プログラミングⅢ 課題

作ったもの

第二次世界大戦時にドイツ軍が採用した暗号機エニグマを再現した。

対応する文字は[a-z]の26文字。空白文字はあらかじめ決めた単語等で代用する。

入力した文字に対してある文字が出力される。特定の設定においてn番目の変換は単一換字式暗号になっており、暗号化された文字列を暗号時と同一の設定で入力し直すことで復号化することができる。

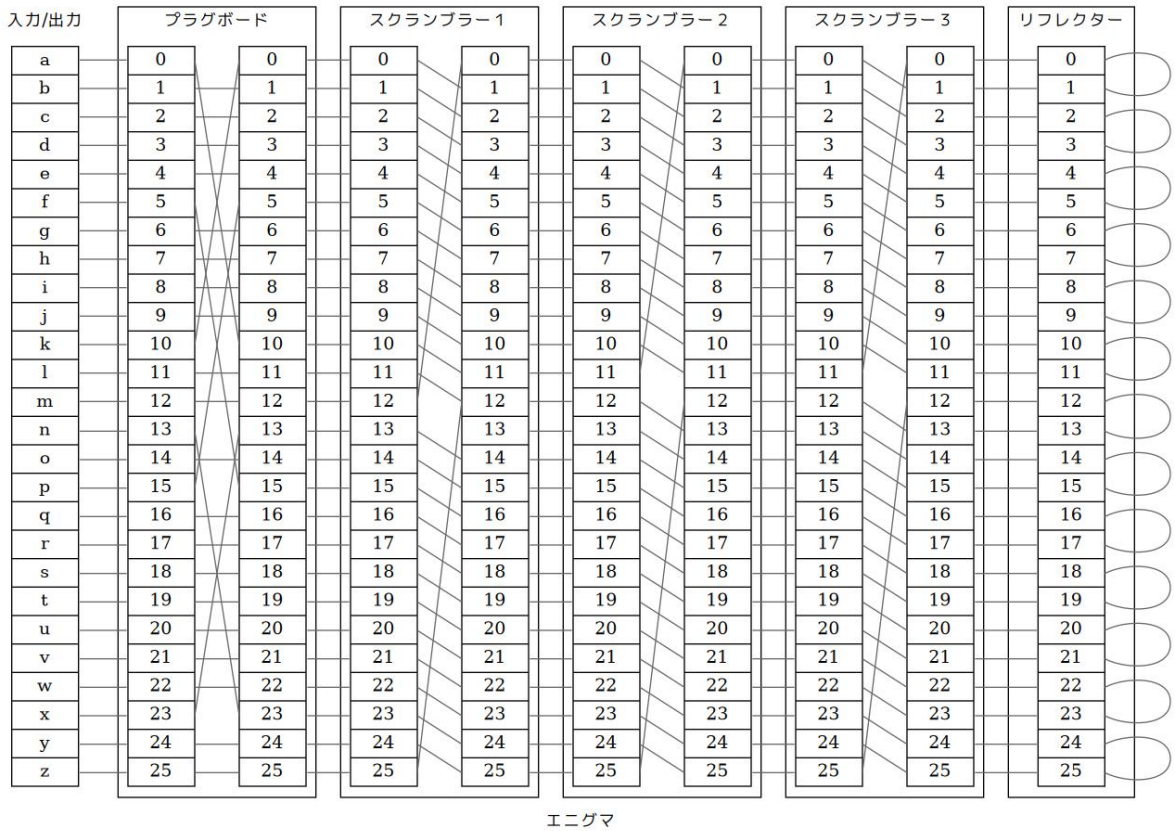
今回、エニグマの設定ファイルを読込、キーボート入力で文字を受取、変換した文字をファイル出力するプログラムを作成した。

エニグマの仕組み

エニグマは大まかに

- プラグボード
- スクランブラー
- リフレクター

によって構成される。



エニグマ

プラグボード

プラグボードは特定の2文字を入れ替える構造を持ち、最大3組の置換を設定することができる。

この設定は任意で変更できるため、${}_{26}C_2 \times {}_{24}C_2 \times {}_{22}C_2$通りのパターンが存在する。

リフレクター

入力信号を1対1対応した信号に変換する。この仕組みによって特定の設定における単一換字方式を実装しており、暗号化と復号化を同一手順で行うことができるようになっている。

※この変換基盤は変更することができない。

スクランブラー

ひとつのエニグマに複数搭載されている機構で文字変換を行う。

円状をしており、

- 1つめのスクランブラーは1文字打つごとに1/26回転
- 2つめ以降のスクランブラーは前のスクランブラーが1回転した際に1/26だけ回転

することで変換表を切り替える。変換のパターン数は3つのスクランブラー搭載で26^3通りにもなる。

プログラムの解説

プログラムは、

- `enigma.c`
- `roter.c`, `rotor.h`
- `keyboard.c`, `keyboard.h`
- `plugs.txt`, `rots.txt`

で構成される。

rotor.h

プラグボード, スクランブラー, リフレクターの基となる構造体`Rotor`を定義。

```
typedef struct {
    // 回転状態
    int rot;
    // 1回転時のフラグ
    int fullturn;
    // 変換表と逆変換表
    int table[26];
    int rtable[26];
} Rotor;
```

配列`table`, `rtable`はそれぞれ変換表と逆変換表を表し、

```
table[IN] = OUT;
rtable[OUT] = IN;
```

のような対応関係になっている。

回転状態`rots`と1回転時のフラグ`fullturn`はスクランブラーの状態保持に用いる。

また、各パーツの初期化関数を定義。

```
Rotor plugboard(int swaps[][2]);
Rotor scrambler(int seed, int rot);
Rotor reflector(int seed);
```

スクランブラーの回転, 1回転判定用の関数と変換表初期化時に用いる関数を定義。

```
// スクランブラーの回転用
void rotate(Rotor *sc);
// 1回転フラグの取得
int fullturn(Rotor *sc);
// 変換表生成に用いる
void shuffle(int arr[], int size, int seed);
```

rotor.c

各関数の実装を行う。

プラグボードの初期化

```
Rotor plugboard(int swaps[][2]) {
    Rotor rt;
    // 無変換表
    int i;
    for (i = 0; i < 26; i++)  rt.table[i] = i;
    // 引数で受け付けたキーの組み合わせで入れ替え
    // 0 ~ 25, a ~ z の両方に対応
    int size = sizeof(*swaps) / sizeof(*swaps[0]);
    for (i = 0; i < size; i++) {
        rt.table[swaps[i][0] % 'a'] = swaps[i][1] % 'a';
        rt.table[swaps[i][1] % 'a'] = swaps[i][0] % 'a';
    }
    return rt;
}
```

引数に置換する2文字のペアを2次元配列で受取、入力と出力が同一の変換表`table`を置換

この変換は2文字ペアになり`table`と`rtable`が同一となるので逆変換表`rtable`は用いない。

スクランブラーの初期化

```
Rotor scrambler(int seed, int rot) {
    Rotor rt;
    // 初期回転状態
    rt.rot = rot;
    rt.fullturn = 0;
    // 変換表を生成
    int i;
    for (i = 0; i < 26; i++)  rt.table[i] = i;
    shuffle(rt.table, 26, seed);
    // 逆変換表の生成
    for (i = 0; i < 26; i++)  rt.rtable[rt.table[i]] = i;

    return rt;
}
```

初期状態でのスクランブラーの回転角は設定できる項目であり、引数で受け取った値を用いる。

変換表は引数で受け取ったシードを基にランダムで生成する。`shuffle`関数は与えられた配列の要素をシードに基づいて入れ替える。これによって特定の変換表を持つスクランブラーを生成できる。

```
void shuffle(int arr[], int size, int seed) {
    // シードで変換表を固定
    srand(seed);
    int i;
    for (i = 0; i < size; i++) {
        int j = rand() % size;
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

また、スクランブラーの回転を行う関数`rotate`はRotorのポインタを引数に受取、スクランブラーの変換表をひとつずらし、回転状態を更新, 1回転（rots = 0）したら`fullturn`を1にする。

```
void rotate(Rotor *rt) {
    // 回転状態を更新
    rt->rot = (rt->rot + 1) % 26;
    // 1回転したらフラグを立てる
    if (rt->rot == 0)  rt->fullturn = 1;
    else rt->fullturn = 0;
    // 変換表の更新
    int tmp = rt->table[0];
    int i;
    for (i = 0; i < 25; i++)  rt->table[i] = rt->table[i + 1];
    rt->table[25] = tmp;
    for (i = 0; i < 26; i++)  rt->rtable[rt->table[i]] = i;
}
```

この1回転の判定はfullturn関数によって行う。Rotorのポインタを受取、`fullturn`が1であれば0に書き換えて1を返す。

```
int fullturn(Rotor *rt) {
    // 1回転のフラグが上がっていたら下げて1を返却
    if (rt->fullturn) {
        rt->fullturn = 0;
        return 1;
    }
    return 0;
}
```

リフレクターの初期化

リフレクターもプラグボードと同様に2文字ペアになっているので逆変換表は用いない。変換表は`shuffle`関数を用いて[0-25]がランダムに並んだ配列を生成、2要素ずつ取り出してそれをペアとする。

```
Rotor reflector(int seed) {
```

```
    Rotor rt;
    // 無変換表の生成
    // 逆変換表は直接用いない
    int i;
    for (i = 0; i < 26; i++)  rt.rtable[i] = i;
    // シードを基にシャッフル
    shuffle(rt.rtable, 26, seed);
    // 配列から2要素ずつ取り出してペアにする
    // リフレクターの対応関係構築
    for (i = 0; i < 26; i += 2) {
        rt.table[rt.rtable[i]] = rt.rtable[i + 1];
        rt.table[rt.rtable[i + 1]] = rt.rtable[i];
    }

    return rt;
}
```

keyboard.c, keyboard.h

LinuxではVC++で利用できる`kbhit`による関数リアルタイム入力監視が実装されていない。

以下のサイトを参考に`kbhit`関数を実装した。

http://tricky-code.net/mine/c/mc06linuxkbhit.php

enigma.c

`main`関数を持つファイル。このエニグマは プラグボードの変換は最大3組, スクランブラーは3つ搭載されている。

Ctrl+Cのシグナル受取時にもファイルを閉じられるように以下の変数, 関数を定義

```
int running = 1;
void handler(int signo) { running = 0; }
```

プラグの入換, スクランブラーの回転状態の初期値は以下のように設定。

```
int swaps[3][2] = {{'a', 'a'}, {'b', 'b'}, {'c', 'c'}};
int scrots[3] = {0, 0, 0};
```

上の設定は`plugs.txt`と`rots.txt`を読込、記述された設定によって上書きする。記述形式は以下の通り。

plugs.txt

```
a,c
e,g
i,k
```

rots.txt

```
1
2
3
```

プラグボード, スクランブラー, リフレクターをそれぞれ生成。スクランブラーの数は`scsize`で保持。

```
Rotor pb = plugboard(swaps);
Rotor sc[3] = {
    scrambler(0, scrots[0]),
    scrambler(1, scrots[1]),
    scrambler(2, scrots[2]),
};
Rotor rf = reflector(3);
```

書込用ファイル`output.txt`を開き、runningが1である間はキー入力受付, 変換処理を行う。シグナルハンドラでCtrl+Cを受け取った際runningを0に変更, また大文字のQの入力を受け取った際にbreakしループを抜け、書込用ファイルを閉じプログラムを終了する。

ループ処理部

入力文字をchに格納、 [a-z]の入力を受けた際、

1. プラグボードを通す: `ch = pb.table[ch - 'a'];`
2. スクランブラーを通す: `for (i = 0; i < scsize; i++)  ch = sc[i].table[ch];`
3. リフレクターを通す: `ch = rf.table[ch];`
4. スクランブラーを通す: `for (i = scsize - 1; i >= 0; i--)  ch = sc[i].rtable[ch];`
5. プラグボードを通す: `ch = 'a' + pb.table[ch];`

以上の処理で暗号化/復号化を行う。

1つめのスクランブラーを回転し、

```
rotate(&sc[0]);
```

前のスクランブラーが1回転した際に隣のスクランブラーを回転

```
for (i = 0; i < scsize - 1; i++) {
    if (fullturn(&sc[i]))  rotate(&sc[i + 1]);
}
```

変換後の文字の表示とファイル書込

```
printf(" -> %c\n", ch);
// ファイル書込
fputc(ch, dfp);
```

どのような技術を使ったか (条件)

- **制御構文の利用：**キー入力受付や規定入力の判定, 各スクランブラーへの同一処理
- **関数の利用：**スクランブラー回転処理や配列要素のシャッフル
- **ポインタの利用：**ファイル型オブジェクトや関数への構造体受け渡し
- **構造体の利用：**各パーツの基盤となる構造体`Roter`の生成
- **ファイル操作：**設定ファイル読込と変換文字の書込
- **250行以上：**$124 + 33 + 6 + 103 + 30 = 296_{\text{行}} \geq 250$

実行例

実行スクリプト run.sh

実行ファイルが見つかればコンパイルせずに実行, 見つからなければコンパイルして実行

```
#!/bin/sh
set -x
if [ ! -e ./enigma ]; then
    echo "実行ファイルenigmaが見つからなかったためコンパイルします"
    gcc ./enigma.c ./rotor.c ./keyboard.c -o enigma
fi
./enigma
```

以下は、「ａｂｃｄｅｆｇ Q(終了)」を入力して暗号化した場合。

```
┌ ~/.../enigma
└ >>> ./run.sh
+ '[' '!' -e ./enigma ']'
+ gcc ./enigma.c ./rotor.c ./keyboard.c -o enigma
+ ./enigma
プラグ設定ファイルplugs.txtを開きました
{a, c} {e, g} {i, k}
初期回転状態設定ファイルrots.txtを開きました
1 2 3
初期化完了...
a -> u
b -> t
c -> s
d -> w
e -> g
f -> d
g -> a
```

暗号化された文字列「ｕｔｓｗｇｄａ」が得られた。逆にこの文字列を入力して復号化する。

```
┌ ~/.../enigma
└ >>> ./run.sh
+ '[' '!' -e ./enigma ']'
+ ./enigma
プラグ設定ファイルplugs.txtを開きました
{a, c} {e, g} {i, k}
初期回転状態設定ファイルrots.txtを開きました
1 2 3
初期化完了...
u -> a
t -> b
s -> c
w -> d
g -> e
d -> f
a -> g
```

元の文字列を得られることが確認できた。

スクランブラーの初期回転状態を変更することで変換が変わる他、同じ文字入力に対して変換された文字が同一にならないことも以下から確認できる。

```
┌ ~/.../enigma
└ >>> ./run.sh
+ '[' '!' -e ./enigma ']'
+ ./enigma
プラグ設定ファイルplugs.txtを開きました
{a, c} {e, g} {i, k}
初期回転状態設定ファイルrots.txtを開きました
13 8 22
初期化完了...
a -> u
a -> s
a -> y
a -> o
```

感想

オートマトンの講義でチューリング機械について紹介され、発明者であるアラン・チューリングの半生を描いた『イミテーション・ゲーム』という映画を知った。この作品で出てきたエニグマについて興味を持ったので今回の作成テーマに決めた。

エニグマは一見複雑なように思えたが、（簡易的であれ）実装してみると非常にシンプルな構造をしていることが分かった。パターン数もスクランブラーやプラグボードでの入換数を増やすことで容易に拡張することができる。また、空白文字を打つことはできず、予め決めた単語等で置き換えるといった手法を用いていたことも学ぶことができた。

C言語での実装は他言語にあるクラスがないために構造体を上手く利用することが重要であると感じた。ポインタはオブジェクトをコピーしたくない場合やコピー元に変更を加えたい場合に用いるものの、配列に対する扱いは十分に理解できているか不安な部分もある。

ファイルからの値の読込は、

```
fscanf(sfp, "%c,%c", &ch1, &ch2)
```

では上手く取得できず、

```
fscanf(sfp, "%c,%c\n", &ch1, &ch2)
```

だと取得できた理由が謎だったので解決したい。