_permutation_left[32] = {
    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,
};

// 右半部分的初始置换表
static const unsigned char initial_permutation_right[32] = {
    56, 48, 40, 32, 24, 16, 8,  0, 58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6,
};
```

Part 2. f-函数

f-函数是多个**置换函数**和**替代函数**的组合函数，它将32位比特的输入变换为32位的输出。R_i经过扩展运算E变换后扩展为48位的比特串$E(R_i)$，与进行异或运算后输出的结果分成8组，每组6比特。每一组再经过一个S盒（共8个S盒）运算转换为4位，8个4位合并为32位后再经过P变换输出为32位的。其中，扩展运算E与置换P主要作用是增加算法的扩散效果。



- 扩展函数E：这个函数将32位的输入扩展成48位的输出，并将一些比特位进行重复。

```
static const unsigned char expansion_permutation[48] = {
    31, 0,  1,  2,  3,  4,  3,  4,  5,  6,  7,  8,  7,  8,  9,  10,
    11, 12, 11, 12, 13, 14, 15, 16, 15, 16, 17, 18, 19, 20, 19, 20,
    21, 22, 23, 24, 23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31, 0,
};
```

  同样地，该表含义为：第i项的值j代表了变换后的第i位比特是原来的字符串的第j个比特位。

- S盒替换：令48位的比特串为$B = B_0B_1B_2B_3B_4B_5B_6B_7$，那么每个6比特的比特串$B_i = b_0b_1b_2b_3b_4b_5$，将其首末位提取得到行号$r = b_0b_5$，剩余位作为列号$c = b_1b_2b_3b_4$后，在对应的$S_i$盒中进行替换。

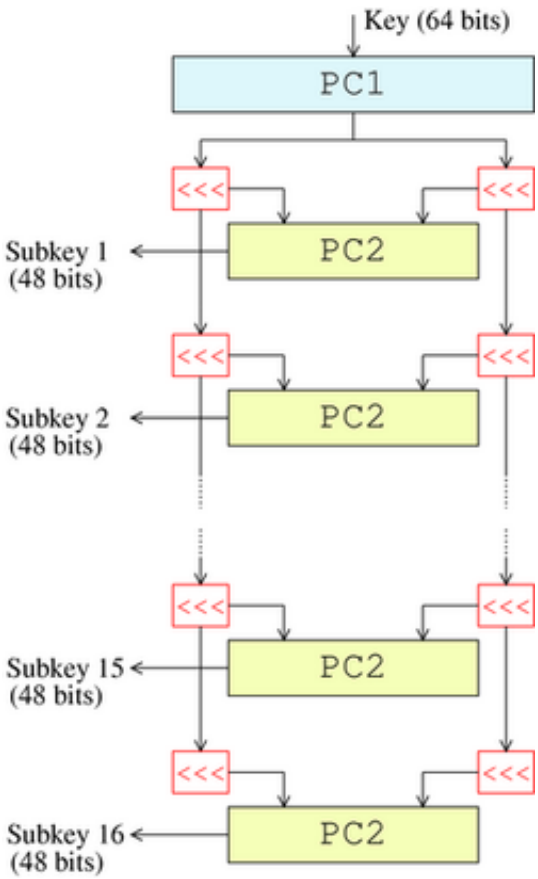
- P置换：每轮Feistel网络会再次打乱输出结果的顺序，可用数组表示如下：

```
static const unsigned char feistel_end_permutation[32] = {
    15, 6, 19, 20, 28, 11, 27, 16, 0,  14, 22, 25, 4,  17, 30, 9,
    1,  7, 23, 13, 31, 26, 2,  8,  18, 12, 29, 5,  21, 10, 3,  24,
};
```

P

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

Part 3. 主密钥和轮密钥的生成



上图展示了DES中密钥调度算法的详细流程。

首先，使用**选择置换1**（PC-1）从64位输入密钥中选出56位的密钥—剩下的8位要么直接丢弃，要么作为奇偶校验位。然后，56位分成两个28位的半密钥；每个半密钥接下来都被分别处理。在接下来的回次中，两个半密钥都被左移1或2位（由回次数决定），然后通过**选择置换2**（PC-2）产生48位的子密钥—每个半密钥24位。移位（图中由**<<<**标示）表明每个子密

钥中使用了不同的位，每个位大致在16个子密钥中的14个出现。在解密过程中，除了子密钥输出的顺序相反外，密钥调度的过程与加密完全相同。

- 密钥移位表：

```
static const unsigned char key_shift_amounts[16] = {1, 1, 2, 2, 2, 2, 2, 2,
                                                    1, 2, 2, 2, 2, 2, 2, 1};
```

- 选择置换表：

```
static const unsigned char permuted_choice_1[56] = {
    // 左半部分密钥
    56, 48, 40, 32, 24, 16, 8, 0, 57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42,
34,
    26, 18, 10, 2, 59, 51, 43, 35,
    // 右半部分密钥
    62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 60, 52, 44,
    36, 28, 20, 12, 4, 27, 19, 11, 3};

static const unsigned char permuted_choice_2[48] = {
    13, 16, 10, 23, 0,  4,  2,  27, 14, 5,  20, 9,  22, 18, 11, 3,
    25, 7,  15, 6,  26, 19, 12, 1,  40, 51, 30, 36, 46, 54, 29, 39,
    50, 44, 32, 47, 43, 48, 38, 55, 33, 52, 45, 41, 49, 35, 28, 31};
```

DES分组密码的C++实现

在使用C/C++实现DES分组密码的时候需要额外注意比特、数组、指针之间的转换关系，以及对应的接口规范。因此在这一部分有必要对接口以及数据的格式进行说明。

- 数据格式：均以`unsigned char`数组（每个`unsigned char`都是8比特）的形式进行实现，为了传参方便，所有的数组都是指针。这一部分数据解读的时候都是二进制字符串。
- 接口：
  - 将二进制字符串打印出来，形成可读的结果：

    ```
    char* bin_to_string(const unsigned char* input, int length);
    ```

  - 逐比特异或：

```
void xor_op(const unsigned char* left, const unsigned char* right,
            unsigned char* output, unsigned char nbytes);
```

考虑单独实现一个逐比特异或算法是为了一次性进行多个比特的异或，实现解耦的目的。

- 置换算法：

```
void permute(const unsigned char* input, const unsigned char* table,
             unsigned char* output, unsigned char nbytes);
```

该置换算法是置换表茫然的，即给定输入、置换表、输出字符串指针和长度，它能够有效地进行置换，具体实现如下：

```
void permute(const unsigned char* input, const unsigned char* table,
             unsigned char* output, unsigned char nbytes) {
  const unsigned char* table_cell = table;

  for (unsigned char i = 0; i < nbytes; i++) {
    unsigned char result_byte = 0x00;
    for (unsigned char j = 0; j < 8; j++) {
      unsigned char bit_pos = *table_cell % 8;
      unsigned char mask = 0x80 >> bit_pos;
      unsigned char result_bit = (input[*table_cell / 8] & mask) <<
bit_pos;
      result_byte |= result_bit >> j;

      table_cell++;
    }
    output[i] = result_byte;
  }
}
```

以64位初始置换为例，此时输入的置换表就是`initial_permutation`，`nbytes = 8`。该算法每次以字节为单位对输入的比特串进行处理，而对于每个字节内部的比特串，它会根据置换表的数据找到相应的比特。

1. 计算输入的比特串`input`中第i个比特所处的相对位置：`unsigned char bit_pos = *table_cell % 8`；
2. 计算掩码：`unsigned char mask = 0x80 >> bit_pos`，此处的`0x80`是指`10000000`，根据比特所在的位置屏蔽其他无用比特位。

3. 取出输出比特串中对应的字节，并用掩码取出目标比特位：`result = input[*table_cell / 8] & mask`；
4. 根据相对位置将比特进行偏移：`result <<= pos_bit`。
5. 放置到结果字节中即可。

- S盒替换：

```
void des_substitution_box(const unsigned char* input, unsigned char*
output);
```

由于这一部分没办法较好地使用循环等操作进行求值，所以采用手工计算八个S盒的替换。

- 轮密钥生成：

```
void des_key_shift(unsigned char* key, unsigned char* output,
                   unsigned char amount);
```

它根据输入的数量进行移位。

- f-函数：

```
void des_feistel(const unsigned char* input, const unsigned char*
subkey,
                 unsigned char* output);
```

- DES加密主函数：

```
void des_encrypt(unsigned char* block, unsigned char* key,
                 unsigned char* output);
```

雪崩效应检验

由于DES采用了f-函数，因此密钥和明文的一点扰动就会造成密文发生眼中的不同，为了验证这一效应，我们可以随机选取一些密钥和明文进行测试。

- 测试1：

```
明文：0x12 0x34 0x56 0x78 0x12 0x34 0x56 0x78
密钥：0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01

```

```
输出：
./des_crypto
0x4a438ac15d8074b5

改动后
明文：0x12 0x34 0x56 0x78 0x12 0x34 0x56 0x77
密钥：0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01

输出：
./des_crypto
0x172f912e6bd7146e
```

- 测试2：

```
明文：0x12 0x34 0x56 0x78 0x12 0x34 0x56 0x78
密钥：0x10 0x31 0x6E 0x02 0x8C 0x8F 0x3B 0x4A

输出：
./des_crypto
0x82ec79fa7fd7ad61

改动后
明文：0x12 0x34 0x56 0x78 0x12 0x34 0x56 0x78
密钥：0x20 0x31 0x6E 0x02 0x8C 0x8F 0x3B 0x4A

输出：
./des_crypto
0x81ecc519998e257a
```

可见雪崩效应存在。

使用方法

安装好CMake，版本大于等于3.20：

```
$ sudo apt-get install cmake  # Ubuntu 用户
```

```
$ brew install cmake          # macOS 用户
```

可以参阅CMake安装。

Linux或macOS系统下执行以下命令：

```
$ mkdir -p build
$ cd build
$ cmake ..
$ make
$ ./des_crypto
```