姓名：佘崧林

学号：1613574

古典密码算法及攻击方法

实验目的

通过 C++编程实现移位密码和单表置换密码算法，加深对经典密码体制的 了解。并通过对这两种密码实施攻击，了 解对古典密码体制的攻击方法。

实验原理

1. 移位密码 移位密码:将英文字母向前或向后移动一个固定位置。例如向后移动 3 个位置，即对字母表作置换 (不分大小写)。 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z D E F G H I J K L M N O P Q R S T U V W X Y Z A B C 设明文为:public keys,则经过以上置换就变成了:sxeolf nhbv。 如果将 26 个英文字母进行编 码:A→0，B→1，...，Z→25，则以上加密过程 可简单地写成: 明文:m=m1m2...mi..., 则有 密文:c=c1c2... ci..., 其中 ci=(mi+key mod26)，i=1，2，...。
2. 对移位密码的攻击 移位密码是一种最简单的密码，其有效密钥空间大小为 25。因此，很容易用穷举的方法攻 破。穷举密钥攻击是指攻击者对可能的密钥的穷举，也就是用所有可能的密钥解密密文，直到得到有意义的明 文，由此确定出正确的密钥和明文的攻击方法。对移位密码进行穷举密钥攻击，最多只要试译 25 次就可以得到 正确的密钥和明文。
3. 单表置换密码 单表置换密码就是根据字母表的置换对明文进行变换的方法，例如，给定置换 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z H K W T X Y S G B P Q E J A Z M L N O F C I D V U R 明文:public keys, 则有 密文:mckebw qxuo。 单表置换实现的一个关键问题是关于置换表的构造。置换表的构造可以有各种不同的途 径，主要考虑的是记忆的方便。如使用一个短语或句子，删去其中的重复部分，作为置换表的前面的部分，然 后把没有用到的字母按字母表的顺序依次放入置换表中。
4. 对单表置换密码的攻击方法 在单表置换密码中，由于置换表字母组合方式有 26!种，约为 4.03×1026。 所以 采用穷举密钥的方法不是一种最有效的方法。对单表置换密码最有效的攻击方法是利用自然语言的使用频率: 单字母、双字母组/三字母组、短语、词头/词尾等，这里仅考虑英文的情况。英文的一些显著特征如下: 短单词 (small words):在英文中只有很少几个非常短的单词。因此，如果在一个加密的文本中可以确定单词的范围， 那么就能得出明显的结果。一个字母的单词只有 a 和 I。如果不计单词的缩写，在从电子邮件中选取 500k 字节 的样本中，只有两个字母的单词仅出现 35 次，而两个字母的所有组合为 26×26=676种。而且，还是在那个样 本中，只有三个字母的单词出现 196 次，而三个字母的所有组合为 26×26×26=17576 种。 常用单词 (common words):再次分析 500k 字节的样本，总共有 5000 多个不同的单词出现。在这里，9 个最常用的单词 出现的总次数占总单词数的 21%，20个最常用的单词出现的总次数占总单词数的 30%，104个最常用的单词占 50%，247 个最常用的单词占 60%。样本中最常用的 9 个单词占总词数的百分比为: the 4.65 to 3.02 of 2.61 I 2.2 a 1.95 and 1.82 is 1.68 that 1.62 in 1.57 字母频率 (character frequency):在 1M 字节旧的电子文本中，对 字母"A"到"Z"(忽略大小写)分别进行统计。发现近似频率(以百分比表示): e 11.67 t 9.53 o 8.22 i 7.81 a 7.73 n 6.71 s 6.55 r 5.97 h 4.52 l 4.3 d 3.24 u 3.21 c 3.06 m 2.8 p 2.34 y 2.22 f 2.14 g 2.00 w 1.69 b 1.58 v 1.03 k 0.79 x 0.30 j 0.23 q 0.12 z 0.09 从该表中可以看出，最常用的单字母英文是 e 和 t，其他字母使用频率相 对来说就小

得多。这样，攻击一个单表置换密码，首先统计密文中最常出现的字母，并据此猜出两个最常用的字母，并根据英文统计的其他特征(如字母组合)等进行试译。

移位密码

算法原理：

- 明文：$M = m_1 m_2 \dots m_i \dots$
- 密文：$C = c_1 c_2 \dots c_i \dots$, where $c_i = (m_i + key \ mod \ 26)$, $i = 1, 2, \dots$

攻击方法：

- 密钥空间大小为 25，可用穷举法攻击。最多试译 25 次即可得到正确的密文和明文。

算法实现：

```
#include <iostream>
#include <string>

const char *src = nullptr;
char *dst = nullptr;
int length = 0;
int key = 0;

void shift_char(int index) {
    assert(index < length);
    if (src[index] <= 'Z' && src[index] >= 'A') {
        dst[index] = char('A' + (src[index] - 'A' + key) % 26);
    } else if (src[index] <= 'z' && src[index] >= 'a') {
        dst[index] = char('a' + (src[index] - 'a' + key) % 26);
    }
}

void shift_all() {
    assert(src != nullptr && dst != nullptr);
    std::cout << "Plain text:" << src << std::endl;
    for (int index = 0; index < length; index++) {
        shift_char(index);
    }
    dst[length] = '\0';
    std::cout << "Cipher text:" << dst << std::endl;
}

int main() {
    std::string input;
    std::cout << "text key> ";
    while (std::cin >> input >> key) {
        length = input.length();
        delete []dst;
        src = input.c_str();
```

```
        dst = new char[length + 1];
        shift_all();
        std::cout << "text key> ";
    }
}
```

结果：

```
text key> niaho 5
Plain text:niaho
Cipher text:snfmt
```

单表置换密码

算法原理

- 单表置换密码就是根据字母表的置换对明文进行变换的方法。 单表置换实现的 一个关键问题是关于置换表的构造。置换表的构造可以有各种不同的途径，主要考虑的是记忆的方便。如使用一个短语或句子，删去其中的重复部分，作为置换表的前面的部分，然后把没有用到的字母按字母表的顺序依次放入置换表中。

攻击方法：对字母进行频率分析，并根据英文统计的其他特征（如字母组合等）进行试译。

```
e  11.67   t  9.53   o  8.22   i  7.81   a  7.73   n  6.71  s  6.55
r  5.97    h  4.52   l  4.3    d  3.24   u  3.21   c  3.06  m  2.8
p  2.34    y  2.22   f  2.14   g  2.00   w  1.69   b  1.58  v  1.03
k  0.79    x  0.30   j  0.23   q  0.12   z  0.09
```

算法实现：

```
//
// Created by 佘崧林
//
#include <iostream>
#include <map>
#include <string>

const std::map<char, char> conv_table = {
        {'a', 'z'}, {'b', 'y'}, {'c', 'x'}, {'d', 'w'}, {'e', 'v'},
        {'f', 'u'}, {'g', 't'}, {'h', 's'}, {'i', 'r'}, {'j', 'q'},
        {'k', 'p'}, {'l', 'o'}, {'m', 'n'}, {'n', 'm'}, {'o', 'l'},
        {'p', 'k'}, {'q', 'j'}, {'r', 'i'}, {'s', 'h'}, {'t', 'g'},
        {'u', 'f'}, {'v', 'e'}, {'w', 'd'}, {'x', 'c'}, {'y', 'b'},
        {'z', 'a'},
};
const char *src = nullptr;
char *dst = nullptr;
int length;
```

```
void repl_char(int index) {
    assert(index < length);
    if (src[index] <= 'z' && src[index] >= 'a') {
        dst[index] = conv_table.at(src[index]);
    } else if (src[index] <= 'Z' && src[index] >= 'A') {
        dst[index] = char('A' + (conv_table.at('a' + (src[index] - 'A')) -
'a'));
    } else {
        dst[index] = src[index];
    }
}

void repl_all() {
    assert(src != nullptr && dst != nullptr);
    std::cout << "Plain text:" << src << std::endl;
    for (int index = 0; index < length; index++) {
        repl_char(index);
    }
    dst[length] = '\0';
    std::cout << "Cipher text:" << dst << std::endl;
}

int main () {
    std::string input;
    std::cout << "text: ";
    while( std::getline(std::cin, input) ) {
        src = input.c_str();
        length = input.length();
        delete []dst;
        dst = new char[length + 1];
        repl_all();
        std::cout << "text: ";
    }
    return 0;
}
```

运行结果：

```
Plain text:Single table inheritance is a way to emulate object-oriented inheritance in a relational database. When mapping from a database table to an object in an object-oriented language, a field in the database identifies what class in the hierarchy the object belongs to.[1] All fields of all the classes are stored in the same table, hence the name "Single Table Inheritance". In Ruby on Rails the field in the table called 'type' identifies the name of the class. In Hibernate (Java) and Entity Framework this pattern is called Table-Per-Class-Hierarchy and Table-Per-Hierarchy (TPH) respectively.,[2][3] and the column containing the class name is called the Discriminator column.

Cipher text:Hrmtov gzyov rmsvirgzmxv rh z dzb gl vnfozgv lyqvxg-lirvmgvw rmsvirgzmxv rm z ivozgrlmzo wzgzyzhv. Dsvm nzkkrmt uiln z wzgzyzhv gzyov gl zm lyqvxg rm zm lyqvxg-lirvmgvw ozmtfztv, z urvow rm gsv wzgzyzhv rwvmgrurvh dszg xozhh rm gsv srvizixsb gsv lyqvxg yvolmth gl.[1] Zoo urvowh lu zoo gsv xozhhvh ziv hglivw rm gsv hznv gzyov, svmxv gsv mznv "Hrmtov Gzyov Rmsvirgzmxv". Rm Ifyb lm Izroh gsv urvow rm gsv gzyov xzoovw 'gbkv' rwvmgrurvh gsv mznv lu gsv xozhh. Rm Sryvimzgv (Qzez) zmw Vmgrgb Uiznvdlip gsrh kzggvim rh xzoovw Gzyov-Kvi-Xozhh-Srvizixsb zmw Gzyov-Kvi-Srvizixsb (GKS) ivhkvxgrevob.,[2][3] zmw gsv xlofnm xlmgzrmrmt gsv xozhh mznv rh xzoovw gsv Wrhxirnrmzgli xlofnm.
```

词频分析

程序：

```
//
// Created by 佘崧林
//

#include <iostream>
#include <vector>
#include <map>
#include <string>

const std::vector<char> alphabet_to = {'e', 't', 'o', 'i', 'a', 'n', 's', 'r',
'h', 'l', 'd', 'u', 'c',
                                       'm', 'p', 'y', 'f', 'g', 'w', 'b', 'v',
'k', 'x', 'j', 'q', 'z', };
std::map<char, char> conv_table;

const char *src = nullptr;
char *dst = nullptr;
int length = 0;

int count_char(char message) {
    assert(message >= 'a' && message <= 'z');
    char Message = char('A' + (message - 'a'));
    int counter = 0;
    for (int i = 0; i < length; i++) {
```

```
        if (src[i] == message || src[i] == Message) counter++;
    }
    return counter;
}

void revert_map() {
    std::vector<int> counter_vec;
    counter_vec.reserve(26);
    for (int alpha = 0; alpha < 26; alpha++) {
        counter_vec.push_back( count_char(char('a' + alpha)) );
    }

    for (int alpha = 0; alpha < 26; alpha++) {
        int max_pos = alpha, max_count = counter_vec.at(alpha);
        for (int i = alpha + 1; i < 26; i++) {
            if (counter_vec.at(i) > max_count) {
                max_count = counter_vec.at(i), max_pos = i;
            }
        }
        std::swap(counter_vec[alpha], counter_vec[max_pos]);
        conv_table[char('a' + max_pos)] = alphabet_to[alpha];
        std::cout << alpha << " " << char('a' + max_pos) << " " <<
alphabet_to[alpha] << std::endl;
    }
}

void repl_char(int index) {
    assert(index < length);
    if (src[index] <= 'z' && src[index] >= 'a') {
        dst[index] = conv_table[ src[index] ];
    } else if (src[index] <= 'Z' && src[index] >= 'A') {
        dst[index] = char('A' + (conv_table['a' + (src[index] - 'A')] - 'a'));
    } else {
        dst[index] = src[index];
    }
}

void repl_all() {
    assert(src != nullptr && dst != nullptr);
    std::cout << "Plain text:" << src << std::endl;
    for (int index = 0; index < length; index++) {
        repl_char(index);
    }
    dst[length] = '\0';
    std::cout << "Cipher text:" << dst << std::endl;
}

int main () {
    std::string input;
```

```
    std::cout << "text: ";
    while( std::getline(std::cin, input) ) {
        src = input.c_str();
        length = input.length();
        delete []dst;
        dst = new char[length + 1];
        revert_map();
        repl_all();
        std::cout << "text: ";
    }
    return 0;
}
```

效果不太理想，再手动分析可知：

```
Single table inheritance is a way to emulate object-oriented inheritance in a relational database. When mapping from a database table to an object in an object-oriented language, a field in the database identifies what class in the hierarchy the object belongs to.[1] All fields of all the classes are stored in the same table, hence the name "Single Table Inheritance". In Ruby on Rails the field in the table called 'type' identifies the name of the class. In Hibernate (Java) and Entity Framework this pattern is called Table-Per-Class-Hierarchy and Table-Per-Hierarchy (TPH) respectively.,[2][3] and the column containing the class name is called the Discriminator column.
```