README.md 34.4 KB

Homework 1 - CSE 320 - Fall 2017

Professor Jennifer Wong-Ma & Professor Eugene Stark

Due Date: Wednesday 09/13/2017 @ 11:59pm

Read the entire doc before you start

Introduction

In this assignment, you will write a command line utility to encrypt and decrypt strings and files. The first encryption scheme is a Polybius cipher and the second is a Fractionated Morse cipher. The goal of this homework is to familiarize yourself with C programming, with a focus on input/output, strings in C, and the use of pointers.

You **MUST** write your helper functions in a file separate from `main.c`. The `main.c` file **MUST ONLY** contain `#include`s, local `#define`s and the `main` function. This is the only requirement for project structure. Beyond this, you may have as many or as few additional `.c` files in the `src` directory as you wish. Also, you may declare as many or as few headers as you wish. In this document, we use `hw1.c` as our example file containing helper functions.

> 😱 Array indexing (**'A[]'**) is not allowed in this assignment. You **MUST USE** pointer arithmetic instead. All necessary arrays are declared in the `const.h` header file. You **MUST USE** these arrays. **DO NOT** create your own arrays. We **WILL** check for this.

Getting Started

Fetch base code for `hw1` as described in `hw0`. You can find it at this link: https://gitlab02.cs.stonybrook.edu/cse320/hw1 (https://gitlab02.cs.stonybrook.edu/cse320/hw1).

Both repos will have a file named `.gitlab-ci.yml` with different contents. Simply merging these files will cause a merge conflict. To avoid this, we will merge the repos using a flag so that the `.gitlab-ci.yml` found in the `hw1` repo will be the file that is preserved.

To merge, use this command:

```
git merge -m "Merging HW1_CODE" HW1_CODE/master --strategy-option theirs
```

Here is the structure of the base code:

```
hw1
├── include
│   ├── const.h
│   ├── debug.h
│   └── hw1.h
├── Makefile
├── rsrc
│   ├── ans.txt
│   ├── camelcase.txt
│   ├── plain.txt
│   └── stonybrook.txt
├── src
│   ├── const.c
│   ├── hw1.c
│   └── main.c
└── tests
    └── hw1_tests.c
```

> 🤓 Reference for pointers: http://beej.us/guide/bgc/output/html/multipage/pointers.html (http://beej.us/guide/bgc/output/html/multipage/pointers.html).
>
> 🤓 Reference for command line arguments: http://beej.us/guide/bgc/output/html/multipage/morestuff.html#clargs (http://beej.us/guide/bgc/output/html/multipage/morestuff.html#clargs)

NOTE: All commands from here on are assumed to be run from the `hw1` directory.

NOTE ABOUT PROGRAM OUTPUT

Print statements are VERY important and we will grade this assignment using them. In the UNIX world stringing together programs with piping and scripting is commonplace. Although combining programs in this way is extremely powerful, it means that each program must not print extraneous output. For example, you would expect `ls` to output a list of files in a directory and nothing else. Similarly, your program must follow the specifications for normal operation.

Use the debug macro `DEBUG` (described in the 320 reference document in the Piazza resources section) for any other program output or messages you many need while coding (e.g. debugging output).

Part 1: Program Operation and Argument Validation

In this part, you will write a function to validate the arguments passed to your program via the command line. Your program will support the following flags:

- If the `-h` flag is provided, you will display the usage for the program and exit with an `EXIT_SUCCESS` return code
- If no flags are provided, you will display the usage and return with an `EXIT_FAILURE` return code
- If the `-p` flag is provided, you will perform Polybius encryption/decryption (Part 2)
- If the `-f` flag is provided, you will perform Fractionated Morse encryption/decryption (Part 3)

> The `-p` and `-f` flags are not allowed to be used in combination with each other
>
> 🤓 `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>` which represent success and failure return codes respectively.

Some of these operations will also need other command line arguments which are described in each part of the assignment. The two usages for this program are:

```
usage: ./hw1 -h [any other number or type of arguments]
usage: ./hw1 [-h] -p|-f -e|-d [-k KEY] [-r ROWS] [-c COLUMNS]
    -h       Display this help menu.
    -p       Polybius Cipher
                 -e (Positional) - Encrypt using the Polybius cipher
                 -d (Positional) - Decrypt using the Polybius cipher
                 -k (Optional) - KEY is the key to be used in the cipher.
                                 It must have no repeated characters and each character in th
                 -r (Optional) - ROWS is the number of rows for the Polybius cipher table.
                                 Must be between 9 and 15, inclusive. Defaults to 10.
                 -c (Optional) - COLUMNS is the number of columns for the Polybius cipher tab
                                 Must be between 9 and 15, inclusive. Defaults to 10.

    -f       Fractionated Morse Cipher
                 -e (Positional) - Encrypt using the Fractionated Morse cipher
                 -d (Positional) - Decrypt using the Fractionated Morse cipher
                 -k (Optional) - KEY is the key to be used in the cipher.
                                 It must have no repeated characters and each character in th
```

> 🤓 `stdin`, `stdout`, and `stderr` are special files that are opened upon execution for all programs and do not need to be reopened.

When testing your program, we guarantee the following:

- All positional arguments (`-p|-f, -e|-d`) will come before any optional arguments (`-k, -r, and -c`). The optional arguments can come in any order after the positional ones.

- If an optional flag is given, the corresponding argument will be provided, i.e. you do not have to worry about a missing argument after an optional flag (e.g. `-r -k` cannot occur)
- If the `-h` flag is provided, it will be the first positional argument after the program executable.
- If `-r` or `-c` are given, the ROWS and COLUMNS arguments must be parsed as positive integers
- If `-k` is given, the corresponding argument will be a single word (i.e. will have no whitespace).

> 😱 You may only use `argc` and `argv` for argument parsing and validation. Using any libraries that parse command line arguments (e.g. `getopt`) is prohibited.
>
> 😱 Any libraries that help you parse strings are prohibited as well (`string.h` , `ctype.h` , etc). *This is intentional and will help you practice parsing strings and manipulate pointers.*
>
> 😱 You **CAN NOT** use dynamic memory allocation in this assignment (i.e. `malloc` , `realloc` , `calloc` , `mmap` , etc)

For example, the following are a subset of the possible valid argument combinations:

- `$ bin/hw1 -h ...`
- `$ bin/hw1 -p -e  -k KEY -c COLUMNS`
- `$ bin/hw1 -p -d -k KEY`
- `$ bin/hw1 -p -e -r ROWS -c COLUMNS`
- `$ bin/hw1 -f -e`

Some examples of invalid orderings would be:

- `$ bin/hw1 -e -p`
- `$ bin/hw1 -d -f -r ROWS`

NOTE: Do not explicitly code to handle these invalid cases. Rather, while parsing `argv` , immediately return 0 if you encounter an unexpected argument (e.g. neither `-p` or `-f` in `argv[1]`)

NOTE: The makefile compiles the `hw1` executable into the `bin` folder. Assume all commands in this doc are run from from the `hw1` directory of your repo.

> 😱 The `...` means that all arguments, if any, are to be ignored, i.e. the usage `bin/hw1 -h -p -e -r ROWS -k KEY` is equivalent to `bin/hw1 -h`

Required Validate Arguments Function

In `const.h` , you will find the following function prototype (function declaration) already declared for you. You **MUST** implement this function in `hw1.c` as part of the assignment.

```
/*
 * Validates the program arguments.
 * @return An unsigned short representing the mode of execution, or 0 if there was an error.
 */
unsigned short validargs(int argc, char** argv);
```

> 😱 This function must be implemented as specified as it will be tested and graded independently. It should always return. The USAGE macro should never be called from validargs.

The `validargs` function validates all arguments passed to it and returns an unsigned short (2 bytes on our VM) that contains the information necessary for the proper execution of the program.

The unsigned short returned represents the `mode of operation` (*mode*) of the program.

The *mode* is 0 if there is any form of failure. This includes, but is not limited to:

- Invalid number of arguments (too few or too many)
- Invalid ordering of arguments
- Incorrect arguments for the specified cipher (e.g. `-r` or `-c` being passed in with `-f`)
- Invalid key (if one is specified). A key is invalid if it contains characters not in the alphabet or has repeated characters. **Note:** The ciphers use different alphabets.
- The number of rows or columns is invalid (i.e. more than 16 or less than 9)
- `(rows * columns) < length of polybius_alphabet` for the Polybius cipher

For valid arguments, `validargs` sets the *mode* as follows:

- If the `-h` flag is specified, the most significant bit is 1
- The second most significant bit is 0 if `-p` is passed (i.e. user wants the Polybius cipher) and 1 if `-f` is passed (i.e. user wants the Fractionated Morse cipher)
- The third most significant bit is 0 if `-e` is passed (i.e. user wants to encrypt) and 1 if `-d` is passed (i.e. user wants to decrypt)
- If the polybius cipher is used: the 8 least significant bits will be split into 2 groups of 4. The upper 4 bits will encode the number of rows for the table and the lower 4 will encode the number of columns. If the Fractionated Morse cipher is used, the values of the lower 8 bits do not matter.

If -p is given but no (-r) ROWS or (-c) COLUMNS are specified this function **MUST** set the lower bits to the default value of 10. If one or the other (rows/columns) is specified then you **MUST** keep that value rather than assigning the default.

Here is a visual representation of the *mode*:

```
┌──┐┌──┐┌──┐┌──┐┌──┐┌──┐┌─┐┌─┐┌─┐┌─┐┌─┐┌─┐┌─┐┌─┐┌─┐┌─┐
 15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
└──┘└──┘└──┘└──┘└──┘└──┘└─┘└─┘└─┘└─┘└─┘└─┘└─┘└─┘└─┘└─┘
 |   |   |   |                |  |        |  |        |
 |   |   |   |                |  |        |  |        |
 |   |   |   |                |  |        |  |        |
 |   |   |   └────────────────*  └────────*  └────────*
 |   |   |
  |   |   |       don't care     rows (-p)    columns(-p)
  |   |   |
  |   |   └──────────────────────────── 1 (-d), 0 (-e)
  |   |
  |   └──────────────────────────────── 1 (-f), 0 (-p)
  |
  └──────────────────────────────────── 1 (-h), 0 otherwise
```

If `validargs` returns 0 indicating failure, your program must print `USAGE(program_name, return_code)` and return `EXIT_FAILURE`.

If `validargs` sets the most significant bit of the return code to 1 (i.e. the `-h` flag was passed), your program must print `USAGE(program_name,
return_code)` and return `EXIT_SUCCESS`.

> 🤓 The `USAGE(program_name, return_code)` macro is already defined for you in `const.h`.

If the *mode* is non-zero, your program must run a Polybius or Fractionated Morse cipher accordingly and return `EXIT_SUCCESS` upon successful completion.

If `-k` is provided, you must check to confirm that the specified key is valid.

- For the Polybius cipher, the key must have a non-repeating subset of the characters in the `polybius_alphabet` variable defined in `const.c`.
- For the Fractionated Morse Cipher, the key must have a non-repeating subset of the characters in the `fm_alphabet` variable defined in `const.c`.

If the key is not valid, this is an error case. If it is valid, you must assign it to the `key` global variable declared in `const.h`

> 🤓 Remember `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in `<stdlib.h>`. Also note, EXIT_SUCCESS is 0 and EXIT_FAILURE is 1.
>
> 🤓 We suggest that you create functions for each of the operations defined in this document. Writing modular code will help you isolate and fix problems.

Sample validargs Execution

The following are examples of `validargs` return values for given inputs. Each input is a bash command that can be used to run the program. In the examples, all don't care bits (bits 8-12 in the diagram above) have been set to 0. For Fractionated Morse cipher examples, the lower 8 bits have also been set to 0.

- Input: `bin/hw1 -h`. Return value: 0x8000 (`-h` bit is set. All other bits are don't cares.)
- Input: `bin/hw1 -f -d`. Return value: 0x6000 (`-f` and `-d` bits are set)
- Input: `bin/hw1 -p -e`. Return value: 0x00AA (`-p` and `-e` bits are 0. Both rows and columns default to 10).

- Input: `bin/hw1 -p -d -r 11`. Return value: 0x20BA (`-p` bit is 0. `-d` bit is 1, rows are 11 and columns default to 10).
- Input: `bin/hw1 -e -p -r 15 -c 15`. Return value: 0x0000. This is an error case because the argument ordering is invalid (`-e` is before `-p`).

Part 2: Polybius Cipher

Background

In this part, you will implement encryption and decryption using a Polybius cipher. This cipher uses an alphabet (i.e. list of characters) to construct a table. We will use the `polybius_alphabet` declared in `const.c`, which has ASCII characters 0x21-0x7e (33-126), to fill the table. You must use the `polybius_table` variable defined in `const.c` to hold this table.

The dimensions of the table are decided by command line arguments `-r` and `-c`. If either flag is not given, the corresponding variable defaults to 10. For this cipher to work correctly, the row and column labels need to be single characters, so we will use hex digits (0-9 and A-F).

After `validargs`, you have guaranteed that `(rows * cols) >= length of polybius_alphabet`. Fill in the table in row major order (i.e. elements in rows are stored next to each other in memory) with `polybius_alphabet` and fill the remaining space (if any) with null characters (`\0`).

For example, let's assume the user runs `bin/hw1 -p -e -r 11 -c 9`. The resulting table of 11 rows and 9 columns looks like this:

	0	1	2	3	4	5	6	7	8
0	!	"	#	$	%	&	'	(	)
1	*	+	,	-	.	/	0	1	2
2	3	4	5	6	7	8	9	:	;
3	<	=	>	?	@	A	B	C	D
4	E	F	G	H	I	J	K	L	M
5	N	O	P	Q	R	S	T	U	V
6	W	X	Y	Z	[	\	]	^	_
7	`	a	b	c	d	e	f	g	h
8	i	j	k	l	m	n	o	p	q
9	r	s	t	u	v	w	x	y	z
A	{	|	}	~	\0	\0	\0	\0	\0

In this table, '!' is at index (0, 0) and '~' is at index (A, 3). The last five spaces are null characters (indices (A, 4) - (A,8)).

> 🤔 It might seem like a 2D-array is needed for this, but it's not! All you need is a 1D-array of characters (i.e. a string, or `char*`). To do this, think about how you can convert a row/column pair to an index. For example, in the table above, `~` is at index (A, 3) in a 2D matrix. If we lay this out as a 1D array in row-major order, `~` would be at index 93. Think about how you can use the base address of the array to find the address of `~` .
>
> 🤓 Could you still convert row/column pairs to 1D indices if the table had `int` values rather than `char` values?

To encrypt a message, take each character from `stdin` and write its two- character position in the table to `stdout` . Continue until your program reads `EOF` (control-d in a UNIX system).

```
input: CSE320{}

// 'C' is at (3, 7)
// 'S' is at (5, 5)
// 'E' is at (4, 0)
// '3' is at (2, 0)
// '2' is at (1, 8)
// '0' is at (1, 6)
// '{' is at (A, 0)
// '}' is at (A, 2)

output: 375540201816A0A2
```

This encryption scheme can be improved by adding a key. The key consists of a subset of the characters in the alphabet and cannot have repeated characters. The characters in the key are placed first in the table and the remaining alphabet characters are placed after. For example, if the user runs `bin/hw1 -p -e -k cse320` to specify a key of `cse320` and a table of 10 rows and 10 columns:

	0	1	2	3	4	5	6	7	8	9
0	c	s	e	3	2	0	!	"	#	$
1	%	&	'	(	)	*	+	,	-	.
2	/	1	4	5	6	7	8	9	:	;
3	<	=	>	?	@	A	B	C	D	E
4	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y

	0	1	2	3	4	5	6	7	8	9
6	Z	[	\	]	^	_	`	a	b	d
7	f	g	h	i	j	k	l	m	n	o
8	p	q	r	t	u	v	w	x	y	z
9	{	|	}	~	\0	\0	\0	\0	\0	\0

```
input: CSE320{}
output: 3753390304059092
```

As you can see, the characters in `cse320` are first and all other characters are after. Using a key doesn't change the number of characters in the table (only changes the ordering of the alphabet) but it makes the cipher more secure.

During encryption, you must preserve whitespace.

> We will only consider the space, tab, and new line characters for whitespace. For example, using the same table as above:

```
input: CSE     320 {}
output: 375339     030405 9092
```

Invalid Characters

If you encounter a non-whitespace character that is not in `polybius_alphabet` during encryption, your program must stop encryption and exit with `EXIT_FAILURE` .

Sample Encryption Execution

NOTE: In the following examples, the program encrypts one line at a time and stops encrypting after it reads `^d` (control-d) from `stdin` . Entering `^d` into a terminal in a UNIX system signals an `EOF` (end of file) to the program.

> `EOF` is not actually a character in a file (i.e. it does not have a mapping in the ASCII table). It is a macro that signals that no more input can be read from a file or stream.

```
$ bin/hw1 -p -e -r 11 -c 9 -k cse320
CSE320 IS MY FAVORITE CLASS!!
415843030405 4758 5265 4438625457476043 41513858580606
#NOT@ALL
0853546037385151
C IS AWESOME!!!
```

```
41 4758 38634358545243060606
$ echo $?
0
```

> 🤓 `$?` is an environment variable in bash which holds the return code of the previous program run.

```
$ bin/hw1 -p -e -r 15 -c 7
CsE320 Is My FaVoRiTe ClAsS
46B551242321 55B5 62C4 529174B170A27295 46A544B571
$ echo $?
0
$
```

For decryption, you must reconstruct the same table used for encryption (i.e. the same number of rows and columns and the same key must be used). Then, the first two characters correspond to a row/column pair, which can be used to find the plaintext character in the table. For example:

```
command: `bin/hw1 -p -d -k cse320`
encrypted string: "375339     030405 9092"

// (3, 7) is 'C'
// (5, 3) is 'S'
// ...
// (9, 2) is '}'

plaintext: "CSE     320 {}"
```

Like encryption, the decryption process preserves whitespace. **You do not have to worry about invalid decryption files.** All test files for decryption will be the result of encryption, so they will not have invalid characters.

Sample Decryption Execution

```
$ bin/hw1 -p -d
348236181715 4082 4488 3764537849725168 3475328250
CsE320 Is My FaVoRiTe ClAsS
$ echo $?
0
$
```

Testing With Bash

There are a few different ways of testing your program. You can encrypt a file and verify manually, but this is tedious. You can also write criterion unit tests, but this only verifies that individual units are working properly (more on this later). A benefit of writing a command line utility that takes command line arguments is that you can test your entire program with bash. In the UNIX world, running command line tools on bash or another shell is extremely common. Many of these tools (e.g. `cat` , `echo` , `grep` , etc) are simple, but can be used in combination to write complex scripts.

The command below will allow you to encrypt a string and pass the result of the encryption to decrypt it.

```
$ echo "HeLLo WorlD" | bin/hw1 -p -e | bin/hw1 -p -d
$ HeLLo WorlD
```

To explain this command:

- `echo` takes a string and writes it back to `stdout`
- The first pipe (|) allows the the `stdout` of `echo` to become the `stdin` of the encryption program. This allows the middle command to encrypt the output of `echo`
- The second pipe allows the `stdout` of encryption program to become the `stdin` of the decryption program. Therefore, the decryption will decrypt the encrypted version of the original string. The result should be the original string.

We can take this a step further using `diff` :

```
$ diff <(echo "HeLLo WorlD") <(echo "HeLLo WorlD" | bin/hw1 -p -e | bin/hw1 -p -d)
$ echo $?
$ 0
```

- `diff` is a program that displays the difference between two files.
- `<(...)` is known as process substitution. It is allows the output of the
- program(s) inside the parentheses to appear as a file for the outer program.

Because both strings are identical, `diff` outputs nothing.

Finally, we can test the program on entire files with a similar command:

```
$ diff <(cat rsrc/stonybrook.txt) <(cat rsrc/stonybrook.txt | bin/hw1 -p -e -r 15 -c 8 | bin/
$ echo $?
$ 0
```

🤓 `cat` is a command that outputs a file to the console.

Part 3 - Fractionated Morse Cipher

Morse Code (https://en.wikipedia.org/wiki/Morse_code) is a standardized method of transmitting text information. The text is translated into various sequences of "dots" `.` and "dashes" `-`. Fractionated Morse Cipher (http://practicalcryptography.com/ciphers/classical-era/fractionated-morse/) takes this translated text and ciphers it using a key back into text that can only be decoded using that same key.

> It is advised that you read the linked web page on Fractionated Morse Cipher! We will be implementing the cipher similarly to the one displayed there.

`const.h` defines `char *morse_table[]` which will be our translation table when translating between text and Morse Code. To simplify lookup for translations, the `morse_table` array will be indexed based on the ASCII table starting with the character `!`.

For example: `morse_table[0]` will contain the translation for `!`, and `morse_table[32]` will contain the translation for 'A'.

Symbols that appear in the ASCII table between the characters `! (0x21)` and `z (0x7A)` that do not have a Morse Code encoding will have an empty string (`""`) at their index in `morse_table`

> This means that if a character without a valid encoding appears in the string to encode this is an **ERROR** and your program must exit with `EXIT_FAILURE`!
>
> You MUST use the `morse_table` array when translating to Morse Code. We **WILL** be changing `const.c` during grading, this means that the Dash and Dot combos assigned to ASCII characters will vary!

`const.h` also defines `char *fractionated_table[26]` which is the alphabet used in conjunction with the key to cipher the Morse Code. This array will **NEVER** change, and it will **ALWAYS** be associated with the indexes of the key, i.e., `fractionated_alphabet[0]` associates with `fm_key[0]`. As shown on the web page (http://practicalcryptography.com/ciphers/classical-era/fractionated-morse/), the triple `...` from the Morse Code ciphers to `R` with the key `ROUNDTABLECFGHIJKMPQSVWXYZ`.

The key for the cipher is a mixed alphabet determined by the `-k` argument. To show this we will explain how we determine the key with the `-k` argument being `ROUNDTABLE`. This `-k` argument will expand to the key being `ROUNDTABLECFGHIJKMPQSVWXYZ`. The letters in the `-k` argument will be the start of the key, and the rest of the key is filled out with the letters of the alphabet that are not in the `-k` argument. Note that `C` is the first character in the key after `ROUNDTABLE` due to both `A` and `B` appearing in the `-k` argument.

Our program will read from the input file, translate the given text to Morse Code, and then cipher the Morse Code translation to Fractionated Morse Cipher. In our morse translation, we will be putting one `x` (0x78) between each character and `xx` at the end of each word. We will be treating a word as any amount of characters followed by any amount of whitespace.

> Multiple spaces between words **MUST** be treated as a single space in translation.
>
> New Lines **MUST** be preserved in your translation and cipher.
>
> **DO NOT** read the entire input file into memory. Your program should function for files of any size (e.g. 1 byte to 1 terabyte!)

In the ciphering of the intermediate Morse translation, your program will cipher groups of three characters until there are no more groups of three characters left. This means that any leftover characters will be truncated off the end and not ciphered.

Fractionated Morse Cipher Example

In this section we will be ciphering the text `I LOVE CS` with a detailed explanation of each step. We will be using the key `HELP`.

We must first translate the text to Morse Code. **We will be using the translation table as defined in the `const.c` we have provided.**

`I LOVE CS` will be translated to `..xx.-..x---x...-x.xx-.-.x...xx`

`I` is mapped to the Morse Code `..` (dot dot). We then put the word divider `xx` to indicate that this is the end of the word. Next we translate `LOVE` as `.-..x---x...-x.xx`. As you can see, `L` translates to `.-..` and `O` translates to `---`. In between these characters, we have the letter divider, which is a single `x`. This routine is done for the rest of the text. To end the entire sequence, we have a terminating word divider.

Next we will determine the full key using the input value `HELP`. When we expand the key, it will become: `HELPABCDFGIJKMNOQRSTUVWXYZ`. This is because the full key is filled out with the rest of the letters of the alphabet that were unused.

Using the key, we will cipher the Morse Code. To do this, we must first associate the letters of the key with the Fractionated Table values. This is done in the following fashion:

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

The table shows each possible Morse Code triple cipher. `...` is ciphered to `H`, `..-` is ciphered to `E`, `..x` is ciphered to `L`, and so on.

> Take note that there is no `xxx` on the table!

We will split the Morse Code into triples, and then cipher the triples:

> 😱 **As a visual aid, the character `|` will be used to separate the triples in this example.**

`..x | x.- | ..x| --- | x.. | .-x | .xx | -.- | .xx`

will be ciphered to:

`L | T | L | M | S | B | F | I | F`

or just

`LTLMSBFICL`

To decrypt our message, we do the same steps in reverse. First we must decipher the encrypted message back into Morse Code. For this to properly be done, we must decipher using the same key so that our table is the same as when we encrypted the message. As a reminder, the table will be listed again:

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

Using the key we will build the original Morse Code translated message from the ciphered text, which is `..xx.-..x---x...-x.xx-.-.x...xx`

Next we must translate the Morse Code back into plain text. This step requires translating all of the dot-and-dash combinations to their plain text equivalents, and translating word dividers accordingly.

`..` will be translated to `I`. Since it is followed by a word divider, we will be adding a space after `I` to get `I `. Next, we translate `.-..` to `L`. Since it is simply followed by a letter divider we will ignore it and translate the next `---` to `O`. This routine is done for the rest of the phrase, and we are left with our original phrase `I LOVE CS`.

Fractionated Morse Cipher Encryption Program Usage

```
$ bin/hw1 -f -e [-k KEY_STRING]
```

Sample Encryption Execution:

Basic Encryption

Input File:

```
$ cat rsrc/message.txt
DEFEND THE EAST
```

```
$
```

```
$ cat rsrc/message.txt | bin/hw1 -f -e -k ROUNDTABLE
ESOAVVLJRSSTRX
$
```

Encryption with Multiple Lines

Input File:

```
$ cat rsrc/message2.txt
I LOVE C!
I LOVE 320!
$
```

```
$ cat rsrc/message2.txt | bin/hw1 -f -e -k FIXMYPROG
XQXENPGBOMH
XQXENPGFHIHEHBY
$
```

Encryption with Multiple Spaces between Words

Input File:

```
$ cat rsrc/message3.txt
I    LOVE     C   !
$
```

```
$ cat rsrc/message3.txt | bin/hw1 -f -e -k FROPY
OSOJQADGDGY
$
```

Encryption with character with no Morse Translation

Input File:

```
$ cat rsrc/message4.txt
2^64 IS A LARGE NUMBER
$
```

> In the `morse_table` that we provide in `const.c`, `^` is an invalid character.

```
$ cat rsrc/message4.txt | bin/hw1 -f -e -k DONTWERK
$ echo $?
1
$
```

> Remember that `$?` is the Bash variable that holds the return code of the last program that executed. In this case it is 1 because this is what `EXIT_FAILURE` is defined as for our VM.
>
> The output of a program when the input is invalid is arbitrary, and will not be tested.

Fractionated Morse Cipher Decryption Program Usage

```
$ bin/hw1 -f -d [-k KEY_STRING]
```

Sample Decyption Execution:

Basic Decryption

Input File:

```
$ cat rsrc/encoded_message.txt
ESOAVVLJRSSTRX
$
```

```
$ cat rsrc/encoded_message.txt | bin/hw1 -f -d -k ROUNDTABLE
DEFEND THE EAST
$
```

Decryption with Multiple Lines

Input File:

```
$ cat rsrc/encoded_message2.txt
XQXENPGBOMH
XQXENPGFHIHEHBY
```

```
$ cat rsrc/encoded_message2.txt | bin/hw1 -f -d -k FIXMYPROG
I LOVE C!
I LOVE 320!
```

> Like Polybius, you may always assume that the given files for Decryption will be valid!

Sample Execution using shell redirection

```
$ echo "HELLO, WORLD!" | bin/hw1 -f -e -k DEKU | bin/hw1 -f -d -k DEKU
HELLO, WORLD!
```

```
$ echo -e "HELLO\n\nMULTI\nLINE!" | bin/hw1 -f -e -k ALMIGHT | bin/hw1 -f -d -k ALMIGHT
HELLO

MULTI
LINE!
$
```

```
$ cat rsrc/message5.txt
REDIRECTION IS FUN!
```

```
$ bin/hw1 -f -e -k URAVITY < rsrc/message5.txt | bin/hw1 -f -d -k URAVITY
REDIRECTION IS FUN!
```

> Using `echo` with the `-e` flag causes it to interpret escape characters such as `\n` .

Unit Testing

Unit testing is a part of the development process in which small testable sections of a program (units) are tested individually to ensure that they are all functioning properly. This is a very common practice in industry and is often a requested skill by companies hiring graduates.

> Some developers consider testing to be so important that they use a work flow called test driven development. In TDD, requirements are turned into failing unit tests. The goal is to write code to make these tests pass.

This semester, we will be using a C unit testing framework called Criterion (https://github.com/Snaipe/Criterion), which will give you some exposure to unit testing. We have provided a basic set of test cases for this assignment.

The provided tests are in the `tests/hw1_tests.c` file. These tests do the following:

- `validargs_help_test` ensures that `validargs` sets the help bit correctly when the `-h` flag is passed in.

- `validargs_encrypt_test` ensures that `validargs` sets the Fractionated Morse bit correctly when the `-f` flag is passed in.
- `help_system_test` uses the `system` syscall to execute your program through Bash and checks to see that your program returns with `EXIT_SUCCESS` .

Compiling and Running Tests

When you compile your program with `make` , a `hw1_tests` executable will be created in your `bin` directory alongside the `hw1` executable. Running this executable from the `hw1` directory with the command `bin/hw1_tests` will run the unit tests described above and print the test outputs to `stdout` . To obtain more information about each test run, you can use the verbose print option: `bin/hw1_tests --verbose=0` .

The tests we have provided are very minimal and are meant as a starting point for you to learn about Criterion, not to fully test your homework. You may write your own additional tests in `tests/hw1_tests.c` . However, this is not required for this assignment. Criterion documentation for writing your own tests can be found here (http://criterion.readthedocs.io/en/master/).

Hand-in instructions

TEST YOUR PROGRAM VIGOROUSLY!

Something *fun to try*: message a classmate an encrypted message and see if they can decrypt it using their program, that will ensure that you are both implementing the program correctly.

Make sure your directory tree looks like this and that your homework compiles (i.e. you see a green check for passing CI in Gitlab):

```
hw1/
├── include
│   ├── debug.h
│   ├── hw1.h
│   ├── info.h
│   └── ... Any additional .h files you defined
├── Makefile
├── rsrc
│   └── ... Any sample text files given or created (will not used in graded)
├── src
│   ├── const.c
│   ├── hw1.c
│   ├── main.c
│   └── ... Any other .c files you defined
└── tests
    ├── testfile
    ├── validargs_tests.c
    └── ... Any additional criterion test files you may have written
```

This homework's tag is: `hw1`

```
$ git submit hw1
```

> 🤓 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!