

CSE220 Fall 2014 - Homework 7





Due Friday 11/07/2014 @ 6pm

In this assignment you will create a program which implements two classical encryption algorithms. These encryption algorithms will help familiarize you with 1D arrays, 2D arrays, and pointers in C. You will continue to use command line arguments, learn to use preprocessor directives to assist with debugging, and you will also become familiar with using files. You should create functions instead of writing everything in main. We will also provide you with a header file `hw7.h` which contains the array declarations you need for your program. To use `hw7.h` the file will need to be in the same directory as your `.c` file. Also you must not modify this file, you can only submit one file on Sparky which is your `.c` file.

⚠ Array indexing (`A[]`) are not allowed in this assignment. You must use pointer arithmetic instead. All necessary arrays will be declared in the header file. You must use these arrays, not create your own. If you need to reference the size of a buffer you should use the preprocessor `#define` value that is used in the header file. Do **NOT hardcode the value in your program. We **WILL** test for this.**

i You can refer to chapters 7.5 through 7.7 for handling file input and output.

Getting Started

-  Create a new c program file `hw7.c`
-  Download `hw7.h` from piazza and review the defined arrays.
 - The arrays in this file are the **only arrays** you are allowed to use in your program.
-  At the top of your program in comments put your name and id.
-  Add the imports for `hw7.h` and `stdlib.h`

```
/*  
Homework #7  
name: MY_NAME
```

```
sbuid: MY_SBU_ID
*/
#include <stdlib.h>
#include "hw7.h"
```

Program operation

You will create a command line tool which will encrypt text using these classical encryption techniques. If the `-s` is provided your program should perform a substitution cipher. If the `-a` flag is provided your program should perform the Autokey cipher. None of these flags are allowed to be used in combination with each other. If they are your program should do nothing and return `EXIT_FAILURE`. On successful execution your program should return `EXIT_SUCCESS`. Some of these operations will also need other command line arguments which will be described below. Finally there is the `-h` option which will display the usage for this program and exit with an `EXIT_SUCCESS` return code. If no flags are provided to the program the help menu should be displayed and the program should return `EXIT_FAILURE`.

```
usage: ./a.out [-s | -a | -h]
    -s          Substitution cipher
                Additional parameters: [-e | -d] n INPUT_FILE OUTPUT_FILE
                -e          Encrypt using the substitution cipher.
                -d          Decrypt using the substitution cipher.
                n           The amount of position to shift by.
                INPUT_FILE  This can be any file on the file system or -
                           which specifies stdin.
                OUTPUT_FILE This can be any file on the system or -
                           which specifies stdout.

    -a          Autokey cipher
                Additional parameters: [-e | -d] n INPUT_FILE KEY_FILE
                OUTPUT_FILE
                -e          Encrypt using the autokey cipher
                -d          Decrypt using the autokey cipher
                n           The initial shift value
                INPUT_FILE  This can be any file on the file system or -
                           which specifies stdin.
                KEY_FILE    This can be any file on the file system or -
                           which specifies stdin.
                OUTPUT_FILE This can be any file on the system or -
                           which specifies stdout.
```

-h Display this help menu.

- ❗ Remember `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in `<stdlib.h>`
- ❗ Again we suggest that you should create functions for each of the operations defined here in this document.

Part 1 - Another type of debugging

So far you have learned a bit about using GCC and GDB to help debug and fix your programs. While these are very useful tools, sometimes the easiest way to see what is happening in your program is to just use print statements. We shouldn't just put `printf` all over our program though. We might not always want to see these print outs (way too much information for normal operation). One possible solution to this is passing a command line argument that turns debugging on and off. This might be an acceptable solution but it will clutter our code with lots of if statements to check if debugging is enabled or not, make our binary larger when we don't want debugging enabled, etc. Instead we will use some preprocessor tricks to give us some logging statements when we **compile with** the with the flag `-DDEBUG`. When we **compile without** the flag `-DDEBUG`, none of these debugging statements will be printed.

Logging with the preprocessor

Somewhere at the top of your source file put the following line of code defining out debug logging macro

```
#define debug(msg) printf("DEBUG: %s", msg)
```

Then in your program use the `debug` macro

```
#include <stdlib.h>
#include <stdio.h>

#define debug(msg) printf("DEBUG: %s", msg)
```

```
int main(int argc, char *argv[]) {  
    debug("Hello, World!\n");  
    return EXIT_SUCCESS;  
}
```

Then compile your program and run it.

```
$ gcc debug.c  
$ ./a.out  
DEBUG: Hello, World!
```

Great! You just created your first [preprocessor macro](#). Unfortunately this is no better than just adding a print statement. Lets fix that!

The preprocessor has *if*, *elif*, and *else* [directives](#) that we can use to control what gets added during compilation. Lets create an *if* directive that will include a section of code if `DEBUG` is defined within the preprocessor.

```
#include <stdlib.h>  
#include <stdio.h>  
  
#define debug(msg) printf("DEBUG: %s", msg)  
  
int main(int argc, char *argv[]) {  
    #ifdef DEBUG  
        debug("Debug flag was defined\n");  
    #endif  
    printf("Hello, World!\n");  
    return EXIT_SUCCESS;  
}
```

When we compile this program it will check to see if `#define DEBUG` was defined in our program. Lets test this out.

```
$ gcc debug.c  
$ ./a.out  
Hello, World!
```

Cool the debug message didn't print out. Now lets define `DEBUG` during the compilation process, and run the program again.

```
$ gcc -DDEBUG debug.c  
$ ./a.out
```

```
DEBUG: Debug flag was defined
Hello, World!
```

Here you can see that debug was defined so that extra code between `#ifdef DEBUG` and `#endif` was included. This technique will work for certain situations, but if we have a lot of logging messages in our program this will quickly clutter our code and make it unreadable. Fortunately we can do better.

Instead of doing `#ifdef DEBUG` all over our program we can instead do `#ifdef DEBUG` around our `#define debug` macro.

```
#include <stdlib.h>
#include <stdio.h>

#ifdef DEBUG
    #define debug(msg) printf("DEBUG: %s", msg)
#endif

int main(int argc, char *argv[]) {
    debug("Debug flag was defined\n");
    printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

There is an issue with this, but lets try to compile the program.

```
$ gcc -DDEBUG debug.c
$ ./a.out
DEBUG: Debug flag was defined
Hello, World!
```

Cool it works. Now lets try to compile it without defining `-DDEBUG`.

```
$ gcc debug.c
/tmp/cc6F04VW.o: In function `main':
debug.c:(.text+0x1a): undefined reference to `debug'
collect2: error: ld returned 1 exit status
```

Whoops. What happened here? Well when we used `-DDEBUG` the `debug` macro was defined, so it worked as expected. When we don't compile with `-DDEBUG` the `#define debug` is never declared in our file so it is never substituted in our program. Since we used `debug` in the middle of our code the preprocessor and compiler have no idea what `debug` symbol is so it fails. Luckily this is easy to fix. We simply have to add another case to our preprocessor *if, else* statement to handle this case.

```
#include <stdlib.h>
```

```
#include <stdio.h>

#ifdef DEBUG
    #define debug(msg) printf("DEBUG: %s", msg)
#else
    #define debug(msg)
#endif

int main(int argc, char *argv[]) {
    debug("Debug flag was defined\n");
    printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

Here we tell the preprocessor to replace any occurrences of `debug(msg)` with nothing, so now when we don't compile with `-DDEBUG`. The preprocessor simply replaces `debug("Debug flag was defined\n")` with an empty space. Lets compile again.

```
$ gcc debug.c
$ ./a.out
Hello, World!
```

Cool. Now we can embed `debug` macros all over our program that look like normal functions. There's still a few more cool tricks we can do to make this better.

The preprocessor has a few special macros defined called `__LINE__`, `__FILE__`, and `__FUNCTION__`. These macros will be replaced by the preprocessor to evaluate to the *line number* where the macro is called, the *file name* that the macro is called in, and the *function name* that the macro is called in. Lets play with this a bit.

```
#include <stdlib.h>
#include <stdio.h>

#ifdef DEBUG
    #define debug(msg) printf("DEBUG: %s:%s:%d %s", __FILE__,
__FUNCTION__, __LINE__, msg)
#else
    #define debug(msg)
#endif

int main(int argc, char *argv[]) {
    debug("Debug flag was defined\n");
    printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

Lets compile this program and run.

```
gcc -DDEBUG debug.c
$ ./a.out
DEBUG: debug.c:main:11 Debug flag was defined
Hello, World!
```

As you can see all the `__FILE__`, `__FUNCTION__`, and `__LINE__` were replaced with the corresponding values for when debug was called in the program. Pretty cool, but we can still do even better! Normally when we want to print something we use `printf()` and use the format specifiers and variable arguments to print useful information. With our current setup though we can't do that. Fortunately for us the preprocessor offers up a `__VA_ARGS__` macro which we can use to accomplish this.

i I want to point out that the syntax for this gets a bit crazy and hard to understand (complex preprocessor stuff is a bit of a black art). I'll try my best to describe it but you may need to do some more googling if the below explanation is not sufficient.

```
#include <stdlib.h>
#include <stdio.h>

#ifdef DEBUG
    #define debug(fmt, ...) printf("DEBUG: %s:%s:%d " fmt, __FILE__,
__FUNCTION__, __LINE__, ##__VA_ARGS__)
#else
    #define debug(fmt, ...)
#endif

int main(int argc, char *argv[]) {
    debug("Program has %d args\n", argc);
    printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

First lets compile and run the program and see the results.

```
$ gcc -DDEBUG debug.c
$ ./a.out
DEBUG: debug.c:main:11 Program has 1 args
Hello, World!
$ gcc debug.c
$ ./a.out
```

```
Hello, World!
```

The macro works as expected, but let's try to explain it a bit.

First we changed the definition of the macro to be `#define debug(fmt, ...)`. The first argument `fmt` is the format string that we normally define for `printf` and `...` is the way to declare a macro that accepts a variable number of arguments.

Next we have `"DEBUG: %s:%s:%d " fmt`. This was very confusing to me at first, but the C preprocessor can concatenate string literals that are next to each other. So if `fmt` was the string `"crazy %d concatenation"` then this statement evaluates to `"DEBUG: %s:%s:%d crazy %d concatenation"`. Then we have our predefined preprocessor macros that are used for the string `"DEBUG: %s:%s:%d "`, and then we reach this next confusing statement: `, ##__VA_ARGS__`. The macro `__VA_ARGS__` will expand into the variable arguments provided to the debug statement, but then we have this crazy `, ##`. This is a hack for allowing no arguments to be passed to the debug macro, Ex. `debug("I have no varargs")`. If we didn't do this, the previous debug statement would throw a warning/error during the compilation process as it would expect a `__VA_ARGS__` value.

This is one of the many interesting things we can use the C preprocessor for. I'll leave you off with a final example of another interesting example that you might see in real code. Have fun and experiment!

```
#include <stdlib.h>
#include <stdio.h>

#ifdef DEBUG
    #define debug(fmt, ...) do{printf("DEBUG: %s:%s:%d " fmt, __FILE__,
__FUNCTION__, __LINE__, ##__VA_ARGS__);}while(0)
    #define info(fmt, ...) do{printf("INFO: %s:%s:%d " fmt, __FILE__,
__FUNCTION__, __LINE__, ##__VA_ARGS__);}while(0)
#else
    #define debug(fmt, ...)
    #define info(fmt, ...) do{printf("INFO: " fmt,
##__VA_ARGS__);}while(0)
#endif

int main(int argc, char *argv[]) {
    debug("Program has %d args\n", argc);
    printf("Hello, World!\n");
    info("Info statement. Should print out always, with varying amounts of
information depending on flags provided.\n");
    return EXIT_SUCCESS;
}
```

```
$ gcc debug.c
```



```
$ ./a.out
Hello, World!
INFO: Info statement. Should print out always, with varying amounts of
information depending on flags provided.
$ gcc -DEBUG debug.c
$ ./a.out
DEBUG: debug.c:main:13 Program has 1 args
Hello, World!
INFO: debug.c:main:15 Info statement. Should print out always, with varying
amounts of information depending on flags provided.
```

i Some programmers like to wrap the code in macros with a `do{ /* some code here */ }while(false)` loop. They do this because if your macro is made up of multiple statements, it will force you to add `;` to all the statements in the do while loop. Then you still have to terminate this macro with a `;` when you use it which makes it seem like a normal function in your C code.

tl;dr; It is a way to prevent yourself from making stupid mistakes with macros.

i Normally you would declare these macros in your programs header file instead of directly in your `.c` file. Unfortunately because we cannot submit multiple files on Sparky we need to compromise and you **must** declare them directly in our C source file.

In your program you should define an `#ifdef CSE220` directive where you will declare macros for printing certain debug information for grading. We **WILL** check for this output. Keep all other debugging macros under a different `#ifdef` to avoid cluttering the grading output.

```
#ifdef CSE220
    /* Define assignment macros here */
#else
    /* Define assignment macros here */
#endif
```

Part 2 - Substitution Cipher

You will implement a generic substitution cipher. One of the most popular of these substitution ciphers is the [Caesar Cipher](#). Another popular variation of the substitution cipher is [ROT13](#) which is sometimes used on forums to hide spoilers.

In a substitution cipher, to encrypt our text we take our alphabet and shift it to the left by `n % 26` positions. To decrypt the text we shift the alphabet to the right by `n % 26` positions. Lets show an example using the Caesar cipher.

```
n = 3          // Shift amount

Original Alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ      // Original Alphabet
Shifted Alphabet:  DEFGHIJKLMNOPQRSTUVWXYZABC      // Alphabet shifted to
left by 3

Plaintext:  CSE220 IS MY FAVORITE CLASS
Ciphertext: FVH220 LV PB IDYRULWH FODVV
```

For this operation, space within the plaintext should be preserved. Also, numbers and special characters should be preserved. Any lowercase text should be converted to uppercase. When your program is called you need to pass the `-s` flag to invoke this cipher option. Also you will need to provide the `-e` or `-d` flags but not both. If both `-e` and `-d` are both provided your program should do nothing and return with the value `EXIT_FAILURE`.

The `-e` flag specifies that you want to encrypt and the `-d` flag specifies that you want to decrypt. Finally there are 3 positional arguments which need to be provided in order after the flags. The first is the value `n` which determines how many positions to shift your alphabet by. The second is the input file to encrypt or decrypt. The third is the output file to place the results in.

Substitution cipher example commands

```
$ ./a.out -s [-e | -d] n INPUT_FILE OUTPUT_FILE
```

If the input file argument is provided as the special value `-` then the program expects input to come from `stdin`. If the output file is provided as the special value `-` then the program will output to `stdout`. If any of the positional arguments are missing your program should exit with `EXIT_FAILURE`.

If your program is compiled with `-DCSE220`, during the operation of substitution cipher, your program should print out `shift amount: n` where `n` is the value passed on the command line, `input file: INPUT_FILE` where `INPUT_FILE` is the value passed on the command line (If you provide `-` you should print out `STD_IN`), `output file: OUTPUT_FILE` where `OUTPUT_FILE` is the value passed on the command line (if you provide `-` you should print out `STD_OUT`), `cipher type: substitution cipher`, and the `cipher operation: OPERATION` where `OPERATION` is encryption or decryption.

Encryption

Input File: plain.txt

```
$ cat plain.txt
CSE220 IS MY FAVORITE CLASS
```

Sample output:

```
$ ./a.out -s -e 3 plain.txt -
FVH220 LV PB IDYRULWH FODVV
$ echo $?
0
```

```
$ ./a.out -s -e 7 plain.txt -
JZL220 PZ TF MHCVPAL JSHZZ
$ echo $?
0
```

```
$ ls
a.out plain.txt
$ ./a.out -s -e 7 plain.txt cipher.txt
$ echo $?
0
$ ls
a.out cipher.txt plain.txt
$ cat cipher.txt
JZL220 PZ TF MHCVPAL JSHZZ
```

Decryption

Input File: cipher.txt

```
$ cat cipher.txt
JZL220 PZ TF MHCVPAL JSHZZ
```

Sample output:

```
$ ./a.out -sd 7 cipher.txt -
CSE220 IS MY FAVORITE CLASS
$ echo $?
0
```

Debug output

Input File: cipher.txt

```
$ cat cipher.txt
JZL220 PZ TF MHCVPAL JSHZZ
```

Sample output:

```
$ gcc -DCSE220 -Wall -Werror hw7.c
$ ./a.out -s -d 7 cipher.txt -
CSE220: shift amount: 7
CSE220: input file: cipher.txt
CSE220: output file: STD_OUT
CSE220: cipher type: substitution cipher
CSE220: cipher operation: decryption
CSE220 IS MY FAVORITE CLASS
$ echo $?
0
```

i `$?` is an environment variable in bash which holds the error code. Most *nix systems, cygwin, and OSX default to bash. Unfortunately on Sparky the shell defaults to `csch` (you can switch by simply typing the command `bash`). You can see the return code in `csch` by using `echo $status`. If you are using the regular windows command prompt (not cygwin) you can view the return code by using `echo %errorlevel%`. If you are using another shell, its up to you to figure out where this return code is stored if you want it.

i Its ok for these debug printouts to vary slightly, as long as the base information as described above is printed out.

Part 3 - Autokey Cipher

The [Autokey cipher](#) is a much more complex version of the substitution cipher in part 2. The version we create will use a [Tabula Recta](#) to encrypt and decrypt the text provided.

A tabula recta is classically a 26x26 2D array of letters where each row is a different substitution cipher shift. For example:

	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
A	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
B	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	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	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

Lets say we have the plaintext: I have the answers

Next we need to a key which is a special word that we will use to encrypt our plaintext message.

We will use the key TWOTWENTY to encrypt and decrypt our message in this example.

i Note that the key TWOTWENTY is shorter than the plaintext message I have the answers . In this case we must pad the end of the key so that the new key and the plaintext are of the same length (only counting alphabetic characters).

```

Plaintext: I HAVE THE ANSWERS
Key:      T WOTW ENT Y
Pad:      IHAVET
New Key:  T WOTW ENT YIHAVET

```

We create the ciphertext by using the key we picked out and then we fill in the rest of the positions with the plain text message until our key is as long as our plaintext. You should use the beginning of the plaintext to pad as necessary.

To encrypt using our tabula recta we find the letter I in the first row, the letter T in the first column. The intersection between the column I and the row T results in the encrypted character (This case

B). We keep doing this operation (H first row, W first column...) until we have encrypted the entire message.

```
Plaintext:   I HAVE THE ANSWERS
Key:         T WOTW ENT YIHAVET
Ciphertext:  B DOOA XUX YVZWZVL
```

To decrypt the message we do the opposite:

```
Ciphertext:  B DOOA XUX YVZWZVL
Key:         T WOTW ENT Y
```

We look for the letter T in the first column. We look for the letter B in the row that starts with T . The unencrypted letter is the letter in the first row which is the same column as the letter B (Which should be I). Repeat this process to decrypt up to the entire key.

```
Ciphertext:  B DOOA XUX UVZWZVL
Key:         T WOTW ENT Y
Plaintext:   I HAVE THE A
```

We still have a few letters left that are not decrypted. If you remember we padded the original key with the original message. We now have to use the letters that we already unencrypted to decrypt the rest of the message.

```
Ciphertext:  B DOOA XUX YVZWZVL
Key:         T WOTW ENT YI
Plaintext:   I HAVE THE A
```

So we would use the first decrypted letter I and pair with V to get N . Then use H and Z to get S . Keep doing this until we decrypt the entire message.

```
Ciphertext:  B DOOA XUX YVZWZVL
Key:         T WOTW ENT YIHAVET
Plaintext:   I HAVE THE ANSWERS
```

You should skip any whitespace and special characters in the strings. You should make all lowercase letters uppercase.

To use this cipher option you will pass the -a flag to invoke this operation. Also you will need to provide the -e or -d flags but not both. If -e and -d are both provided your program should do nothing and return with the value EXIT_FAILURE .

The -e flag specifies that you want to encrypt and the -d flag specifies that you want to decrypt. Finally there are 4 positional arguments.

The first is the value `n` which determines how many positions to shift your alphabet by. This value `n` should be bound between `[0-25]`, if the inputted `n` is greater than 25 you must use `n % 26`. The first row of the tabula recta will be shifted by `n % 26`. The second row will be shifted by `(n+1) % 26`, the third row will be shifted by `(n+2) % 26`, etc. Until you have created 26 shifted rows.

The second is the input file to encrypt or decrypt. The third is the key used to perform the encryption with. The fourth is the output file to place the results in.

Autokey Cipher

```
$ ./a.out -a [-e | -d] n INPUT_FILE KEY_FILE OUTPUT_FILE
```

If the input file is provided as the special value `-` then the program expects input to come from `stdin`. If the key is provided as the special value `-` then the program expects the key to come from `stdin`. If the output file is provided as the special value `-` then the program will output to `stdout`. If any of the positional arguments are missing your program should exit with `EXIT_FAILURE`.

If your program is compiled with `-DCSE220`, during the operation of autokey cipher, your program should print out `initial shift amount: n` where `n` is the value passed on the command line, `input file: INPUT_FILE` where `INPUT_FILE` is the value passed on the command line (if you provide `-` you should print out `STD_IN`), `output file: OUTPUT_FILE` where `OUTPUT_FILE` is the value passed on the command line (if you provide `-` you should print out `STD_OUT`), `cipher type: autokey cipher`, `cipher operation: OPERATION` where `OPERATION` is encryption or decryption, and the tabula recta matrix.

Encryption

Input File: plain.txt

```
$ cat plain.txt
I HAVE THE ANSWERS
$ cat key.txt
TWOTWENTY
```

Sample output:

```
$ ./a.out -a -e 0 plain.txt key.txt -
B D00A XUX YVZWZVL
$ echo $?
0
```

Decryption

Input File: cipher.txt

```
$ cat cipher.txt
B D00A XUX YVZWZVL
```

Sample output:

```
$ ./a.out -a -d 0 cipher.txt key.txt -
I HAVE THE ANSWERS
$ echo $?
0
```

Debug output

Input File: cipher.txt

```
$ cat cipher.txt
X ZKKW TQT URVSVRH
$ cat key.txt
TWOTWENTY
```

Sample output:

```
$ gcc -DCSE220 -Wall -Werror hw7.c
$ ./a.out -a -d 4 cipher.txt key.txt -
CSE220: initial shift amount: 0
CSE220: input file: cipher.txt
CSE220: output file: STD_OUT
CSE220: cipher type: autokey cipher
CSE220: cipher operation: decryption
CSE220: Tabula Recta

E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
I J K L M N O P Q R S T U V W X Y Z A B C D E F G H
J K L M N O P Q R S T U V W X Y Z A B C D E F G H I
K L M N O P Q R S T U V W X Y Z A B C D E F G H I J
L M N O P Q R S T U V W X Y Z A B C D E F G H I J K
M N O P Q R S T U V W X Y Z A B C D E F G H I J K L
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
O P Q R S T U V W X Y Z A B C D E F G H I J K L M N
```



```
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Q R S T U V W X Y Z A B C D E F G H I J K L M N O P
R S T U V W X Y Z A B C D E F G H I J K L M N O P Q
S T U V W X Y Z A B C D E F G H I J K L M N O P Q R
T U V W X Y Z A B C D E F G H I J K L M N O P Q R S
U V W X Y Z A B C D E F G H I J K L M N O P Q R S T
V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
W X Y Z A B C D E F G H I J K L M N O P Q R S T U V
X Y Z A B C D E F G H I J K L M N O P Q R S T U V W
Y Z A B C D E F G H I J K L M N O P Q R S T U V W X
Z 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
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
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 A
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
```

```
I HAVE THE ANSWERS
$ echo $?
0
```

Hand-in instructions

Before submitting your assignment make sure your program will compile with no warnings and errors using the `-Wall` and `-Werror` flags.

See Sparky Submission Instructions on [piazza](#) for hand-in instructions.

i When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly. If your program doesn't work correctly the grader may be able to give you partial points, assuming he can read the code that you have written.