C++: Simulating the Enigma Machine

Second Year Computing Laboratory
Department of Computing
Imperial College London

Exercise Objectives:

- To demonstrate your understanding of 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.

Summary

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.

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 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, which is why two identically configured machines are needed to securely send and receive a message.

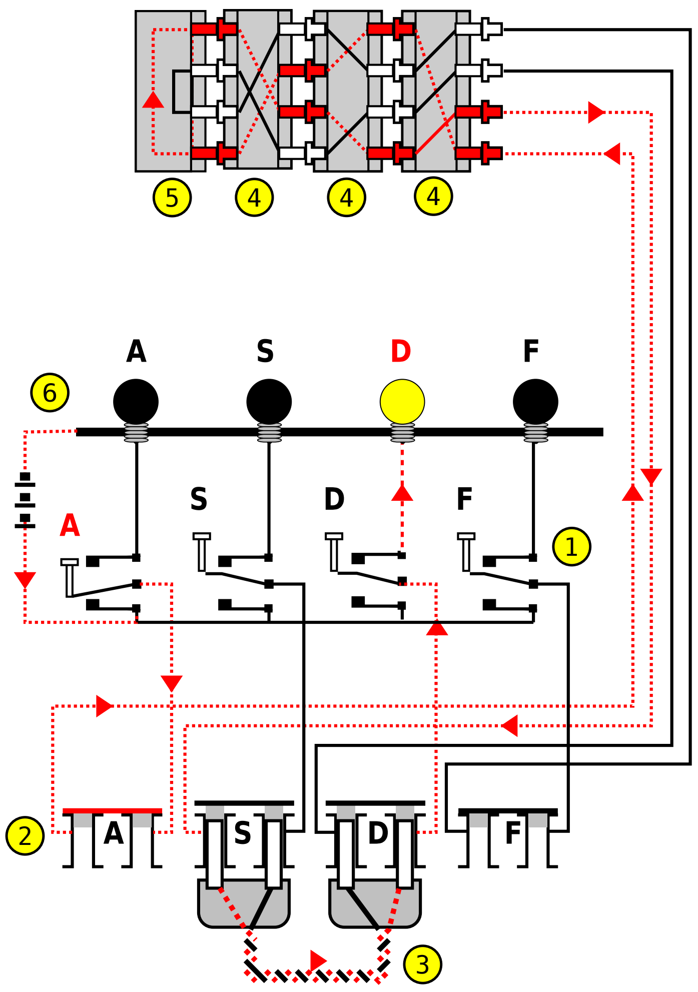

Figure 1: The wiring of an Enigma machine (original version from Wikipedia).

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 rotors, reflector and plugboard in more detail.

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. 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 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 (this controls which letter on each rotor is visible).

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 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).

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 a letter is reflected back to a different letter.

Plugboard

The plugboard consists of a series of contacts that can be connected by plugboard cables to swap the input and output letters. 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 plugboard acts as an identity mapping. For example, in Figure 1 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 machine is translated to:

$$(p \circ f_1^{-1} \circ f_2^{-1} \circ \cdots \circ f_n^{-1} \circ r \circ f_n \circ \cdots \circ f_2 \circ f_1 \circ p)(x)$$

The plugboard, rotor and reflector maps all 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, that is 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. Common words can then be deduced from this and more entries can be added to the inverse function.

Polyalphabetic Encoding

The idea of a *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 the letter frequency analysis (described above) from being an effective decoding technique.

The encoding map is changed in an Enigma machine by rotating the rotors after each key is pressed (as shown in Figure 2). Specifically after each key press the first rotor is rotated by one step (so, for example, an input on 'A' becomes an input on 'B'). Notice that this rotation affects the return path through the rotors, as well as the initial path to the reflector.

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 of all rotors, then after 26 steps of the first rotor the second rotor would also be rotated one step. When the second rotor has completely rotated the third rotor would then be stepped once. Note that the reflector does not rotate.

For a 3 rotor system, unless a message is longer than $17,576$ (26^3) characters, each character would be encoded with a different encoding map.

Submit by 19:00 on Tuesday 27th October 2015

What To Do:

You are to implement a general Enigma machine in C++ in an *Object Oriented* style. Your program will be configured through its command line arguments, and will then encrypt/decrypt messages passed to it on the standard input stream, printing the result on the standard output stream.

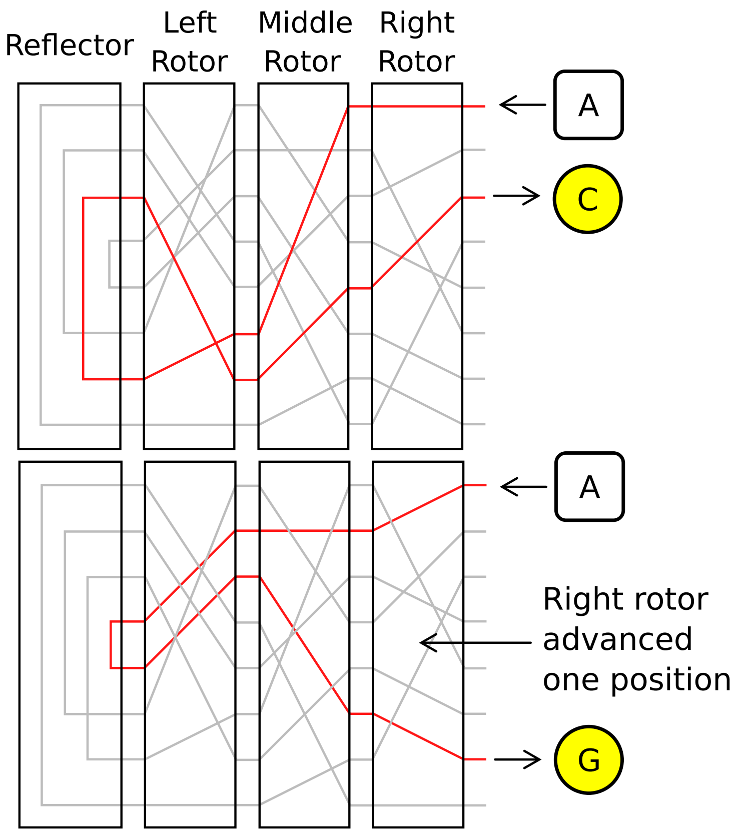

Figure 2: The scrambling action of the Enigma rotors (original version from Wikipedia).

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 these restrictions.

Getting the files required for the exercise

You have each been provided with a Git repository on the department's `GitLab` server that contains the files needed for this exercise. To obtain this skeleton repository you will need to clone it into your local workspace. You can do this with the following command:

```
prompt> git clone https://gitlab.doc.ic.ac.uk/lab1516_autumn/cenigma_<login>.git
```

replacing `<login>` with your normal college login.

You should work on the files in your local workspace, making regular commits back to this Git repository. Your final submission will be taken from the `GitLab` git repository, so make sure that you push your work to it correctly.

The provided files for this exercise are:

`makefile`	a skeleton makefile that you will need to edit
`Main.cpp`	a skeleton C++ program that you will need to edit and extend
`plugboards/<name>.pb`	a number of plugboard configuration files
`rotors/<name>.rot`	a number of rotor configuration files
`tests.hs`	a sample test-suite
`moby.txt.gz`	a large compressed text file, useful for testing

Inputs and Outputs

Your program should be called `enigma` and will be compiled from the command line by calling `make`. You should therefore ensure that you edit the provided makefile in your git repository so that it will build `enigma` from your source code.

Your program will be invoked on the command line (as `enigma`) and will be passed configuration file names as arguments. All but the last configuration file will specify the wiring maps 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 Git repository you have been provided with for this exercise.

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

```
prompt> ./enigma rotors/I.rot rotors/II.rot rotors/II.rot plugboards/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`. The plugboard would use the mapping described in `I.pb`.

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 an error to be thrown. Once the standard input stream is closed, 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. $f_{\texttt{I.rot}}(0) = 1$, so 'A' becomes 'B'), and maps the alphabet down one when mapped backwards ($f^{-1}_{\texttt{I.rot}}(0) = 25$, so 'A' becomes 'Z').

However, *after* each character is encrypted the first rotor is rotated by one position. Inputs that would previously have gone in through the 'A' mapping will instead go in 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 rotated - an output on 'B' is actually an output on 'A' after one rotation. After 26 rotations of a rotor (i.e. when the rotor returns to its original position) the next rotor along in the machine should also be rotated once.

Reflector

The reflector wiring is not configurable. It uses the following function to map an input at alphabet x to its output:

$$r(x) = (x + 13) \; \% \; 26$$

This shifts the letters of the alphabet forward by 13 places, wrapping 'Z' around to 'A'. For example, 'A' is mapped to 'N', 'B' is mapped to 'O', 'M' is mapped to 'Z' and 'N' is mapped to 'A'.

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, 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
```

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.

Hints

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

- 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 behaviours in terms of mapping one input to another, but not all of them perform this operation in the same way. You should consider using inheritance and overriding to model this nicely.
- 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:

  ```
  prompt> ./enigma rotors/II.rot rotors/III.rot plugboards/IV.pb < 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.

Testing

It is possible to incrementally test your Enigma machine, and you should do so thoroughly. A good place to start is to test your program 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, with the same initial settings, should always decrypt it back to the original text again.

The provided Git repository includes several sample configuration files and a small script that locally runs a subset of the labs automated tests. You are encouraged to write your own configuration files and test cases. You can also use `LabTS` to test your work on a subset of the final test cases.

Important: code that fails to compile and run will be awarded **0 marks** for implementation correctness! You should be periodically (but not continuously) testing your code on `LabTS`. If you are experiencing problems with the compilation or execution of your code then please seek help/advice as soon as possible.

Submission

As you work, you should *add*, *commit* and *push* your changes to your Git repository. Your `GitLab` repository should contain the source code and header files for your program.

`LabTS` can be used to test any revision of your work that you wish. However, you will still need to submit a *revision id* to CATe so that we know which version of your code you consider to be your final submission.

Prior to submission, you should check the state of your `GitLab` repository using the `LabTS` webpages: `https://teaching.doc.ic.ac.uk/labts`

If you click through to your `enigma` repository you will see a list of the different versions of your work that you have pushed. Next to each commit you will see a link to that commit on `GitLab` as well as a link to download the `cate_token` (or revision id) corresponding to this commit.

You should download the `cate_token` for the commit that you want to submit as your "final" version. You can of course change this later by resubmitting to CATe as usual.

You should submit your chosen revision ID (`cate_token.txt`) to CATe by 19:00 on Tuesday 27th October 2015.

Important: Make sure you submit the correct `cate_token.txt` file to CATe. It is possible that you may already have a file with this name on your system when you download it. Most browsers will not override the original file, they will instead create a new file called `cate_token(1).txt` or similar. Please take extra care to submit the correct revision id for this exercise.

Assessment

In total there are 30 marks available in this exercise. These are allocated as follows:

Correctness of implementation	15 marks
Appropriate use of C++ standard libraries	5 marks
Appropriate use of object-orientation	5 marks
C++ programming style	5 marks

Any program that does not compile and run will score **0 marks** for implementation correctness. Feedback on the exercise will be returned by Tuesday 10th November 2015.