Part A – Building the rotors.ini file

For our Rotor Machine to work, the rotors need to be established.

What is a Rotor?

The rotor basically establishes a substitution cipher for text. It's a 1D array where the indexes line up with the alphabet, and the contents of the array give an integer.

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	_	.
18	13	2	12	27	9	19	20	22	11	10	14	1	3	24	15	16	23	8	6	0	17	25	7	5	26	21	4

Required Function:

- Prototype: void buildIni(char *)
- Definition: void buildIni(char *filename)

With some simple math we can convert the ASCII value of a character to a matching array index `(int)'A' - 65 = 0`.

To ensure the Rotor Machine works correctly every time, the rotors need to be consistent. We will store the rotor configuration in a file called *rotors.ini.*

This file will be built from another file that will be provided to the Rotor Machine through the -i <config file> syntax.

Rotors.ini will be encrypted through a simple XOR encryption with the value 42.

Pipeline:

- Open the configuration file
- Read an integer value
- XOR it with the value 42
    - value = value ^ 42
- Write encrypted value to rotors.ini

The configuration file should contain 56 integer values 0 – 27 in a shuffled pattern, twice. A sample configuration file will be provided.

Part B – Building the Rotors

In your main() function you'll build two arrays. Since we haven't done arrays in detail yet, I'll be providing some syntax here.

```
int rotor1[28];
int rotor2[28];
```

These two lines will build two integer arrays size 28 (26 letters + space + '.') on the Stack.

These arrays will need to be passed to several functions. Because these arrays "act as their own reference" we can change the contents of them in a function like how we can do that in java.

Required Function:

- Prototype: void buildRotors(int[28], int[28])
- Definition: void buildRotors(int rotor1[28], int rotor2[28])

This function takes the two rotor arrays and populates them from the rotors.ini file.

Read each value from rotors.ini and decrypt them with XOR 42 again.

value = value ^ 42

The first 28 values populate rotor1, the second 28 values populate rotor2.

This function will be called before you encrypt or decrypt any files.

NOTE: if the rotors.ini file doesn't exist, give the error to the user:

```
Error: Rotor machine not initialized.  Run with -i option and provide an
appropriate configuration file.
```

Part C – Setting the Rotors their initial positions

Encryption algorithms usually require a 'key' to encrypt and decrypt data. If your key matches, then you can decrypt a message.

The 'key' in a rotor machine is the initial setting of the rotors. For the sake of our machine, rotor1 will rotate *right* and rotor2 will rotate left.

Required Functions

- Prototypes:
  - void rotateRotorRight(int[28])
  - void rotateRotorLeft(int[28])
- Definitions:
  - void rotateRotorRight(int rotor[28]);
  - void rotateRotorLeft(int rotor[28]);

These two functions will rotate the single dimension arrays.

Rotate Right will move every value down 1 index to the right and move the last index into the first index.

Before:

18	13	2	12	27	9	19	20	22	11	10	14	1	3	24	15	16	23	8	6	0	17	25	7	5	26	21	4

After:

4	18	13	2	12	27	9	19	20	22	11	10	14	1	3	24	15	16	23	8	6	0	17	25	7	5	26	21

Rotate Left will move every value down 1 index to the left and move the first index into the last index.

Before:

18	13	2	12	27	9	19	20	22	11	10	14	1	3	24	15	16	23	8	6	0	17	25	7	5	26	21	4

After:

13	2	12	27	9	19	20	22	11	10	14	1	3	24	15	16	23	8	6	0	17	25	7	5	26	21	4	18

With these two functions in place you can create two functions to set the initial positions of these rotors.

Required Functions:

- Prototypes:
  - void setRotor1(int[28], int);
  - void setRotor2(int[28], int);
- Definitions:
  - void setRotor1(int rotor[28], int rotations);
  - void setRotor2(int rotor[28], int rotations);

These will simply loop the appropriate rotation functions on each rotor. Remember Rotor1 rotates RIGHT and Rotor2 rotates LEFT.

The initial positions will be provided by the `-r <rotations 1> <rotations 2>` part of the usage.

Part D – Do yourself a favor (character conversions)

One way to make all of this easier is to write functions that you can feed values to and get converted values back.

Required Functions:

- Prototypes:
  - int charToIndex(char);
  - char indexToChar(int);
- Definitions
  - int charToIndex(char convert);
  - char indexToChar(int convert);

These functions simply do the math to convert a character into an array index and an array index into a character.

charToIndex

Use the toupper(char) function from #include <ctype.h> to convert all characters to upper case.

The index can be calculated as the ASCII value – 65. You can cast a character to an integer to get it's ASCII value. On the alphabet this will translate to the values 0 – 25.

However, if the character is a SPACE return 26, and if it's a period return 27.

indexToChar

Take the value in and do the reverse calculation. If it's a 26 return a ' ' and if it's a 27 return '.'; otherwise add 65 to the value and cast it to a character to be returned.

Part E – Process your command line arguments

In your main, you have a lot of logic to do. To keep our live easier, we're going to keep the usage to a very fixed set of syntaxes.

```
usage
./exe -i <file>
./exe -e <file1> <file2> -r <r1> <r2>
./exe -d <file1> <file2> -r <r1> <r2>
./exe -e <file1> <file2> -r <r1> <r2> -i <file>
./exe -d <file1> <file2> -r <r1> <r2> -i <file>
```

The user can simply set up the rotors.ini file through ./exe -i <file>

The user can encrypt a file with ./exe -e <in> <out> -r <r1> <r2> using the current rotors.ini configuration

The user can decrypt a file with ./exe -e <in> <out> -r <r1> <r2> using the current rotors.ini configuration

The user can encrypt a file and set up the rotors.ini with the syntax:
./exe -e <file1> <file2> -r <r1> <r2> -i <file>

The user can decrypt a file and set up the rotors.ini with the syntax:
./exe -d <file1> <file2> -r <r1> <r2> -i <file>

Like the BASH assignment ANYTHING outside of these syntax patterns should throw a usage error to the user. Feel free to use the usage above for this error message.

Part F – Encrypt & Decrypt

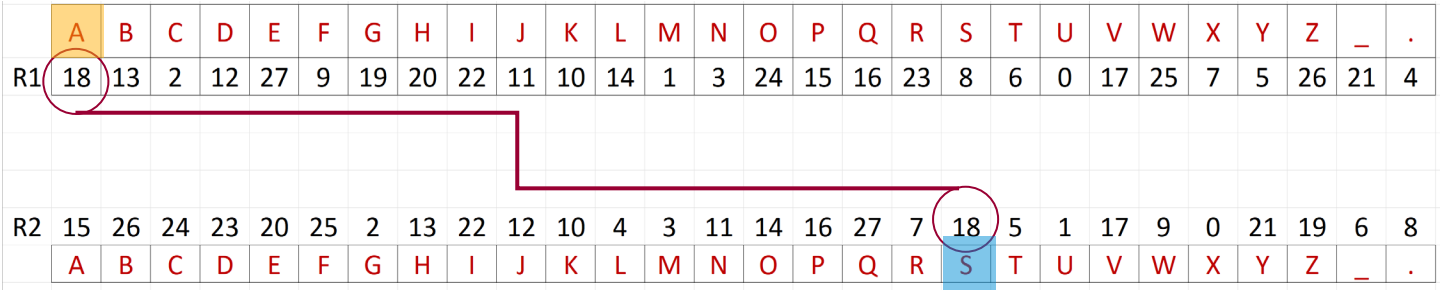
Now for the fun part: Encryption and Decryption.

First let's talk about the machine.

The easiest way to think about a Rotor Machine is that it is a connection puzzle. On either side of the rotors is the alphabet, in between the rotors act effectively as a scrambler.

	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	_	.
R1	18	13	2	12	27	9	19	20	22	11	10	14	1	3	24	15	16	23	8	6	0	17	25	7	5	26	21	4
R2	15	26	24	23	20	25	2	13	22	12	10	4	3	11	14	16	27	7	18	5	1	17	9	0	21	19	6	8
	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	_	.

If I want to encrypt A: I start at Rotor 1 and look up the value for 'A' which would be 18, then I search Rotor 2 for the value 18 and get the matching character, in this case 'S'.



Decrypting goes the other way around. You take the cipher character and start from Rotor 2 and match to Rotor 1. So 'S' turns back into 'A'.

After each character is encrypted or decrypted Rotor 1 & Rotor 2 rotate 1 position (right for Rotor 1 and left for Rotor 2).

If you have the correct rotor configuration and the correct stating positions, you'll be able to encode and decode a file.