Simulating the Enigma Machine

Deadline: 24th October 2011

Second Year Computing Laboratory
Department of Computing
Imperial College London

You may work in pairs

Introduction

Aims

- To demonstrate your understanding of C/C++ and its libraries.
- To implement a non-trivial program from only an abstract specification.
- To gain an understanding of an important historical `computing machine', and of basic encryption strategies.

Introduction

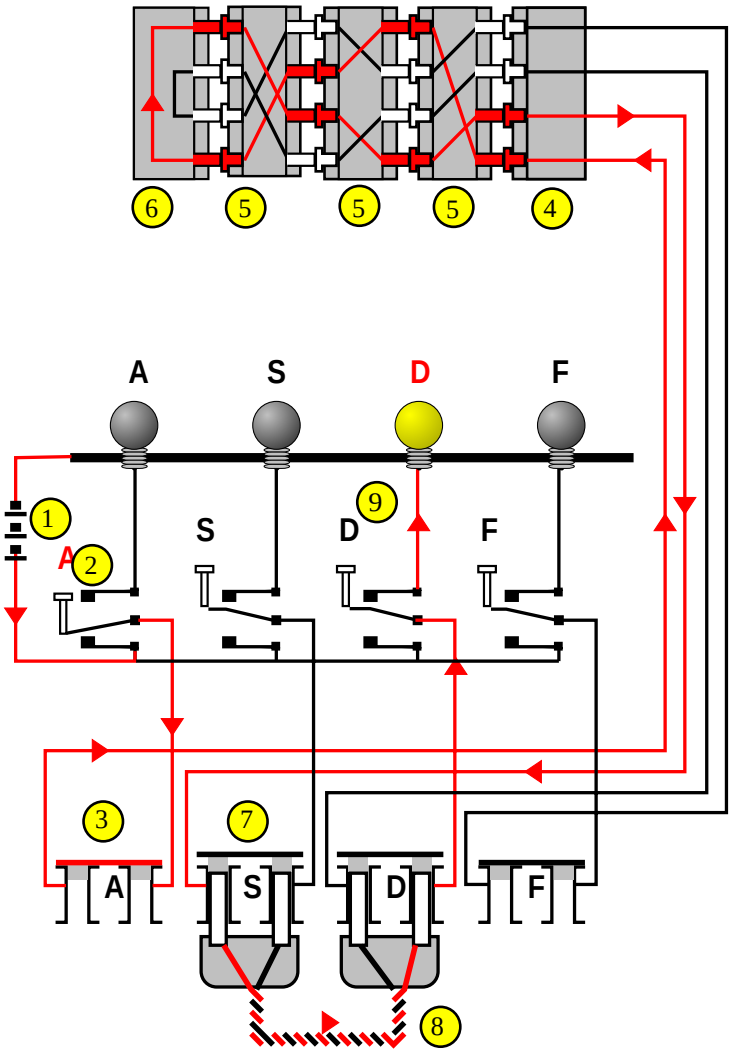
This exercise asks you to implement an *Enigma* machine in C++. Enigma is the common name for the coding machine used by German forces in the Second World War. Two machines set up in the same way allowed the sending of a message securely between their users.

You will need to perform simple input/output operations to configure your Enigma machine from *command line arguments* and *configuration files*. Your Enigma machine should then encrypt (or decrypt) messages provided on the *standard input stream*, outputting the encrypted (or decrypted) message on the *standard output stream*.

You should implement your *Enigma* machine and its components in an object oriented manner using C++ classes and inheritance, if you feel this is appropriate.

The Enigma Machine

An Enigma machine is a device that can encrypt and decrypt messages that are written in a fixed sized alphabet (usually the 26 upper-case Latin letters A-Z). The figure (originally from Wikipedia) shows how a battery powered Enigma machine with a four letter alphabet (ASDF) is wired. The key components of an Enigma machine are a set of *input switches* (2), a *plugboard* (3,7,8), a number of *rotors* (5), a *reflector* (6) and an *output board* (9).



To send a message securely over a public channel (e.g. radio), two identically configured Enigma machines are needed. One operator composes a message and types (2) it one character a time into their Enigma, which causes letters to light up on the output board (9). Another operator writes down this encrypted sequence and transmits it over the radio. At the other end of the radio, an operator receives the encrypted characters and types them into their Enigma machine for decrypting. The decrypted characters can be read off the output board.

Once configured an Enigma machine encrypts using an invertible function, which is why two identically configured machines are needed to securely send and receive a message.

Components

As already mentioned, there are five key components to the Enigma machine. The input switches and output light board are straightforward. We now discuss the rotors, reflector and plugboard in more detail.

Rotors

A rotor (5) is a wheel with the upper-case alphabet in order on the rim and a hole for an axle. On both sides of a rotor are 26 contacts each under a letter. Each contact on one side is wired to a contact on the other side at a different position. A rotor implements an *irreflexive*, *1-1* and *onto* function between the upper-case letters, i.e. each letter is mapped to a different one.

An Enigma machine has a several rotors with different wiring inside each, and which can be arranged in any order on the axle. At the start of the war an Enigma machine had five different rotors available with room on the axle for three. The first part of setting up an Enigma was to put three rotors on the axle in the order set for the day. The second part was to rotate the rotors manually to specified positions (this controls which letter on each rotor is visible).

With three rotors in position there is a connection from each key to the right contact on the first rotor (2) (5) and then through to the left side of the final rotor. The connection then goes through the reflector and then back through the rotors in the reverse order to the lights.

Reflector

The reflector (6) is a device at the end of the rotors which has contacts for each letter of the alphabet on one side only. The letters are wired up in pairs, so that an input current on a letter is reflected back to a different letter.

Plugboard

The plugboard has pairs of wires which can swap input and output letters. In the diagram the plugboard swaps D and S from the keyboard to to the first rotor and also when sending back to the light board. If there is no wire present then it acts as an identity mapping, e.g. in the diagram the key A goes unmodified to the first rotor.

Monoalphabetic Encoding

Writing p for the map of the plugboard, f_n for the map of the nth rotor and r for the map of the reflector, then each input letter x entered into an n-rotor Enigma is translated to:

$$(p \,.\, f_1^{-1} \,.\, f_2^{-1} \,.\, \ldots \,.\, f_n^{-1} \,.\, r \,.\, f_n \,.\, \ldots \,.\, f_2 \,.\, f_1 \,.\, p)(x)$$

The plugboard, rotor and reflector maps have inverses (the plugboard and reflector maps are self-inverse). Therefore entering the output from an Enigma machine into another one with the same set-up returns the original message.

This describes a monoalphabetic encoding, i.e. each letter is mapped to a different one by the machine in a predictable way. This kind of encryption is not too difficult to break. If you know the language a message was written in, and the average frequency distribution of letters in common texts for that language, then the most frequent letters in the message likely gives a partial inversion of the encoding function. Then common words can be deduced and more entries added to the inverse function.

Polyalphabetic Encoding

Polyalphabetic encoding was invented before 1800 by Thomas Jefferson. After a letter has been encoded the encoding map is changed in a regular way (so the intended receiver can decode it). This prevents letter frequency counting from being effective.

The encoding map is changed in an Enigma machine by rotating the rotors after each key is pressed. Specifically after each key press the first rotor is rotated by one step (so e.g. an input on A becomes an input on B). The rotors also featured one or two notches on the rim. When rotating, if a notch was in the correct position then the mechanism that caused the first rotor to rotate would also cause the next rotor along to rotate. If the notch was not in position, only the first (or current) rotor would rotate.

In the simplest set-up, with a notch on the first letter on all rotors, then after 26 steps of the first rotor the second rotor would also be rotated one step, and when the second rotor has completely rotated the third rotor is stepped once. Note that the reflector does not rotate.

For a 3 rotor system, unless a message is longer than 26^3 characters, each character would be encoded with a different map.

What to do

You are to implement a general Enigma machine in C++ as a program that is configured through its command line arguments, and then encrypts / decrypts messages passed to it on the standard input stream, printing the result on the standard output stream.

Normally an Enigma machine is physically limited to requiring a fixed number of rotors, and only a small number of rotors with different wirings exist that can be used. Your program should not have those restrictions.

Inputs and Output

On the command line your program will be invoked with configuration file names as arguments. All but the last configuration file will specify the wiring map for the rotors. Note that there could be any number of rotors, including none at all, and the rotors are specified in order (the first rotor is specified by the first configuration file, etc). The last configuration file will specify the wiring map for the plugboard. There are sample configuration files for the rotors and plugboard in the zip file that comes with the exercise.

So, for example, your program (compiled as `enigma`) could be configured to use three rotors and a sample plugboard as follows:

```
./enigma rotors/I.rot rotors/II.rot rotors/II.rot plugboard/I.pb
```

Here the first rotor would use the mapping described in `I.rot`, and the second and third rotors would use the mapping described in `II.rot`.

Your program will then (in a loop) read input characters from the standard input stream. Whitespace characters (space, tab, carriage-return and newline) should be ignored, upper case characters (A-Z) should be encrypted by the machine with the resulting upper case character output to the standard output stream. All other characters should cause an error to be thrown. Once the standard input stream is closed then your program should exit.

Rotors

The rotor configuration files will contain 26 numbers, separated by white space. The first number will give the (0-based) index into the alphabet that the first letter, 'A', maps to. The second number will give the index for the second letter, 'B', and so on. For example, the sample file `I.rot` contains:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 0

This shifts the alphabet up one when mapped forwards (e.g. $\mathtt{f}_{\mathtt{I.rot}}(0) = 1$, so 'A' becomes 'B'), and maps the alphabet down one when mapped backwards ($\mathtt{f}_{\mathtt{I.rot}}^{-1}(0) = 25$, 'A' becomes 'Z').

However, *after* each character is encrypted the first rotor is rotated by one position. Inputs that would previously have gone through the 'A' mapping will instead go through the 'B' mapping (and 'B' inputs go through the 'C' mapping, etc. all the way up to 'Z' inputs going thorough the 'A' mapping). Don't forget outputs are also shifted as the rotor is rototated - an output on 'B' really is an output on 'A' after one rotation. After 26 rotations of a rotor (i.e. when the rotor returns to its original position) then the next rotor up should also be rotated once.

Reflector

The reflector wiring is not configurable. It maps an input at alphabet index x by the function: $\mathtt{r}(x) = (x + 13) \% 26$. For example, 'A' becomes 'N' and 'M' becomes 'Z'.

Plugboard

The plugboard configuration files will contain an even number (possibly zero) of numbers, separated by white space. The numbers are to be read off in pairs, where each pair specifies one wire in the plugboard connecting two letters. The numbers are, as with the rotors, the (0-based) index into the alphabet.

For example, the sample file `plugboards/III.pb` contains: `23 8 20 22 18 16 24 2 9 12`

This corresponds to the wiring mapping: X - I, U - W, S - Q, Y - C, J - M. All other letters are mapped to themselves.

Hints

- The standard C++ `main` method will be passed in the number of command line arguments as well as a char ** pointer to the arguments themselves. Don't forget that the number of arguments, and the arguments themselves will include the program name in the first argument.
- The rotors, reflector and plugboard all have similar behaviour in terms of mapping one input to another, but not all of them perform this operation in the same way. Could inheritance and overriding model this nicely?
- The modulus operator (%) in C++ can be quite helpful in this exercise, but make sure you understand its behaviour when the first argument is negative.
- To use files for messages redirect the standard input or output: for example: `./enigma II.rot III.rot IV.pb < secret.txt > safe.txt`.
- To strip whitespace from a standard `istream` use the modifier `ws`. For example, if you `\#include <iostream>`, then `cin >> ws` will remove any white space up to the next non-whitespace character, or end of file.

Testing

It is possible to incrementally test your Enigma machine, and you should do so thoroughly. A good place to start is to test with no rotors and no plugboard bindings (use `plugboards/null.pb`). The only change in this case should come from the reflector. Remember that any encrypted text run through the same machine that produced it should always decrypt it back to the original again.

The supplied zip file includes several sample configuration files and a small script that runs some tests.

Unassessed Extensions and Competitions

Once you have successfully implemented your Enigma machine, you should feel free to extend it. For example, you could research and model the notches mechanism described, or add a GUI to visualise the workings of the machine. The best Enigma machine will win a prize, as judged by the lab organisers.

In addition, in the supplied test-suite there is a decryption challenge (see `challenge/README.txt` for details). There will be a prize for the first team to successfully decrypt the message and email it (with a description of how they did it) to `tora@doc.ic.ac.uk`.

Submission

The headers and source files for your program. If you implement any extension or extra features, add a `README.txt` file to your submission that describes them. The tested and marked exercise with feedback will be returned by the 7th November.