Imperial College London
Department of Computing

MSc C++ Programming – Assessed Exercise No. 2

Issued: Tuesday 29 October 2013
Due: **Monday 18 November 2013**

Lab Sessions:

Tuesday 29 October (pm)	Thursday 7 November (am)
Thursday 31 October (am)	Friday 8 November (am)
Friday 1 November (am)	Monday 11 November (am)
Monday 4 November (am)	Thursday 14 November (am)
Tuesday 5 November (pm)	Friday 15 November (am)

Monday 18 November (am)

An Enigma Machine at London's Imperial War Museum

Problem Description

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.

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). Figure 1 shows how a battery powered Enigma machine with a four letter alphabet (ASDF) is wired. The machine is essentially an electric circuit that connects the input switches to the lights on the output board. In between the input switches and output board there is a complex mechanism that continually re-routes the circuit, thus scrambling the original message.

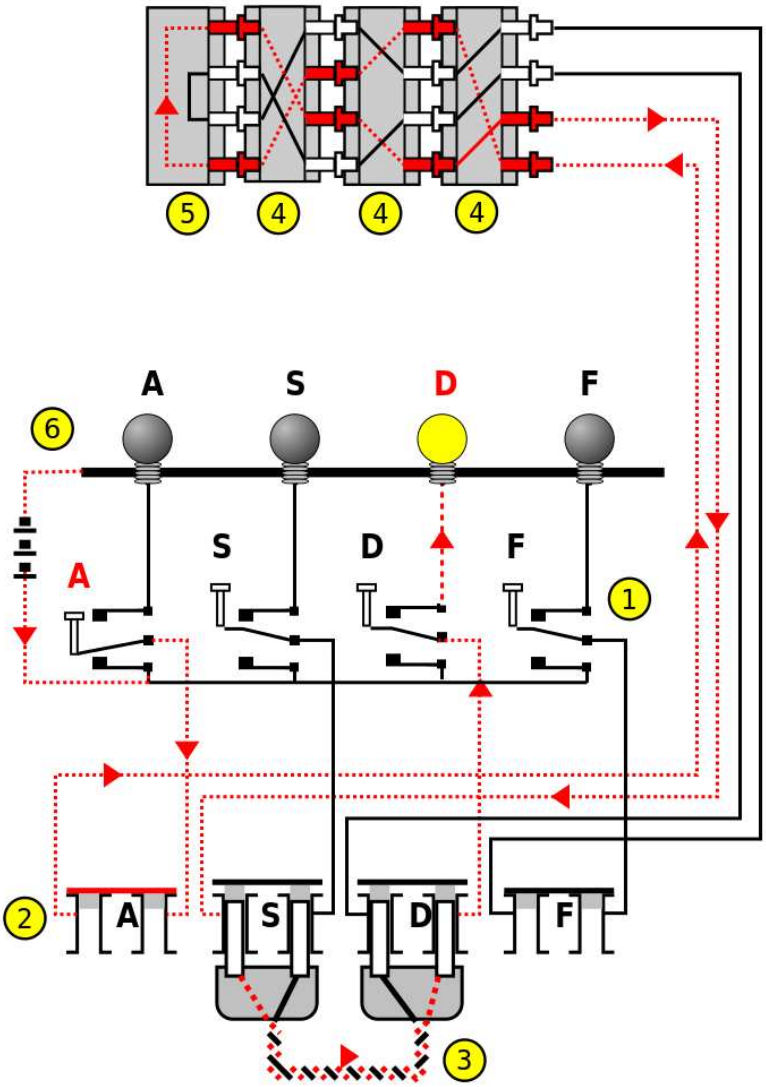


Figure 1: The wiring of an Enigma machine

The key components of an Enigma machine are:

- a set of *input switches* (1);
- a *plugboard* (2) with a number of *plug cables* (3);
- a number of *rotors* (4);
- a *reflector* (5);
- an *output board* (6).

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 it one character a time using their Enigma machine's input switches (2), which causes letters to light up on the output board (6). 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 sequence of characters which he writes down. This sequence can then be typed into the receiver's Enigma machine where the decrypted characters can be read off of the output board.

Once configured, an Enigma machine encrypts its input using an invertible function. Thus, to decrypt a message one needs only to type the encrypted *ciphertext* into an identically configured Enigma machine, and then read the *plaintext* off the output board.

Components

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

Plugboard

When a key on the Enigma machine keyboard is pressed, it closes the electrical circuit, and causes a signal to first be sent to the plugboard. The plugboard consists of a series of contacts that can be connected by plugboard cables. The effect of connecting two letters via plugboard cables is to swap those two letters' input signals before sending them to the rotors. In Figure 1, a plugboard cable swaps the inputs on D and S from the keyboard when sending them to the first rotor and also swaps them when sending a signal back to the output board's lights. If there is no plugboard cable present then the input signal enters the rotors from its original position. For example, in Figure 1 the key A goes unmodified to the first rotor.

Rotors

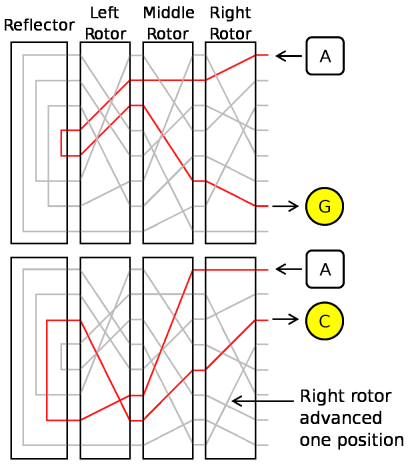
A rotor (4) is a wheel with the upper-case alphabet in order on the rim and a hole for an axle. On both sides of a rotor are 26 contacts each under a letter. Each contact on one side is wired to a contact on the other side at a different position, effectively mapping each letter to a different one.

A diagram demonstrating the wiring of a rotor

An Enigma machine has several rotors with different wiring inside each. These 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 machine 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.

With three rotors in position there is a connection from each key (1) to the right contact on the first rotor (4) and then through each rotor in turn, to the left side of the final rotor. The connection then goes through the reflector (5) and then back through the rotors in the reverse order to the output board's lights (6).

However, as well as simply mapping each input letter to a different output letter, the rotors can also *rotate*. The effect of this rotation is to change the mapping between input and output. As shown in the diagram above, the rotors rotate in the *opposite* direction to the order in which the alphabet letters are inscribed on the rotor's ring. After a rotation of one position, for example, a signal on the 'A' input would actually enter the rotor in the previous 'B' position; if the rotor maps the letter 'B' to the letter 'G', for example, then after the rotation the signal will exit the rotor in the previous 'F' position. This is illustrated by the diagram below



It was this rotation (coupled with the enormous number of possible starting configurations) that made the Enigma code such a powerful encryption, since it implemented a *polyalphabetic* encoding - notice in the diagram above that the two inputs of the letter 'A' are encoded using *different* output letters.

The rotors' rotations were controlled by the input keyboard: every time a key was pressed, a lever would rotate the right hand rotor one position **before** closing the electric circuit. Each rotor also had one or more notches, aligned with the letters on the rim, which would engage the rotor to its left as it turned. Thus, the rotors behaved somewhat like an odometer in a car[1].

Reflector

The reflector (5) 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 one letter is output back to a different letter. As mentioned above, it causes the input signal to be *reflected* back through the rotors and plugboard, and then into one of the bulbs on the output board.

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; similarly with the reflector. Your program should *not* have these restrictions - it should be able to read in details of the rotor, plugboard and reflector wirings from configuration files, whose names are specified on the command line.

You will be provided with a number of sample configuration files. You will also be provided with a header file containing various error codes. You will need to create your own code files. Create a makefile which compiles your submission into an executable file called **enigma**.

[1]The actual rotation mechanism was slightly more complicated than this, however for this exercise we will assume the odometer-like behaviour for simplicity.

Inputs and Outputs

Your program will be invoked on the command line and will be passed configuration file names as arguments. The first configuration file will specify the wiring map for the plugboard, the second file will specify the wiring map for the reflector, and the remaining files will specify the wiring maps for the rotors and the rotors' initial starting positions.

Note that there could be *any* number of rotors, including none at all, and the rotors are specified in order left to right (note, though, that the electric current in an Enigma machines first runs through the rotors right to left, and then left to right after going through the reflector). If an insufficient number of command line parameters are provided, the program should output an informative message to the standard error stream and exit with the `INSUFFICIENT_NUMBER_OF_PARAMETERS` code.

So, for example, your program would be configured to use three rotors and a sample plugboard and reflector as follows:

```
./enigma plugboards/I.pb reflectors/I.rf rotors/I.rot rotors/II.rot rotors/III.rot rotors/I.pos
```

Here the leftmost rotor would use the mapping described in `I.rot`, and the middle and right-hand rotors would use the mapping described in `II.rot` and `III.rot` respectively, with their starting positions specified in rotors/I.pos; the plugboard would use the mapping described in `I.pb` and the reflector the one in `I.rf`.

Your program will then (in a loop) read input characters from the standard input stream. White-space characters (space, tab, carriage-return and newline) should be ignored and 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 the program to output an informative message to the standard error stream and exit with the `INVALID_INPUT_CHARACTER` code. Once the standard input stream is closed, your program should exit normally.

Plugboard

The plugboard configuration files will contain an even number of numbers (possibly zero), separated by white space. The numbers are to be read off in pairs, where each pair specifies a connection made between two contacts on the plugboard. The numbers are 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
```

which corresponds to the plugboard where: 'X' is connected to 'I', 'U' is connected to 'W', 'S' is connected to 'Q', 'Y' is connected to 'C' and 'J' is connected to 'M'. All other letters are mapped to themselves.

Your program should check that this configuration file is well-formed. More specifically:

- if the file attempts to connect a contact with itself, or with more than one other contact, the program should exit with the `IMPOSSIBLE_PLUGBOARD_CONFIGURATION` code;
- if the file contains an odd number of numbers, the program should exit with the `INCORRECT_NUMBER_OF_PLUGBOARD_PARAMETERS` code;
- if the file contains a number that is not between 0 and 25, the program should exit with the `INVALID_INDEX` code;
- if the file contains any characters other than numeric characters, the program should exit with the `NON_NUMERIC_CHARACTER` code.

In case the plugboard configuration file is not well-formed, your code should output an informative error message before exiting. The error codes should be activated as the configuration file is read (e.g. if the file starts with `16 7 7 9 ...` the program should exit with the `IMPOSSIBLE_PLUGBOARD_CONFIGURATION` code, even if it contains an odd number of numbers).

Rotors

When modelling the rotors, it may be useful to keep in mind both a notion of 'absolute' position, such that 'A' goes at the top (i.e. at a 12 o'clock position) and the remaining letters' positions continue round the circumference, and a notion of 'relative' position corresponding to the rotor. So as the rotor rotates, its inputs and outputs move relative to this absolute frame of reference.

The rotor configuration files will contain a sequence of numbers, separated by white space. The first 26 numbers specify the mapping implemented by the rotor: 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. The remaining numbers specify the positions on the rotor where the turnover notches are placed. Thus, if the configuration file specifies that there is a notch at 3 (i.e. the rotor's 'D' position), then when the rotor rotates such that its 'D' position lines up with the top ('A') absolute position, the rotor to its left rotates by one place.

For example, the sample file `shift_up.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 0
```

This shifts the alphabet up one when mapped forwards (so 'A' becomes 'B'), and down one when mapped backwards (so 'A' becomes 'Z'). It also specifies that the rotor to its left should rotate one place whenever 'A' rotates into the top position.

Your program should check that this configuration file is well-formed. More specifically:

- if the file attempts to map more than one input to the same output, or does not provide a mapping for some input, the program should exit with the `INVALID_ROTOR_MAPPING` code;
- if the file contains a number that is not between 0 and 25, the program should exit with the `INVALID_INDEX` code;
- if the file contains any characters other than numeric characters, the program should exit with the `NON_NUMERIC_CHARACTER` code.

In case the rotor configuration file is not well-formed, your code should output an informative error message before exiting. The error codes should be activated as the configuration file is read (e.g. if the file starts with `3 3 ...` the program should exit with the `INVALID_ROTOR_MAPPING` code, even if later on it also specifies a notch at position 27).

The rotors' starting positions should be specified in a separate configuration file. This file will contain a sequence of numbers, the first one corresponding to the starting position of the left-most rotor (e.g. 0 specifies that the rotor's 'A' position coincides with the 'A' position in the absolute frame of reference, and so on), the second corresponding to the starting position of the rotor to its right, etc. Again, your program should check that this configuration file is well-formed, exiting with codes `INVALID_INDEX` and `NON_NUMERIC_CHARACTER` as appropriate. If the configuration file does not contain enough starting positions for the number of rotors specified, your program should exit with the `NO_ROTOR_STARTING_POSITION` code.

The next number will specify the starting position of the rotor.

Reflector

The reflector configuration file, like the plugboard configuration file, contains an even number of numbers separated by whitespace. Unlike the plugboard file however, it must have exactly 13 pairs of numbers. These specify the mapping implemented by the reflector. Each number is again a (0-based) index into the alphabet, and each pair represents a connection, where an input on one member is reflected back as output on the other.

Your program should check that this configuration file is well-formed. More specifically:

- if the file attempts to map an input to itself or pair each index with more than one other, the program should exit with the `INVALID_REFLECTOR_MAPPING` code;
- if the file does not contain exactly 13 pairs of numbers, the program should exit with the `INCORRECT_NUMBER_OF_REFLECTOR_PARAMETERS` code;
- if the file contains a number that is not between 0 and 25, the program should exit with the `INVALID_INDEX` code;

- if the file contains any characters other than numeric characters, the program should exit with the `NON_NUMERIC_CHARACTER` code.

In case the reflector configuration file is not well-formed, your code should output an informative error message before exiting. The error codes should be activated as the configuration file is read (e.g. if the file starts with `0 ...` the program should exit with the `INVALID_REFLECTOR_MAPPING` code, even if later on it also contains a non-numeric character).

Returning Error Codes

Your program should process the cofiguration files in the order that they are given on the command line. Once one error is encountered in a configuration file, no more processing should be done on the remaining configuration files, and the error code corresponding to the first error encountered is the code that should be returned by the whole program. Note that the correct way to signal an error status on exiting is to **return** that code from the `main` function.

Hints

It might be a good idea to compose a UML diagram modelling the problem before you start to write any code. Think about what the logical components of the system are, and what behaviour each component needs to expose to the others.

You may also find the following points useful when designing and debugging your code.

- Think carefully about what happens when a rotor is rotated. In particular, think about how this affects the outputs of that rotor.
- 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 have your program read from and write to files you can redirect the standard input or output. For example:

```
./enigma plugboards/IV.pb reflectors/I.rf rotors/II.rot rotors/III.rot < input.txt > output.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.
- The Enigma machine implements an *invertible* encryption code, thus when you 'encode' your already encoded text, you should get back the original text (as long as you set up the machine in the same way).

Accessing Command Line Arguments

Arguments passed to your program on the command line can be accessed in your code by defining your `main` function to accept two parameters. The first parameter is an integer containing the number of parameters passed. The second parameter is an array of c-strings, containing the values of the command line parameters in the order in which they were specified.

```
int main(int argc, char **argv)
```

Note that the arguments themselves will include the program name in the first argument, and thus the value of the `argc` parameter will be one more than the actual number of command line parameters given.