个人信息

学号：1911410

姓名：付文轩

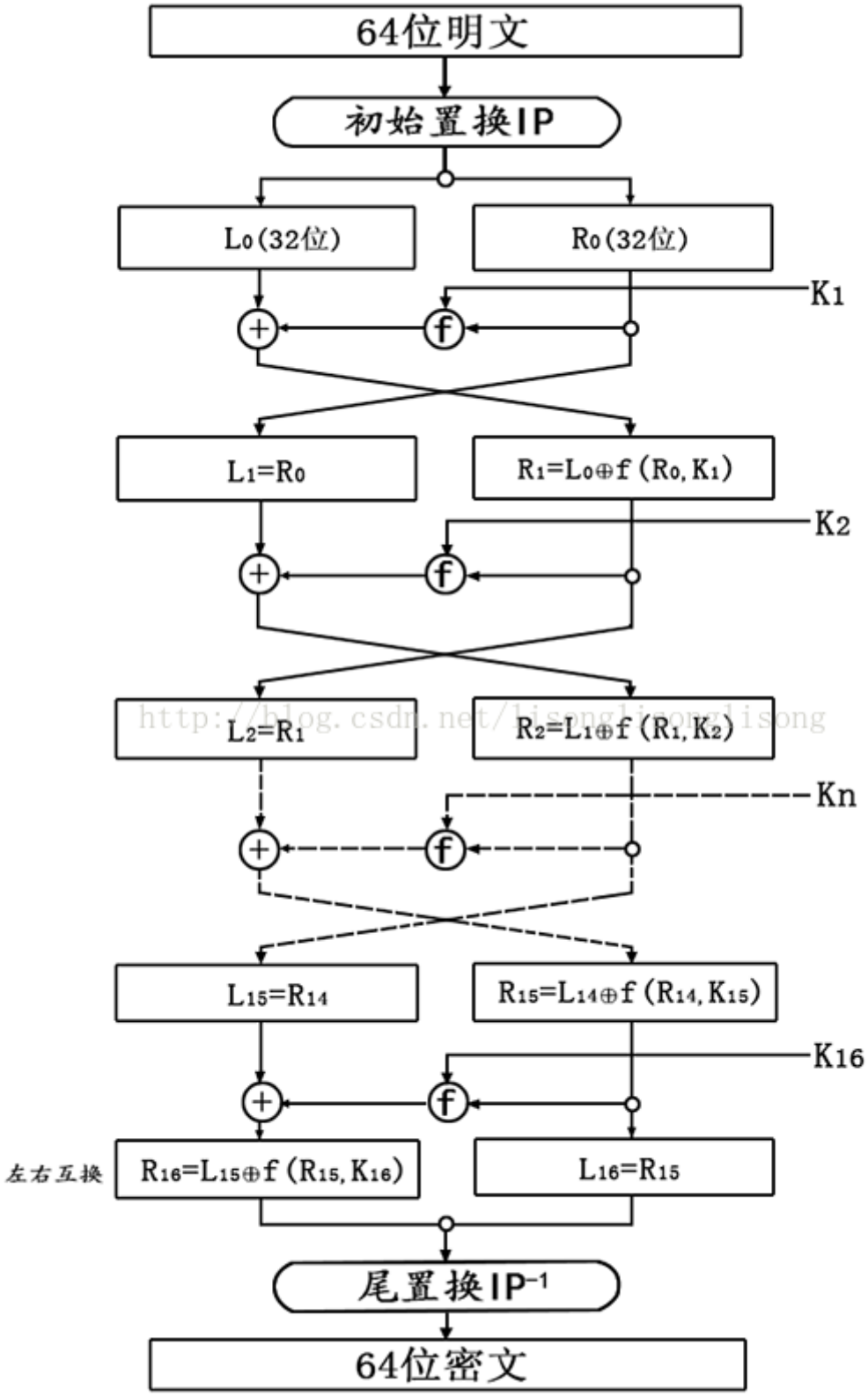
专业：信息安全

实验内容和步骤

1. 对课本中DES算法进行深入分析，对初始置换、E扩展置换，S盒代换、轮函数、密钥生成等环节要有清晰的了解，并考虑其每一个环节的实现过程。
2. DES实现程序的总体设计：在第一步的基础上，对整个DES加密函数的实现进行总体设计，考虑数据的存储格式，参数的传递格式，程序实现的总体层次等，画出程序实现的流程图。
3. 在总体设计完成后，开始具体的编码，在编码过程中，注意要尽量使用高效的编码方式。
4. 利用3中实现的程序，对DES的密文进行雪崩效应检验。即固定密钥，仅改变明文中的一位，统计密文改变的位数；固定明文，仅改变密钥中的一位，统计密文改变的位数。

算法分析

DES加密流程图如下：



根据流程图我认为可以主要分为4个部分：初始IP置换、子密钥的获取、中间主要对明文进行操作的函数f、尾置换IP^{-1}，接下来对这4个部分进行分析

初始IP置换

这一部分比较简单，就是将输入的明文根据一个置换表进行重新排序，最后得到一个64位的输出

置换表为：

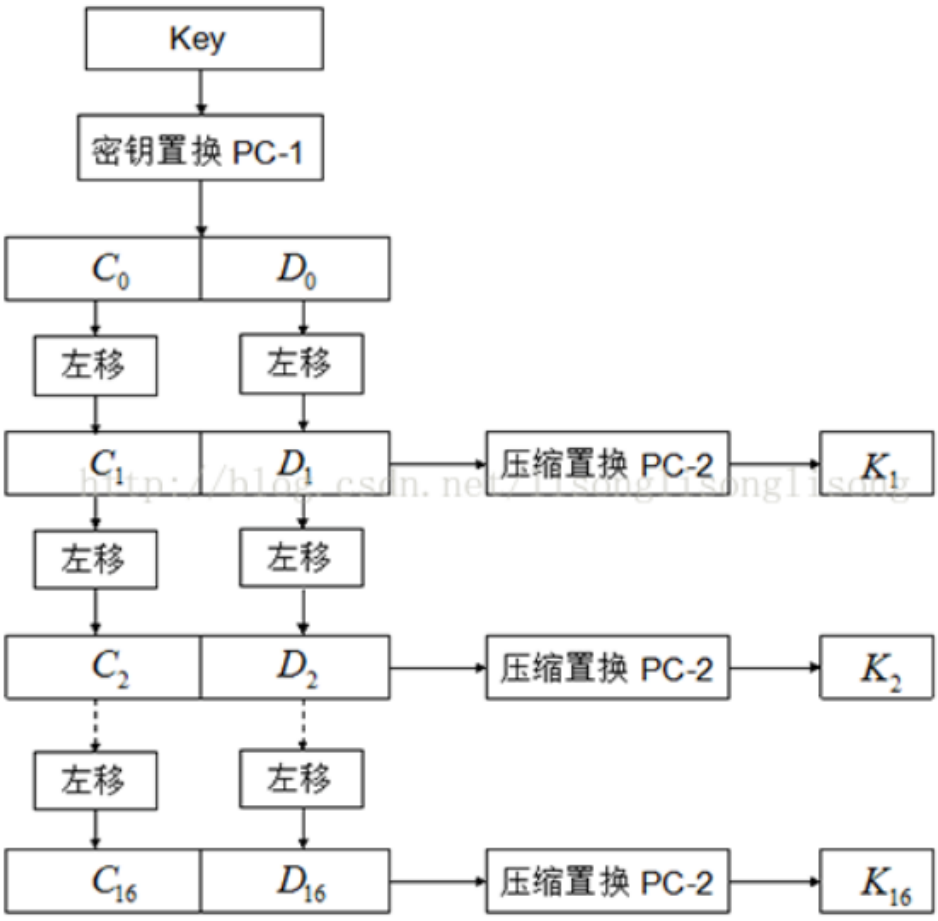
```
int IP[] = { 58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
             62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
             57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
             61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7 };

```

相关代码为：

```
bitset<64> IPReplace(bitset<64> a) {
    bitset<64> result;
    for (int i = 0; i < 64; i++) {
        result[63 - i] = a[64 - IP[i]];
    }
    return result;
}
```

子密钥的获取

子密钥获取的流程图为：



这张流程图可以很清楚的看出：

- 首先Key是由用户输入的（这里书上有一点问题，书上说Key是56位，实际上这里用户输入的应该是64位），在经过一个密钥置换表PC_1作用之后，会把64位的Key变成56位
  PC_1：

```
int PC_1[] = { 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 };
```

- 对得到的56密钥拆分成左右各28位的两部分，然后两部分同时进行**循环左移**

  每轮左移的位数：

  ```
  int shiftBits[16] = { 1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1 };
  ```

  循环左移的辅助函数为：

  ```
  bitset<28> cycleLeftShift(bitset<28> a, int count) {
      bitset<28> tmp = a;
      for (int i = 0; i < 28; i++) {
          a[(i + count) % 28] = tmp[i];
      }
      return a;
  }
  ```

- 将循环左移后的结果拼接成56位，通过PC_2进行压缩置换，得到这一轮的48位子密钥Ki

  PC_2：

  ```
  int PC_2[] = { 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 };
  ```

- 重复2、3步，直到生成共16个48位的子密钥

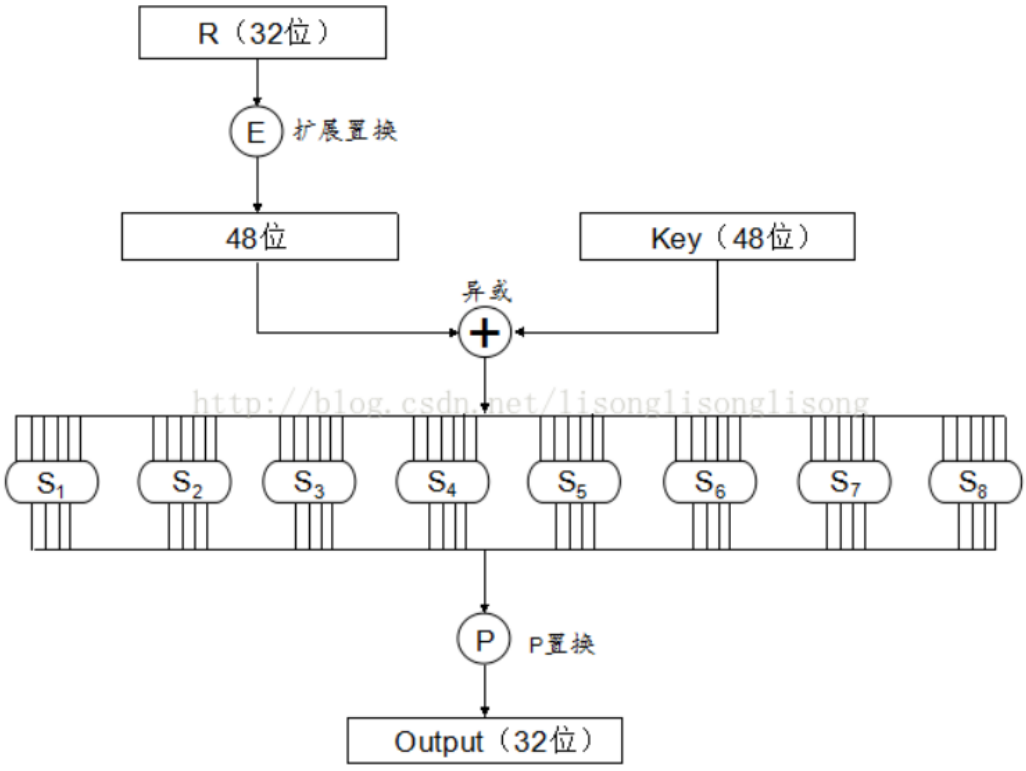
相关代码为：

```
void generateSubKeys() {
    bitset<56> realKey; // 64位的在经过置换以后，会变成56位
    bitset<28> leftOfRealKey; // realKey的左边28位
    bitset<28> rightOfRealKey; // realKey的右边28位
    // 去掉8个校验位，并直接使用PC_1进行代换
    for (int i = 0; i < 56; i++) {
        realKey[55 - i] = key[64 - PC_1[i]];
    }
    //cout << "realKey = " << realKey << endl;
    // 接下来进行16轮的移位操作，生成子密钥
    for (int count = 0; count < 16; count++) {
        // 切分密钥，分成左28和右28
        for (int i = 0; i < 28; i++) {
            leftOfRealKey[i] = realKey[28 + i];
```

```
            rightOfRealKey[i] = realKey[i];
        }
        //cout << "切分后：\nleft: " << leftOfRealKey << endl << "right: " << rightOfRealKey << endl;
        // 进行左移
        leftOfRealKey = cycleLeftShift(leftOfRealKey, shiftBits[count]);
        rightOfRealKey = cycleLeftShift(rightOfRealKey, shiftBits[count]);
        //cout << "循环左移后：\nleft: " << leftOfRealKey << endl << "right: " << rightOfRealKey << endl;
        // 将移位之后的左右两部分，拼回到原本的realKey上
        for (int i = 0; i < 28; i++) {
            realKey[i] = rightOfRealKey[i];
            realKey[i + 28] = leftOfRealKey[i];
        }
        //cout << "拼接回realKey: " << realKey << endl;
        // 经过压缩置换，得到48位的子密钥
        bitset<48> compressKey;
        for (int i = 0; i < 48; i++) {
            compressKey[47 - i] = realKey[56 - PC_2[i]];
        }
        //cout << count << ": " << compressKey << endl;
        // 将得到的子密钥，赋值到数组中去
        subKey[count] = compressKey;
    }
}
```

密码函数f

f的流程图为：



从流程图可以看出f接收的输入是32位的R和48位的Key

- R在经历E的扩展置换后变成48位
  扩展置换表E为：

```
int E[48] = { 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 };
```

- 将扩展后的结果和Key进行异或操作
- 将异或的结果分成8个6位的块，每个块都是6位的输入4位的输出（也就是S盒代换）
  S盒置换表如下：

```
int S_BOX[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}
    }
};
```

- 将S盒代换的结果拼接成32位，进行P置换

```
int 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 };
```

相关代码为：

```
bitset<32> f(bitset<32> R, bitset<48> currentKey) {
    // 1. 进行扩展置换，将32位的数据扩展到48位
    bitset<48> expandR;
    for (int i = 0; i < 48; i++) {
        expandR[47 - i] = R[32 - E[i]];
    }
    // 2. 和当前的key进行异或
    expandR = expandR ^ currentKey;
    // 3. S盒置换
    bitset<32> result;
    int x = 0;
    for (int i = 0; i < 48; i+=6) {
        int row = expandR[47 - i] * 2 + expandR[47 - i - 5];
        int col = expandR[47 - i - 1] * 8 + expandR[47 - i - 2] * 4 +
expandR[47 - i - 3] * 2 + expandR[47 - i - 4];
        int num = S_BOX[i / 6][row][col];
        bitset<4> binary(num);
        result[31 - x] = binary[3];
        result[31 - x - 1] = binary[2];
        result[31 - x - 2] = binary[1];
        result[31 - x - 3] = binary[0];
        x += 4;
    }
    // 4. P置换
    bitset<32> temp = result;
    for (int i = 0; i < 32; i++)
        result[31 - i] = temp[32 - P[i]];

    return result;
}
```

尾置换IP^{-1}

这一步和一开始的初始置换IP从本质上来说没有什么区别，唯一要注意的就是在这一步之前有一个对左右计算得到的结果需要进行一个换位拼接。

尾置换表：

```
int IP_1[] = { 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 };
```

相关代码为：

```
bitset<64> IPReplaceReverse(bitset<64> a) {
    bitset<64> result;
    for (int i = 0; i < 64; i++) {
        result[63 - i] = a[64 - IP_1[i]];
    }
    return result;
}
```

算法设计

加解密算法流程

根据之前的流程图可以很清晰的看出，DES的加密过程就是`初始IP置换 -> 将置换结果分为左右两部分，作用于f函数（共16轮） -> 最终的结果左右互换后合并 -> 尾置换`。其中还有一个步骤是子密钥的生成，这个只要在f函数之前运行就可以。

解密过程则和加密过程完全相同，唯一需要注意的就是在f函数中使用子密钥时，需要逆转顺序使用即可。

数据结构

鉴于本次的加密算法都是按位进行操作的，所以这里的所有数据都按照bit进行存储

在C++中有一个STL：`bitset`，这个STL可以用来操作二进制位，并且对于每个位的访问和数组类似，可以直接使用下角标进行访问，唯一需要注意的是通常情况下我们对于数组的访问都是从左到右，而bitset的访问是从右到左，也就是从低位到高位的顺序。

对于bitset的具体操作这里就不赘述了

代码编写及运行结果

由于本次实验代码较长，就不全贴在实验报告中了，这里仅展示一下main函数，详细代码见工程文件

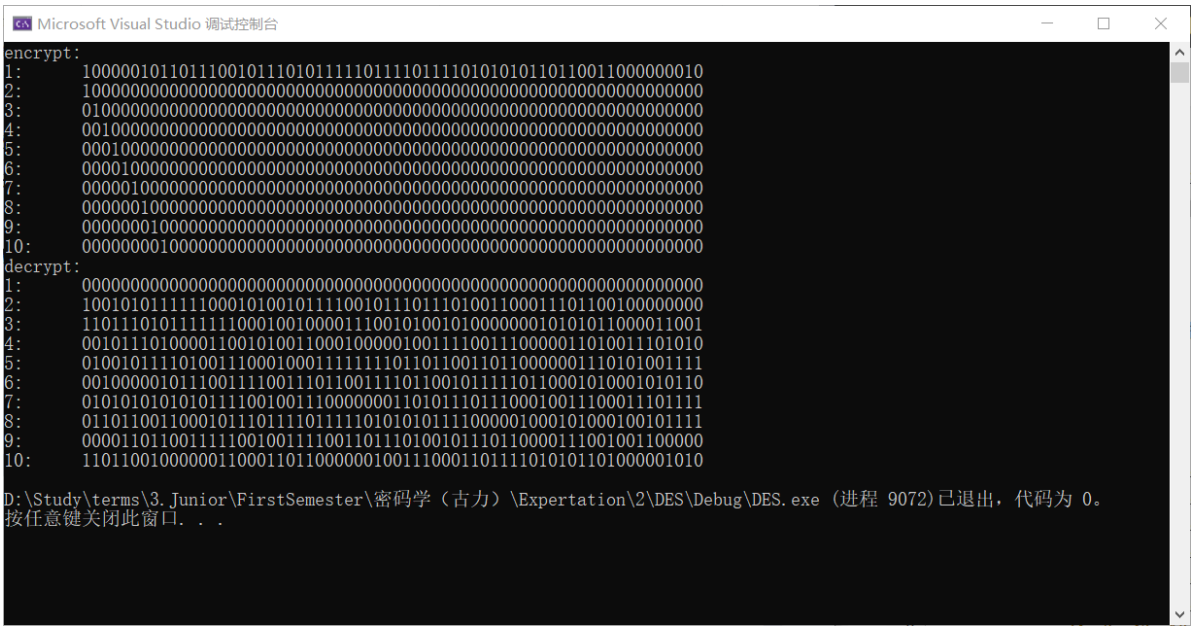
```
int main() {
    //// 一个临时的字符串，用来存放输入
```

```
    //string input;
    //// 获取明文
    //cout << "输入明文\n";
    //getline(cin, input);
    //m = getM(input);

    //// 获取密钥
    //cout << "输入加密密钥\n";
    //getline(cin, input);
    //key = getM(input);


    // 处理测试用例中的struct
    for (int caseNumber = 0; caseNumber < 20; caseNumber++) {
        // 导入一些初始化的数据，虽然这个num可能没有什么用，后面在输出的时候可以体现一些
        int num = cases[caseNumber].num;
        int type = cases[caseNumber].mode;

        // 看类型，type=1是加密，type=0是解密
        if (type) {
            // 加密

            if (num == 1)
                cout << "encrypt:\n";
            // 首先处理结构中的数据，如key等

            // key
            for (int countKey = 0; countKey < 8; countKey++) {
                bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
                for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
                    key[63 - (countKey * 8 + countOfBit)] = tempKey[7 -
countOfBit];
                }
            }

            // txt
            for (int countTxt = 0; countTxt < 8; countTxt++) {
                bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
                for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
                    m[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 -
countOfTxt];
                }
            }

            // 生成子密钥，保存在全局变量的subKey[]中
            generateSubKeys();

            // DES加密
            c = encrypt(m);
            cout << num << ":\t" << c << endl;
        }
        else {
            // 解密，方法同加密
            if (num == 1)
                cout << "decrypt:\n";

            // key
            for (int countKey = 0; countKey < 8; countKey++) {
```

```
                bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
                for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
                    key[63 - (countKey * 8 + countOfBit)] = tempKey[7 -
countOfBit];
                }
            }

            // txt
            for (int countTxt = 0; countTxt < 8; countTxt++) {
                bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
                for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
                    c[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 -
countOfTxt];
                }
            }

            // 生成子密钥，保存在全局变量的subKey[]中
            generateSubKeys();

            // DES解密
            m = decrypt(c);
            cout << num << ":\t" << m << endl;
        }
    }

    return 0;
}
```

其中main函数前半部分是使用字符串类型初始化进行运算的，起初的目的是为了测试中间加密算法的正确性。

运行结果



这里输出的结果是直接将二进制结果进行输出，没有做进制的转换，但是对于结果的正确性没有影响。经过和out中的内容对比之后，可以确定答案是正确的。

雪崩效应测试

测试思路

修改一定位数上的bit，得益于之前存放的形式是bitset，可以直接对某一位进行取反，这样修改非常便捷；

利用已经给出的struct中的结果，将其和修改过后的结果（重新进行相关运算）进行异或操作，然后算出其中有多少个1，就是最终和改变前的结果相差的位数

这里在测试时，仅使用手动输入想要改变的位数的方式（需要修改的位置为`int indexOfTextAva[] = { 1,5,8,6,9,2,3,4 };`）

测试代码

为了测试雪崩效应，将main函数进行修改，得到的main函数如下：

```
int main() {
    //// 一个临时的字符串，用来存放输入
    //string input;
    //// 获取明文
    //cout << "输入明文\n";
    //getline(cin, input);
    //m = getM(input);

    //// 获取密钥
    //cout << "输入加密密钥\n";
    //getline(cin, input);
    //key = getM(input);


    // 处理测试用例中的struct
    for (int caseNumber = 0; caseNumber < 20; caseNumber++) {
        // 导入一些初始化的数据，虽然这个num可能没有什么用，后面在输出的时候可以体现一些
        int num = cases[caseNumber].num;
        int type = cases[caseNumber].mode;

        // 看类型，type=1是加密，type=0是解密
        if (type) {
            // 加密

            if (num == 1)
                cout << "encrypt:\n";
            // 首先处理结构中的数据，如key等

            // key
            for (int countKey = 0; countKey < 8; countKey++) {
                bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
                for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
                    key[63 - (countKey * 8 + countOfBit)] = tempKey[7 - countOfBit];
                }
            }

            // txt
            for (int countTxt = 0; countTxt < 8; countTxt++) {
                bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
                for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
```

```
                    m[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 - countOfTxt];
                }
            }

            // 生成子密钥，保存在全局变量的subKey[]中
            generateSubKeys();

            // DES加密
            c = encrypt(m);
            cout << num << ":\t" << c << endl;



            // 统计雪崩效应
            // 首先改变的是明文
            // 重新统计明文
            for (int countTxt = 0; countTxt < 8; countTxt++) {
                bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
                for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
                    m[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 - countOfTxt];
                }
            }
            // 重新统计密钥
            for (int countKey = 0; countKey < 8; countKey++) {
                bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
                for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
                    key[63 - (countKey * 8 + countOfBit)] = tempKey[7 - countOfBit];
                }
            }
            // 记录应该正常的out
            bitset<64> currectOut;
            for (int countTxt = 0; countTxt < 8; countTxt++) {
                bitset<8> tempTxt(int(cases[caseNumber].out[countTxt]));
                for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
                    currectOut[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 - countOfTxt];
                }
            }
            // 一个数组，用来对相应下角标的内容进行变化
            int indexOfTextAva[] = { 1,5,8,6,9,2,3,4 };
            // 需要改变的下角标的数量
            int lenth = sizeof(indexOfTextAva) / sizeof(indexOfTextAva[0]);
            // 对明文按位取反
            for (int ita = 0; ita < lenth; ita++) {
                m.flip(indexOfTextAva[ita]);
            }
            // DES加密
            c = encrypt(m);
            // 异或，统计结果
            bitset<64> xOut = currectOut ^ c;
            int result = xOut.count();
            numOfAcaM[caseNumber] = result;
            cout << "改变明文" << lenth << "位后，密文改变的位数为：" << result << endl;

```

```
94
95              // 接下来统计改变密钥的雪崩效应
96              // 重新统计明文
97              for (int countTxt = 0; countTxt < 8; countTxt++) {
98                  bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
99                  for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
100                     m[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 -
    countOfTxt];
101                 }
102             }
103             // 重新统计密钥
104             for (int countKey = 0; countKey < 8; countKey++) {
105                 bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
106                 for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
107                     key[63 - (countKey * 8 + countOfBit)] = tempKey[7 -
    countOfBit];
108                 }
109             }
110             // 对密钥按位取反
111             for (int ita = 0; ita < lenth; ita++) {
112                 key.flip(indexOfTextAva[ita]);
113             }
114             // 重新生成子密钥
115             generateSubKeys();
116             // DES加密
117             c = encrypt(m);
118             // 异或，统计结果
119             xOut = currectOut ^ c;
120             result = xOut.count();
121             numOfAcaK[caseNumber] = result;
122             cout << "改变密钥" << lenth << "位后，密文改变的位数为：" << result
    << endl;
123         }
124         else {
125             // 解密，方法同加密
126             if (num == 1)
127                 cout << "decrypt:\n";
128
129             // key
130             for (int countKey = 0; countKey < 8; countKey++) {
131                 bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
132                 for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
133                     key[63 - (countKey * 8 + countOfBit)] = tempKey[7 -
    countOfBit];
134                 }
135             }
136
137             // txt
138             for (int countTxt = 0; countTxt < 8; countTxt++) {
139                 bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
140                 for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
141                     c[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 -
    countOfTxt];
142                 }
143             }
144
145             // 生成子密钥，保存在全局变量的subKey[]中
146             generateSubKeys();
```

```
147
148             // DES解密
149             m = decrypt(c);
150             cout << num << ":\t" << m << endl;
151
152
153             // 统计雪崩效应
154             // 首先改变的是密文
155             // 重新统计密文
156             for (int countTxt = 0; countTxt < 8; countTxt++) {
157                 bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
158                 for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
159                     m[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 -
    countOfTxt];
160                 }
161             }
162             // 重新统计密钥
163             for (int countKey = 0; countKey < 8; countKey++) {
164                 bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
165                 for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
166                     key[63 - (countKey * 8 + countOfBit)] = tempKey[7 -
    countOfBit];
167                 }
168             }
169             // 记录应该正常的out
170             bitset<64> currectOut;
171             for (int countTxt = 0; countTxt < 8; countTxt++) {
172                 bitset<8> tempTxt(int(cases[caseNumber].out[countTxt]));
173                 for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
174                     currectOut[63 - (countTxt * 8 + countOfTxt)] =
    tempTxt[7 - countOfTxt];
175                 }
176             }
177             // 一个数组，用来对相应下角标的内容进行变化
178             int indexOfTextAva[] = { 1,5,8,6,9,2,3,4 };
179             // 需要改变的下角标的数量
180             int lenth = sizeof(indexOfTextAva) / sizeof(indexOfTextAva[0]);
181             // 对密文按位取反
182             for (int ita = 0; ita < lenth; ita++) {
183                 m.flip(indexOfTextAva[ita]);
184             }
185             // DES加密
186             c = encrypt(m);
187             // 异或，统计结果
188             bitset<64> xOut = currectOut ^ c;
189             int result = xOut.count();
190             numOfAcaM[caseNumber] = result;
191             cout << "改变密文" << lenth << "位后，明文改变的位数为: " << result
    << endl;
192
193
194             // 接下来统计改变密钥的雪崩效应
195             // 重新统计密文
196             for (int countTxt = 0; countTxt < 8; countTxt++) {
197                 bitset<8> tempTxt(int(cases[caseNumber].txt[countTxt]));
198                 for (int countOfTxt = 0; countOfTxt < 8; countOfTxt++) {
199                     m[63 - (countTxt * 8 + countOfTxt)] = tempTxt[7 -
    countOfTxt];
```

```
                }
            }
            // 重新统计密钥
            for (int countKey = 0; countKey < 8; countKey++) {
                bitset<8> tempKey(int(cases[caseNumber].key[countKey]));
                for (int countOfBit = 0; countOfBit < 8; countOfBit++) {
                    key[63 - (countKey * 8 + countOfBit)] = tempKey[7 - countOfBit];
                }
            }
            // 对密钥按位取反
            for (int ita = 0; ita < lenth; ita++) {
                key.flip(indexOfTextAva[ita]);
            }
            // 重新生成子密钥
            generateSubKeys();
            // DES加密
            c = encrypt(m);
            // 异或，统计结果
            xOut = currectOut ^ c;
            result = xOut.count();
            numOfAcaK[caseNumber] = result;
            cout << "改变密钥" << lenth << "位后，明文改变的位数为：" << result << endl;
        }
    }

    // 计算雪崩效应的平均位数
    int aveOfChangeM = 0, aveOfChangeMK = 0, aveOfChangeC = 0, aveOfChangeCK = 0;
    for (int i = 0; i < 10; i++) {
        aveOfChangeM += numOfAcaM[i];
        aveOfChangeMK += numOfAcaK[i];
    }
    for (int i = 10; i < 20; i++) {
        aveOfChangeC += numOfAcaM[i];
        aveOfChangeCK += numOfAcaK[i];
    }
    aveOfChangeM = aveOfChangeM / 10;
    aveOfChangeMK /= 10;
    aveOfChangeC /= 10;
    aveOfChangeCK /= 10;
    cout << "\n经统计，雪崩效应的平均改变位数为：\n改明文：" << aveOfChangeM << "\t改明文密钥：" << aveOfChangeMK << "\t改密文："
        << aveOfChangeC << "\t改密文密钥：" << aveOfChangeCK << endl;

    return 0;
}
```

测试结果

运行结果为：

```
encrypt:
1:      1000001011011100101110101111101111011110101010110110011000000010
改变明文8位后，密文改变的位数为：36
改变密钥8位后，密文改变的位数为：27
```

```
2:      10000000000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：25
改变密钥8位后，密文改变的位数为：30
3:      01000000000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：37
改变密钥8位后，密文改变的位数为：32
4:      00100000000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：29
改变密钥8位后，密文改变的位数为：33
5:      00010000000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：33
改变密钥8位后，密文改变的位数为：36
6:      00001000000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：29
改变密钥8位后，密文改变的位数为：35
7:      00000100000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：33
改变密钥8位后，密文改变的位数为：27
8:      00000010000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：35
改变密钥8位后，密文改变的位数为：35
9:      00000001000000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：28
改变密钥8位后，密文改变的位数为：36
10:     00000000100000000000000000000000000000000000000000000000000000000000
改变明文8位后，密文改变的位数为：29
改变密钥8位后，密文改变的位数为：31
decrypt:
1:      00000000000000000000000000000000000000000000000000000000000000000000
改变密文8位后，明文改变的位数为：28
改变密钥8位后，明文改变的位数为：32
2:      10010101111111000101001011110010111011101001100011101100100000000
改变密文8位后，明文改变的位数为：38
改变密钥8位后，明文改变的位数为：34
3:      11011101011111110001001000011100101001010000000101010110000011001
改变密文8位后，明文改变的位数为：26
改变密钥8位后，明文改变的位数为：28
4:      00101110100001100101001100010000010011110011100000110100111010010
改变密文8位后，明文改变的位数为：23
改变密钥8位后，明文改变的位数为：31
5:      01001011110100111000100011111111101101100110110000000111010100111 1
改变密文8位后，明文改变的位数为：34
改变密钥8位后，明文改变的位数为：27
6:      00100000101110011110011101100111101100101111101100010100010100110
改变密文8位后，明文改变的位数为：29
改变密钥8位后，明文改变的位数为：32
7:      01010101010101111001001110000000110101110111000100111000111011111
改变密文8位后，明文改变的位数为：31
改变密钥8位后，明文改变的位数为：34
8:      01101100110001011101111011111101010101111100000100010100010010111 1
改变密文8位后，明文改变的位数为：30
改变密钥8位后，明文改变的位数为：30
9:      00001101100111111001001111001101110100101110110000111001001100000
改变密文8位后，明文改变的位数为：35
改变密钥8位后，明文改变的位数为：32
10:     11011001000000110001101100000010011100011011110101011010000010 10
改变密文8位后，明文改变的位数为：31
改变密钥8位后，明文改变的位数为：31
```

```
63
64 经统计，雪崩效应的平均改变位数为：
65 改明文：31        改明文密钥：32   改密文：30        改密文密钥：31
```