一、 实践目的

完成 DES 加密的四种模式的程序编写，锻炼编程能力的同时加深对 DES 加密的理解。

二、 实践内容

分别实现 ECB、CBC、CFB、OFB 这四种操作模式的 DES。以命令行的形式，指定明文文件、密钥文件、初始化向量文件的位置和名称、加密的操作模式以及加密完成后密文文件的位置和名称。加密时先分别从指定的明文文件、密钥文件和初始化向量文件中读取有关信息，然后按指定的操作模式进行加密，最后将密文（用 16 进制表示）写入指定的密文文件。

三、 实践环境

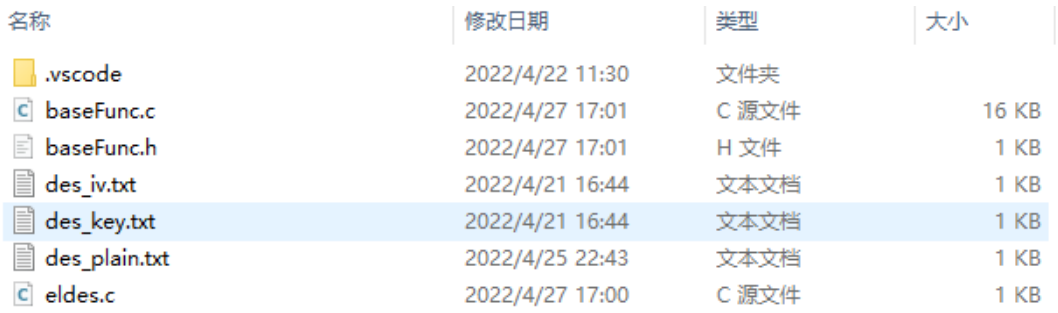
硬件环境：Windows10 系统

软件环境： Visual Studio Code

四、 实践过程与步骤

1.基本加解密实现

首先我们准备好相关文件：如下



名称	修改日期	类型	大小
.vscode	2022/4/22 11:30	文件夹	
baseFunc.c	2022/4/27 17:01	C 源文件	16 KB
baseFunc.h	2022/4/27 17:01	H 文件	1 KB
des_iv.txt	2022/4/21 16:44	文本文档	1 KB
des_key.txt	2022/4/21 16:44	文本文档	1 KB
des_plain.txt	2022/4/25 22:43	文本文档	1 KB
eldes.c	2022/4/27 17:00	C 源文件	1 KB

其中 des_plain.txt 为需要加密的明文文件，des_key 为密钥文件，des_iv 为初始向量文件；eldes.c 为主程序，baseFunc.c 中实现了一些基本的函数，baseFunc.h 是头文件，里面声明了函数的定义。

接下来我们准备编写代码。

首先在 baseFunc.h 文件中，声明我们需要用到的函数名：

```h
#ifndef _BASEFUNC_H_
#define _BASEFUNC_H_

void load(char *buff,char *name);   //读取文件中的内容
void write(char *buff,char *name,int choice);  //往文件中写入内容
void Hex2Bin(int *res,char *s);     //将十六进制字符串转化为二进制整形数组
void Bin2Hex(char *c_16,int *c_2,int len); //将二进制整形数组转化为十六进制字符串
void f(int *r,int *kn);             //f函数
void key_generate(int (*k_16)[80],char *k); //生成子密钥
void encrypt(char *m,int (*kn)[80],char *c);  //基础des加密
void decrypt(char *c,int (*kn)[80],char *m);  //基础des解密
char *zeroPadding(char *m,int *len_m);  //给长度不够的字符串填充零
void XOR(char *m1,char *m2,int len); //两个字符串进行异或
char *ECB_en(char *m,char *k);   //ECB模式加密
char *CBC_en(char *m,char *k,char *v);   //ECB模式加密
char *CFB_en(char *m,char *k,char *v);   //CFB模式加密
char *OFB_en(char *m,char *k,char *v);   //OFB模式加密
char *ECB_de(char *c,char *k);    //ECB模式解密
char *CBC_de(char *c,char *k,char *v); //CBC模式解密
char *CFB_de(char *c,char *k,char *v);  //CFB模式解密
char *OFB_de(char *c,char *k,char *v);  //OFB模式解密

#endif
```

在 baseFunc.c 文件中，实现上面所示的函数

在 baseFunc.c 文件中，定义一些 des 加密需要用到的数据：

```h
//初始置换IP
int pc_ip[80] = {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,
    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
};

//初始逆置换IP-1
int pc_ip_1[80] = {0,
    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
};
```

由于篇幅缘故这里只截取了部分数据，这些表是用于置换，替代用的

然后在 baseFunc.c 文件中实现 DES 加密需要用到的算法

f 函数如下，其中，在 f 函数中完成了对明文块的 E 盒拓展，异或运算，S 盒压缩以及 P 盒置换：

```
/**
 * @brief f函数
 *
 * @param r 明文块，执行函数后被加密为密文
 * @param kn 密钥
 */
void f(int r[80],int kn[80]){
    int x = 0;
    int e[80] = {0};
    int h=0,l=0,idx=0;
    //E盒拓展得到E(Ri)
    for(int i=1; i<=48; i++){
        e[i] = r[pc_e[i]];
    }
    //异或运算得到k^(E(Ri))
    for(int i=1; i<=48; i++){
        r[i] = e[i]^kn[i];
    }
    //S盒压缩将48位分为8组，每组6位，得到S(k^E(Ri))
    for(int i=1; i<=48; i+=6){
        h = r[i]*2 + r[i+5]*1;
        l = r[i+1]*8 + r[i+2]*4 + r[i+3]*2 + r[i+4]*1;
        e[++idx] = (s_box[x][h][l]>>3)&1;
        e[++idx] = (s_box[x][h][l]>>2)&1;
        e[++idx] = (s_box[x][h][l]>>1)&1;
        e[++idx] = (s_box[x][h][l])&1;
        x++;
    }
    //P盒置换
    for(int i = 1; i<=32; i++){
        r[i] = e[pc_p[i]];
    }
}
```

下面是生成子密钥的函数 key_generate，其中，首先将初始密钥进行置换选择 PC-1，然后分为左右两组，进行移位操作后拼接并且再进行置换选择 PC-2，循环十六轮便得到了 16 个子密钥，保存在一个二维数组中，用二进制表示。

```c
/**
 * @brief 生成十六轮子密钥
 *
 * @param k_16 生成的密钥保存在该二维数组中
 * @param k 初始密钥
 */
void key_generate(int (*k_16)[80],char *k){
    int bin_k[80] = {0};
    int bin_kup[80] = {0};
    struct node c_and_d[20];
    for(int i=0;i<20;i++){
        memset(c_and_d[i].c,0,80);
        memset(c_and_d[i].d,0,80);
        memset(c_and_d[i].cd,0,80);
    }
    Hex2Bin(bin_k,k);
    //将密钥用pc_1置换
    for(int i=1; i<=56; i++){
        bin_kup[i] = bin_k[pc_1[i]];
    }
    //左右分组得到C0，D0
    for(int i=1; i<=28; i++){
        c_and_d[0].c[i]=bin_kup[i];
        c_and_d[0].d[i]=bin_kup[i+28];
    }
```

```c
  //16轮生成每轮子密钥
  for(int i=1; i<=16; i++){
      //如果为第1、2、9、16轮，Ci、Di循环左移1位
      if(i==1||i==2||i==9||i==16){
          for(int j=1;j<=27;j++){
              c_and_d[i].c[j] = c_and_d[i-1].c[j+1];
              c_and_d[i].d[j] = c_and_d[i-1].d[j+1];
          }
          c_and_d[i].c[28] = c_and_d[i-1].c[1];
          c_and_d[i].d[28] = c_and_d[i-1].d[1];
      } else {
          //如果为其他轮次，Ci、Di循环左移2位
          for(int j=1;j<=26;j++){
              c_and_d[i].c[j] = c_and_d[i-1].c[j+2];
              c_and_d[i].d[j] = c_and_d[i-1].d[j+2];
          }
          c_and_d[i].c[27] = c_and_d[i-1].c[1];
          c_and_d[i].c[28] = c_and_d[i-1].c[2];
          c_and_d[i].d[27] = c_and_d[i-1].d[1];
          c_and_d[i].d[28] = c_and_d[i-1].d[2];
      }
      //合并每一轮的C、D
      for(int j=1;j<=28;j++){
          c_and_d[i].cd[j] = c_and_d[i].c[j];
          c_and_d[i].cd[j+28] = c_and_d[i].d[j];
      }
  }
  //PC_2置换后获得16轮子密钥
  for(int i=1;i<=16;i++)
      for(int j=1;j<=48;j++)
          k_16[i][j] = c_and_d[i].cd[pc_2[j]];
```

下面我们给出对 64bit 明文加密的函数 encrypt：

首先，我们需要将明文转换为二进制，然后让其进行初始置换 IP，然后进行十六轮变换，然后左右块交换位置，进行初始逆置换 IP-1，得到密文。其中的十六轮变换为：

Li = Ri-1, Ri = Li 异或 f(Ri-1,k)

```c
/**
 * @brief 对明文块进行加密，加密64bit
 *
 * @param m      要加密的明文块
 * @param kn     密钥
 * @param c      所得的密文块
 */
void encrypt(char *m,int (*kn)[80],char *c){
    int ip[80] = {0};
    int res[80] = {0};
    //将明文转化为二进制
    Hex2Bin(res,m);
    //进行初始变换ip,对于
    for(int i=1; i<=64; i++){
        ip[i] = res[pc_ip[i]];
    }
    //初始化得到 l[0]  r[0]
    struct node_1 l_r[20];
    for(int i=0; i<20; i++)
        for(int j=0; j<80; j++) {
            l_r[i].l[j] = 0;
            l_r[i].r[j] = 0;
        }
    for(int i=1; i<=32; i++){
        l_r[0].l[i] = ip[i];
        l_r[0].r[i] = ip[i+32];
    }
```

```c
    //进行十六轮运算
    for(int i=1; i<=16; i++){
        for(int j=1; j<=32; j++){
            //Li = Ri-1得到Li
            l_r[i].l[j]=l_r[i-1].r[j];
        }
        //f函数包含E盒扩展、异或、S盒压缩、P盒置换
        f(l_r[i-1].r,kn[i]);
        //左右合在一起，两者进行最终按位异或得到Ri
        for(int j=1; j<=32; j++){
            l_r[i].r[j]=l_r[i-1].l[j]^l_r[i-1].r[j];
        }
    }
    //将最后得到的L,R合并在一起并交换位置
    int LR[80] = {0};
    for(int i=1; i<=32; i++){
        LR[i] = l_r[16].r[i];
        LR[i+32] = l_r[16].l[i];
    }
    //得到最终变换
    int ans[80] = {0};
    //进行ip逆置换
    for(int i=1; i<=64; i++)
        ans[i] = LR[pc_ip_1[i]];

    //将二进制密文转换为十六进制
    Bin2Hex(c,ans,64);
```

对于解密，算法与加密是一样的，只是十六轮密钥的顺序是倒序。

加密时

```c
//f函数包含E盒扩展、异或、S盒压缩、P盒置换
f(l_r[i-1].r,kn[i]);
```
i从1到16

解密时　　i 从 16 到 1

```c
f(l_r[i-1].r,kn[16-i+1]);
```

2. 四种模式的实现

(1)ECB 模式

ECB 模式就是将明文分成块，一个十六进制字符是 4 位二进制，所以这里以十六个字符为一组明文块进行加密

```
char *ECB_en(char *m,char *k){
    char tmp[16];       //用于暂存明文分组
    int count = 0;
    int len = 0;
    int k_2[20][80] = {0};
    key_generate(k_2,k);
    char *m_std = zeroPadding(m,&len);   //检查待加密字符串大小是否是16的倍数，检查结果为m_std
    char *c = (char*)malloc(sizeof(char)*(len+1)); //用于暂存密文
    memset(c,'\0',len+1);
    //len是待加密的明文的长度
    for(int i = 0; i<len; i++){
        tmp[i%16] = m_std[i];
        //每16个十六进制字符为一组
        if(i%16==15) {
            char c_b[20];
            memset(c_b,'0',20);
            encrypt(tmp,k_2,c_b);       //对该明文分组加密
            for(int j=0; j<16; j++){
                c[count++] = c_b[j];    //保存密文
            }
        }
    }
    return c;
}
```

(2)CBC 模式

CBC 模式与 ECB 模式相比，每一个明文块在加密前需要先与上一个密文块进行异或，第一个明文块与所给的初始向量进行异或

```
char *CBC_en(char *m,char *k,char *v){
    int len = 0,count=0;
    char *m_std = zeroPadding(m,&len);
    //生成子密钥
    int k_2[20][80] = {0};
    key_generate(k_2,k);
    //定义一个存储器来储存密文
    char *c = (char*)malloc(sizeof(char)*(len+1));
    memset(c,'\0',len+1);
    //加密
    char tmp1[17];
    char tmp2[17];
    memset(tmp1,'\0',17);
    memset(tmp2,'\0',17);
    for(int i=0; i<16; i++) tmp2[i] = v[i];
    for(int i=0; i<len; i++){
        //存储明文，16个为一组
        tmp1[i%16] = m_std[i];
        if(i%16==15) {
            //tmp2为上一个密文块,第一次执行时，tmp2是v向量
            //明文块与上一个密文块进行异或
            XOR(tmp1,tmp2,64);
            //加密，tmp2被覆盖为加密后的密文
            encrypt(tmp1,k_2,tmp2);
            for(int j=0; j<16; j++){
                c[count++] = tmp2[j];
            }
        }
    }
    return c;
}
```

与ECB相比，CBC模式在每一个明文块加密前与之前的密文块进行异或

(3)CFB 模式

在 CFB 模式中，明文不直接参与加密，而且明文分组不再是以 16 个字符即 64bit 为一组了，而是以 2 个字符即 8bit 为一组。

为了让每一轮的操作都一样，我定义了一个寄存器 tmp2，一开始将初始向量 v 存储进去。

开始加密时，先对寄存器 tmp2 进行加密，得到 tmp2_c，取 tmp2_c 的前 8bit（即前两个字符）与明文块(8bit)进行异或得到密文块存入 tmp1 中，然后对寄存器 tmp2 中的值进行移位操作，向左移动 8 个 bit（即向左移动两个字符），最后两位则填充进密文块 tmp1(8bit)，这样，一轮加密完成，下一轮加密开始直接重复上述操作。

```
void CFB_en(char *m,char *k,char *v,char *filename){
    int len = 0,count=0;
    char *m_std = zeroPadding(m,&len);
    //生成子密钥
    int k_2[20][80] = {0};
    key_generate(k_2,k);
    //定义一个存储器来储存密文
    char *c = (char*)malloc(sizeof(char)*(len+1));
    memset(c,'\0',len+1);
    //加密
    char tmp1[2];
    char tmp2[17];
    char tmp2_c[17];
    memset(tmp1,'\0',2);
    memset(tmp2,'\0',17);
    memset(tmp2_c,'\0',17);

    for(int i=0; i<16; i++) tmp2[i] = v[i];

    for(int i=0; i<len; i++){
        //存储明文，2个为一组，即8个bit
        tmp1[i%2] = m_std[i];
        if(i%2==1) {
            //tmp2为上一个密文块,第一次执行时，tmp2是v向量
            //对上一个密文块进行加密，第一次执行时，即加密v向量
            encrypt(tmp2,k_2,tmp2_c);
            char tmp2_c_8[2];
            //取分组加密后的前两个十六进制字符，即最左边的8个bit
            for(int j=0;j<2;j++) tmp2_c_8[j] = tmp2_c[j];
            //异或运算
            XOR(tmp1,tmp2_c_8,8);
            for(int j=0; j<2; j++){
                c[count++] = tmp1[j];
            }
            //寄存器向左移位，移动两个十六进制字符，即8bit
            for(int j=0; j<14;j++){
                tmp2[j] = tmp2[j+2];
            }
            tmp2[14] = tmp1[0];
            tmp2[15] = tmp1[1];
        }
    }
    printf("CFB_C:%s",c);
    write(c,filename);
}
```

这里以两个字符为一组，即进行8bit为一组的明文加密

8bit的明文块与加密后的密文块前8bit进行异或

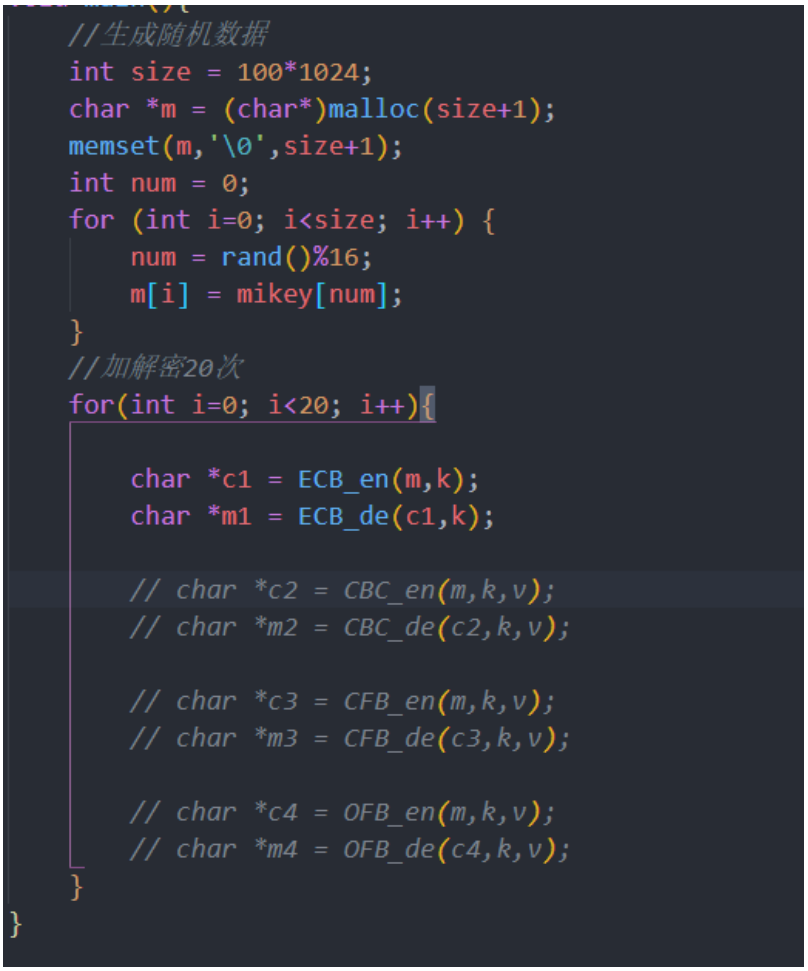
一轮加密完成后，移位

(4)OFB 模式

OFB 模式与 CFB 模式非常相似，只有在寄存器移位的时候稍有不同。从下图可以看到，在寄存器进行移位操作时，最后 8bit 填入的不是密文块 tmp1，而是寄存器加密后得到的 tmp2_c 的前 8bit（即用来与明文块异或的那 8bit）。

3.四种模式的加解密速度测试

编写一个 test.c 文件



执行编译命令 gcc -o test test.c baseFunc.c

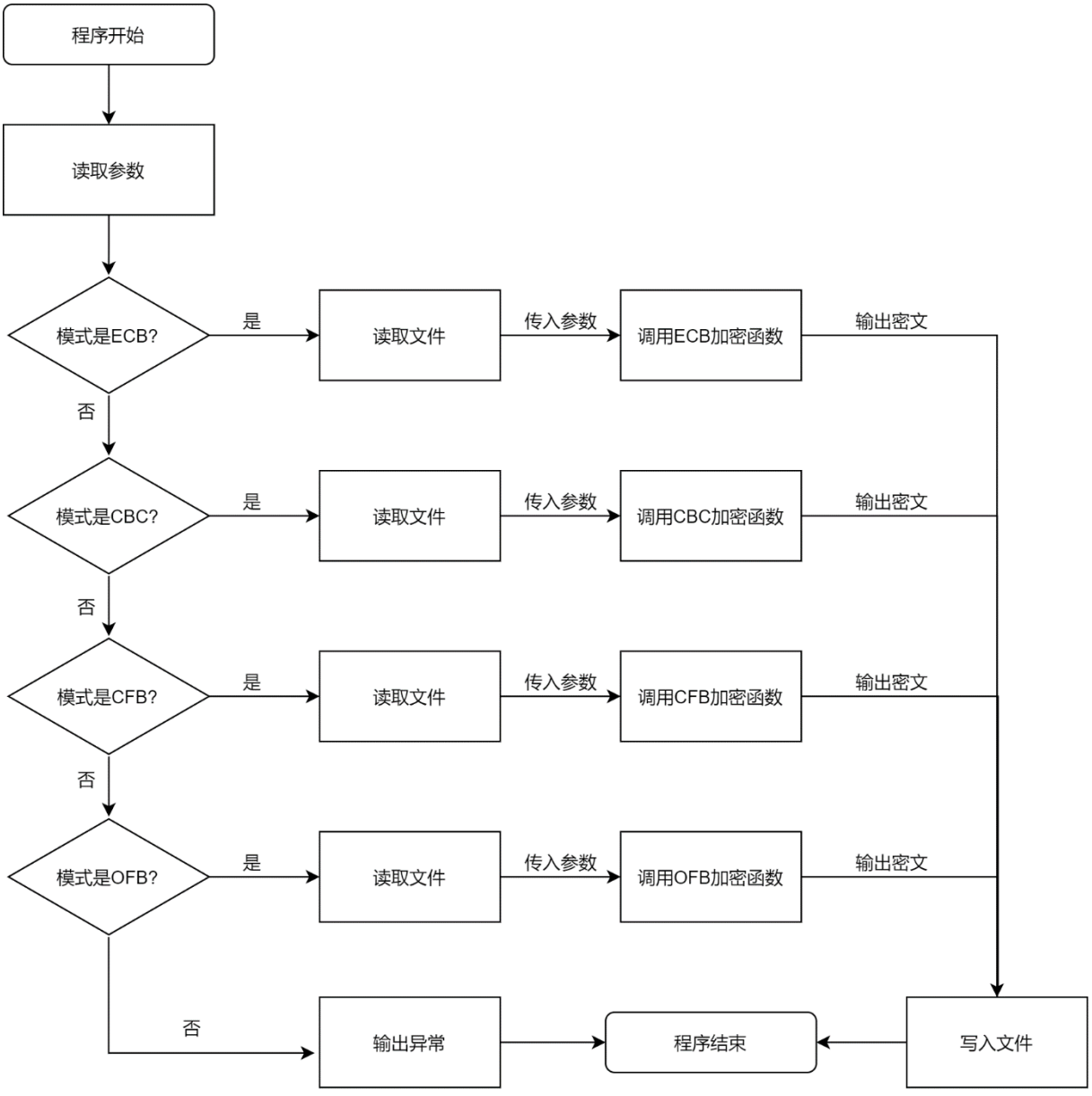
生成可执行文件 test.exe

执行该程序得到结果

实践效果见 六、实践结果与分析

五、 程序设计方案

主程序流程图



数据结构：

用到了许多数组，基本数据类型，指针等等。

自定义的结构体：

```c
//用于生成子密钥时保存左右两边的信息
struct node{
    int c[80];
    int d[80];
    int cd[80];
};
```

```c
//用于加密时保存左右的字符串
struct node_1{
    int l[80];
    int r[80];
};
```

文件读取采用 fscanf

```c
void load(char *buff,char *name){
    FILE *fp = fopen(name,"r");
    fscanf(fp,"%s",buff);
    fclose(fp);
}
```

文件写入采用 fprintf

```c
void write(char *buff,char *name,int choice){
    FILE *fp = NULL;
    if(choice == 0) {
        fp = fopen(name,"w+");
        fprintf(fp,buff);
    } else {
        fp = fopen(name,"a+");
        char buff_1[20];
        memset(buff_1,'\0',20);
        for(int i=0; i<16; i++) buff_1[i] = buff[i];
        buff_1[16] = '\n';
        fprintf(fp,buff_1);
    }
    fclose(fp);
}
```

六、 实践结果与分析

1. 四种模式的加密

程序编写完成后，我们在源文件所在路径打开命令行，输入编译命令

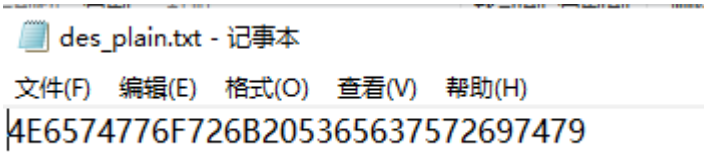
gcc -o eldes eldes.c baseFunc.c

生成可执行程序 eldes.exe

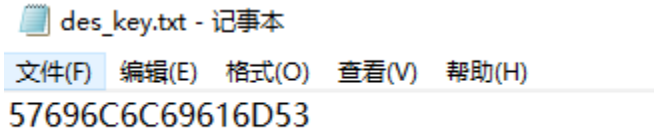
```
问题  输出  终端  调试控制台

PS C:\Users\25026\Desktop\Cipher\TheFirstExp\code> gcc -o eldes eldes.c baseFunc.c
PS C:\Users\25026\Desktop\Cipher\TheFirstExp\code>
```

查看一下我们的文件是否正确

des_plain.txt



des_key.txt

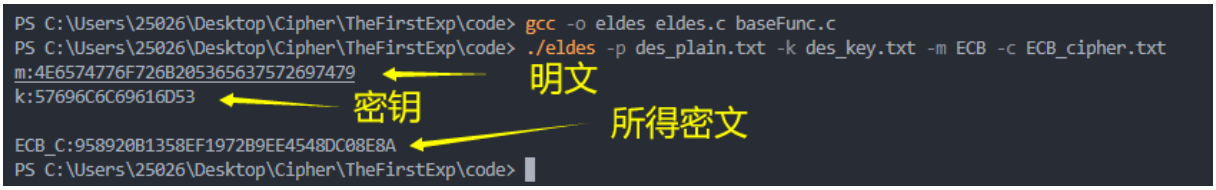


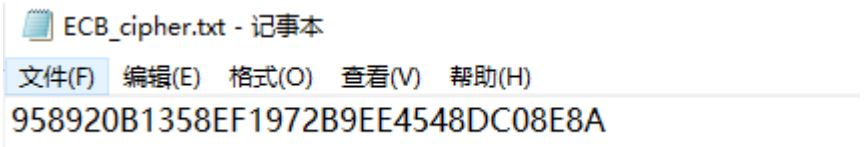
des_iv.txt

（1）ECB 检查：执行以下命令（ECB_cipher.txt 是用来保存密文的文件）

./eldes -p des_plain.txt -k des_key.txt -m ECB -c ECB_cipher.txt

结果如下：





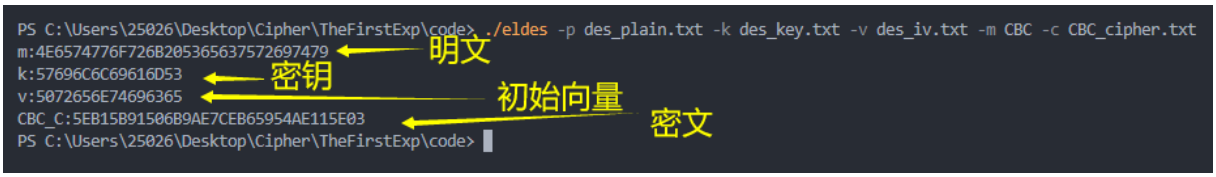
ECB 模式： 958920B1358EF1972B9EE4548DC08E8A

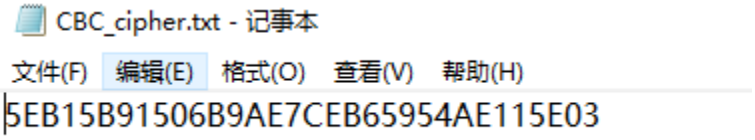
验证正确！

（2）CBC 检查：执行以下命令（CBC_cipher.txt 是用来保存密文的文件）

./eldes -p des_plain.txt -k des_key.txt -v des_iv.txt -m CBC -c CBC_cipher.txt

结果如下：





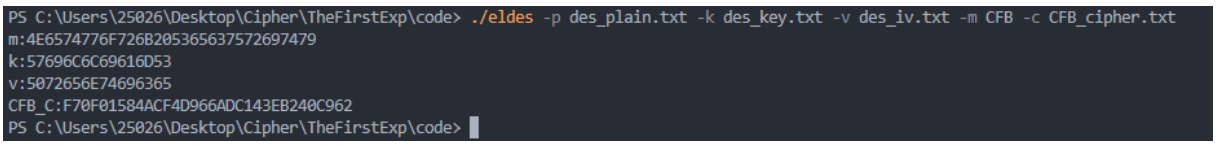
CBC 模式： 5EB15B91506B9AE7CEB65954AE115E03

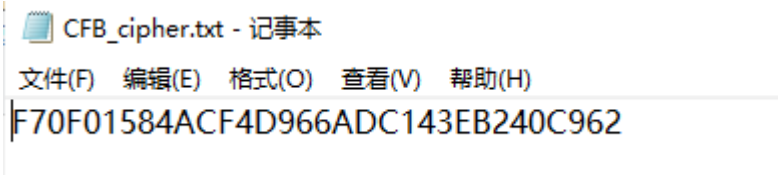
验证成功！

（3）CFB 模式验证：执行以下命令（CFB_cipher.txt 是用来保存密文的文件）

./eldes -p des_plain.txt -k des_key.txt -v des_iv.txt -m CFB -c CFB_cipher.txt

结果如下：





CFB 模式： F70F01584ACF4D966ADC143EB240C962

验证成功！

（4）OFB 模式验证：执行以下命令（OFB_cipher.txt 是用来保存密文的文件）

./eldes -p des_plain.txt -k des_key.txt -v des_iv.txt -m OFB -c OFB_cipher.txt

结果如下：

```
PS C:\Users\25026\Desktop\Cipher\TheFirstExp\code> ./eldes -p des_plain.txt -k des_key.txt -v des_iv.txt -m OFB -c OFB_cipher.txt
m:4E6574776F726B205365637572697479
k:57696C6C69616D53
v:5072656E74696365
OFB_C:F7B0FFCDC0B9BBA76092B929D769417A
PS C:\Users\25026\Desktop\Cipher\TheFirstExp\code>
```

OFB 模式： F7B0FFCDC0B9BBA76092B929D769417A

验证成功！

2. 四种模式的加解密速度测试

ECB 加解密：

```
[Done] exited with code=14409768 in 35.238 seconds
```

CBC 加解密：

```
[Done] exited with code=17817640 in 35.519 seconds
```

CFB 加解密：

```
[Done] exited with code=15196200 in 56.29 seconds
```

OFB 加解密：

```
[Done] exited with code=14999592 in 55.77 seconds
```

从四种模式加解密的时间来看：

ECB 模式与 CBC 模式加解密所需时间相近，说明该两种模式加解密速度差别不大

CFB 模式与 OFB 模式加解密所需时间相近，说明该两种模式加解密速度差别不大

同时可以看到 ECB、CBC 模式加解密所花费的时间比 CFB，OFB 要长，说明 CFB、OFB 加解密速度较 ECB、CBC 相比要慢

总体来说就是对于加解密速度，四种模式有如下关系

$$ECB \approx CBC > CFB \approx OFB$$